Core-Local Reasoning and Predictable Cross-Core Communication with M³

Nils Asmussen*, Sebastian Haas*, Adam Lackorzyński†, Michael Roitzsch*
*Barkhausen Institut, Dresden, Germany and †TU Dresden, Germany
{nils.asmussen,sebastian.haas,michael.roitzsch}@barkhauseninstitut.org, adam.lackorzynski@tu-dresden.de

***Abstract*—Modern cyber-physical systems often require security, heterogeneity, and real-time operation from their hardware platform and operating system. However, highly predictable real-time operating systems such as FreeRTOS do not employ strong component isolation required for platform security. Microkernels implement such isolation using virtual memory and code running in the privileged CPU mode, complicating real-time analysis.**

In this work, we start with a different architectural approach: M³ is an existing hardware/software co-design for heterogeneous systems that features strong isolation between cores. However, the real-time properties of this platform have not been investigated. We first survey M³'s current state for real-time applicability and study both the communication latencies in comparison to other systems and M³'s different approach to task priorities. Furthermore we improve M³'s real-time applicability by adding a network-on-chip traffic regulation and enabling the enforcement of resource limits. With these additions, M³ enables *local reasoning* about application execution. We perform the evaluation with an FPGA-based hardware prototype and in simulation based on gem5.

***Index Terms*—real-time, system architecture, operating systems, message passing**

I. Introduction

Cyber-physical systems (CPSes) and Internet-of-Things (IoT) appliances often are real-time systems deployed in embedded devices that are connected to a cloud-based backend. These type of systems are already deployed in industrial production and are expected to become ubiquitous in use cases like personal robotics, health care, electrical grids, water supply, and transportation.

With such diverse use cases and operating environments, these devices need a platform which simultaneously addresses security concerns, hardware-level heterogeneity, and real-time requirements. Heterogeneity is introduced because the large-scale deployments and the energy-constrained nature of battery-powered devices demand energy-efficient operation. General-purpose processors can flexibly run arbitrary code, but they are often not the most energy-efficient solution to a problem. For specific workloads known ahead of time, it is beneficial to include a processor with specialized instruction-set extensions, or even a dedicated programmable or fixed-function accelerator like a graphics processor or a neural-network engine. Managing these diverse compute resources with a common set of systemwide abstractions is challenging [36], [18], [38], [31]. Many existing operating systems only treat CPU cores as first-class resources and accelerators as peripherals, leading to limitations like file-system access only being available for code on a CPU, but not for code on a GPU.

Since CPSes interact with their physical environment to fulfill control tasks, some workloads must adhere to timing requirements. To prove conformance, the software components serving such workloads as well as the underlying hardware must be predictable to facilitate a rigorous worst-case timing analysis. But this may not be needed for all running applications. Connecting the device to the cloud backend may be less timing-critical, leading to another dimension of system heterogeneity: platforms can combine a processor optimized for predictable execution like an Arm Cortex-R with a processor optimized for average-case performance like an Arm Cortex-A. The Cortex-R would also run a predictable software stack based on a real-time operating system like FreeRTOS, while the Cortex-A can run best-effort work on Linux. This separation allows to analyse the real-time part independently; a property we call *local reasoning*. Such platforms are gaining industry traction for systems in modern cars [2], [3].

Security is the third necessary property. Since CPSes can be deployed in critical infrastructure or other areas with potential harm to humans, they have to pay special attention to attacks by malicious actors. Security best practices teach us that attack prevention requires reducing the attack surface and constructing layered defenses. However, the reality is that FreeRTOS and other real-time operating systems offer no isolation[1] between activities, because MMU-based virtual address spaces and inter-process communication routed through privileged kernel code complicate timing analysis. This problem is worsened when messages leave the current core and are destined for another processor or accelerator. Such messages require signaling based on inter-processor interrupts (IPIs), where the delivery path has to traverse a kernel on both the sender and the receiver, before reaching the target application.

We thus observe a fundamental tension between systems with strong isolation between components, potentially running separated on heterogeneous cores, and the predictability and low latency of inter-component communication. Isolation requires the use of hardware features like MMUs and IPIs and a kernel running in the processor's privileged mode. However, these steps add jitter and complicate worst-case timing analysis. Microkernels improve inter-component messaging by implementing a low-complexity jitter-reduced kernel path, but still experience hardware-level jitter [21]. The obvious alternative of communication based on shared memory avoids

[1]We use the term 'isolation' in the security sense, whereas 'local reasoning' describes modularity of timing analysis. These concepts are orthogonal, with Linux and FreeRTOS being examples of having one without the other.

the kernel on the critical path, but complicates analysis by inviting interference from all applications on all processors accessing the same memory. Technologies like MemGuard [44] try to mitigate this problem, but interference on a globally shared resource remains difficult for real-time system design.

In this paper, we want to explore a different system architecture and its suitability for real-time CPSes. M³ [9] is an existing hardware/software co-design for heterogeneous systems. It is based on a tiled architecture, with strong isolation between tiles, enforced by a hardware component called Data Transfer Unit (DTU). This construction provides the basis for secure system construction, while offering an interesting alternative to traditional MMU- and kernel-based isolation. The follow-up work M³x [8] has demonstrated efficient use of accelerators, underlining the support for heterogeneous systems. M³v [7] added context-switching, an important puzzle piece for task scheduling, but the real-time properties of M³ have not yet been explored. Since previous work already demonstrated the applicability of M³ for secure and heterogeneous systems, we now evaluate communication latency and jitter and perform architectural changes to improve predictability. This paper is structured as follows and makes the following contributions:

1) After the background on our definition of system-level security and the M³ architecture in Section II, we survey and discuss the current state of M³ for its suitability to run real-time workloads in Section III. We identify local reasoning as a key M³ benefit: timing influences of applications can be analysed with tile-local knowledge.
2) We enhance M³ in Section IV by a traffic regulation for the on-chip network to enable analysability of cross-tile communication. The traffic regulation is enforced by the DTU, but configured based on M³'s capability system, extended by *capability derivation*. Furthermore, we improve M³'s energy efficiency by adding a kernel-less and therefore jitter-reduced core sleep feature.
3) In Section V, we first study M³'s current state on the FPGA-based hardware platform by evaluating the communication behavior in comparison to other systems and evaluating M³'s approach to task priorities.
4) Finally, we evaluate the effectiveness of our additions to enable local reasoning and to support a jitter-reduced core sleep on our gem5 prototype.

II. Background

Before introducing the M³ architecture, we explain the notion of system-level security as we use it within this paper.

A. Security and the Trusted Computing Base

The term security in the context of this paper should be understood as an effort to increase resiliency by a systematic reduction of the system's attack surface. This approach is different from other lines of research, which approach security concerns as special scheduler obligations, e.g. scheduling intrusion detection software [17], information flow prevention [28], or covert channel mitigation [39]. These notions are largely orthogonal to attack surface reduction, but within this paper, we exclusively use the latter.

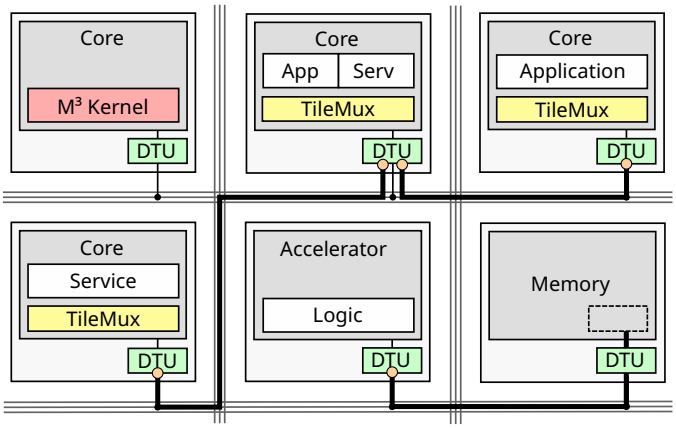


Fig. 1: System architecture of M³: one DTU per tile isolates tiles from each other and selectively allows communication as configured by the M³ kernel. TileMux multiplexes its tile among the applications on this tile.

Reduction of the attack surface is achieved by strict application of the principle of least privilege. Each component in the system should run with the minimal set of permissions necessary to fulfill its duties. In such a system, compromising a component by exploiting a vulnerability in its design or implementation results in granting the attacker the smallest possible amount of additional privileges.

The counter-example are monolithic systems like Linux, where large amounts of complex functionality are packaged into a single privileged kernel component. Therefore, one exploitable vulnerability there can directly lead to full system compromise. In contrast, microkernel-based designs encourage the separation of system functionality into strongly isolated components, thus presenting layered defenses against attacks. We explain in the following subsection how M³ extends this isolation approach from software to the hardware architecture.

Even in a microkernel-based system with isolated components, some parts of the system are relied upon by every application. This Trusted Computing Base (TCB) always contains those pieces of the system implementing inter-component isolation and communication.

B. The M³ Hardware/Software Platform

M³ [9] proposes a new system architecture based on a hardware/software co-design. On the hardware side, M³ builds upon a tiled architecture [43], as shown in Figure 1. M³ 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) and the tiles are connected via a network-on-chip (NoC). In contrast to conventional architectures, M³ 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* (thick black lines in the figure) needs to be established. Communication channels are represented as *endpoints* in the DTU (orange dots). 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 memory region without an endpoint on the memory side.

On the software side, M^3 runs its kernel (red) on a dedicated *kernel 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 M^3 kernel is allowed to establish such channels. By default, no communication channels exist and thus tiles are isolated from each other. Additionally, applications are placed on different tiles by default, but as shown by M^{3}v [7], tiles with general-purpose cores can also be shared efficiently and securely among multiple applications. For that reason, every core-based user tile runs a multiplexer called *TileMux* (yellow), which is responsible for isolating and scheduling the applications on its own tile, similar to a traditional kernel. However, in contrast to a kernel, each TileMux instance has no permissions beyond its own tile. Instead, only the M^3 kernel can make system-wide decisions, hence its name.

The following describes M^3's previous contributions in terms of security and heterogeneity.

1) Security: If the core or accelerator in a user tile wants to access tile-external resources (e.g., DRAM), it can only perform this access via the DTU in its tile. The M^3 kernel can control which resources each user tile can access by controlling its DTU [9]. This enables the M^3 platform to integrate potentially untrusted third-party components such as accelerators or modems into user tiles without risking that such components harm others or leak sensitive data. Furthermore, as tiles do not share any hardware structures (e.g., caches, TLBs, or branch target buffers), these structures cannot be used for side-channel attacks on applications within different tiles, nor on the kernel. In summary, with the M^3 platform, only the kernel tile, all DTUs, and the NoC are part of the TCB. Integrated third-party hardware components do not need to be trusted, which is a key differentiator of M^3 to contemporary MPSoC platforms.

2) Heterogeneity: M^{3}x [8] has shown how the M^3 platform enables better accelerator integration by allowing them to directly access OS services. 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 [29]). However, accessing files from accelerators or pipelining accelerators with software is challenging [36], [18], [38], [31]. M^{3}x showed how hardware accelerators can run 'autonomously' by connecting them directly with OS services or applications on general-purpose cores. This autonomous execution of accelerators also revealed significant speedups and reduced CPU utilization [8].

III. Discussion of M^3's Suitability for Real-time

This section surveys existing M^3 features and discusses how they can support real-time operation. Afterwards, we will present our enhancements to improve M^3's real-time guarantees.

A. Computation

Typical use cases within IoT or CPSes have both real-time tasks and best-effort tasks. An example is an IoT device that listens to a sensor to handle a control loop, but also wants to send the measured data to a cloud backend for further analysis. The sensor and control part might have strict real-time requirements and can also be safety-critical if humans could be harmed in case of missed deadlines or faulty operation. However, the network part is typically not time critical, but complex and therefore error prone and difficult to analyse regarding timing requirements.

Currently, there are two options to choose from: a single OS that supports both real-time and best-effort tasks or two separate and specialized OSes. For example, a single Linux kernel could run on all cores and real-time priorities could be used to ensure that deadlines are met. However, a Linux kernel that is shared among all cores requires that all cores are cache coherent. This in turn increases system jitter, leads to a more difficult timing analysis for the real-time tasks, reduces the energy efficiency, and increases the hardware costs. The shared kernel is also hampering the specialization of individual cores for real-time requirements or best-effort requirements.

The other option, that runs a best-effort OS next to a real-time OS (RTOS), enables the specialization of individual cores. For example, a set of Arm Cortex-A cores could run Linux for best-effort tasks and an Arm Cortex-R core could run an RTOS. These sets of cores do not need to be cache coherent, which avoids some problems of the shared-kernel option. However, missing cache coherency complicates the communication between the two OSes and applications cannot build upon a shared set of abstractions. Thus, collaboration between real-time tasks and best-effort tasks is challenging.

Due to its hardware/operating-system co-design, M^3 allows to combine the advantages of both options. On the hardware side, different tiles in the tiled architecture are not cache coherent and also do not share any hardware state (e.g., caches, branch-target buffers, TLBs). For that reason, the timing behavior of the computation on one tile is independent of the other tiles (ignoring communication for now, which we address in Section III-B). Furthermore, each tile has its own DTU which provides all tiles with a uniform interface. This uniform interface provides the foundation for common software abstractions and easy collaboration with activities on other tiles. The M^3 OS design leverages this uniform interface to allow tiles to contain different types of compute resources. For example, a best-effort tile could use an out-of-order processor to optimize for performance, whereas a real-time tile could employ a simple in-order processor that simplifies the timing analysis and improves the energy-efficiency.

On the software and operating-system side, M^3 runs a per-tile multiplexer (TileMux) on every tile. TileMux can be compared with a microkernel that is only responsible for the isolation between the applications on its tile and their scheduling. In

contrast to a microkernel, TileMux is not concerned with inter-process communication or access control. It also has no permissions beyond its own tile and can thus be specialized on a per-tile basis without affecting any other tile in the system. For example, the TileMux scheduler can be optimized for overall throughput on a best-effort tile and optimized for real time on another tile. Besides the per-tile multiplexer, M^3 runs a kernel on a dedicated tile, separate from the application tiles and only used explicitly by applications.

Based on the omni-present DTUs, activities on all tiles can communicate in a uniform fashion and rely on a set of shared abstractions. Namely, all tiles can leverage abstractions such as message-based communication, DMA-like memory access, file systems, pipes, and network stacks. The access to all of these abstractions is controlled via capabilities as mechanism (managed by the M^3 kernel) and an XML-based configuration file as policy (see Section IV-C). The configuration file describes the permissions of all activities in the system, regardless of whether they require best-effort or real-time scheduling.

In summary, we believe that M^3's system architecture enables per-tile (local) reasoning, because it simplifies timing analysis due to the absence of shared caches and OS interference, while still delivering a shared-system experience.

B. Communication

Besides the ability to specialize tiles for either best-effort or real-time activities, we deem it important that all activities can communicate with each other. Referring again to the example from the previous section, the sensor part of the system needs to communicate with the network part in order to send the measured data to the cloud backend. With M^3, communication between activities is performed via DTU and we will discuss three different aspects of this communication mechanism: communication latency, energy-efficient communication, and buffer-space management.

1) Low-latency Communication: Communication in IoT devices or CPSes is often time critical, requiring low latency between different activities in the system. For example, emitting a command to an actuator after receiving a sensor signal can be time critical to avoid physical damage or even to prevent harm to humans. On non-M^3 systems, communication between tiles is typically performed based on shared memory to exchange data, and interrupts for notifications. System calls are used to communicate with other applications, allowing the operating system to suspend applications, and thereby avoid polling to wait for communication partners. However, as we will show in our evaluation, both interrupts and kernel entries and exits are expensive.

M^3 with its DTU-based communication was therefore designed not to involve the kernel during communication. That is, although the M^3 kernel is required to setup communication channels, neither the M^3 kernel nor TileMux are involved in the actual communication. Instead, even untrusted applications can interact directly with the DTU to communicate over established channels. Since every tile has its own dedicated DTU, it is always immediately available for the running application. We show in Section V that avoiding the kernel (and interrupts) during communication reduces the latency by about one order of magnitude. Furthermore, DTU-based communication is asynchronous in the sense that the sender can always deliver its message into the receive buffer of the recipient and continue working, regardless of whether the recipient is running or whether it is ready to handle the message. These points in combination result in low and predictable communication latencies on the sender side.

On the receiver side, M^3 also avoids interrupts to further reduce the latency. If the recipient is already running, the DTU does not inject an interrupt. Only if a message targets a recipient that is not running, the DTU informs TileMux via interrupt to potentially switch to the recipient [7]. We observe a general trade-off between low-latency communication and latency guarantees. If interrupts are never used to notify the recipient about a message, best-case latency is reduced, but we rely on the recipient to check for the new message in time (the recipient might still be busy with other work when the message arrives). On the other hand, if interrupts are always used, the best-case communication latency is higher, but we have the chance to handle each message immediately.

2) DTU Interrupt Injection: The DTU injects an interrupt in case the ID of the currently running activity is not equal to the ID of the recipient activity. If a low-priority recipient shares a core with a high-priority application, any message destined for the low-priority activity can therefore cause a brief delay for the high-priority activity while TileMux is handling this interrupt. Such delays are not analysable, if the low-priority code can communicate at an arbitrarily high rate. This is a known problem for interrupt-based signaling [23].

We can offer two mitigations: One is the DTU's credit system, which allows recipients to choose how many credits their senders have and when they are refilled (see Section III-B3). But this mechanism requires global parameterization of message flows to control the local rate of interrupts, weakening the principle of local reasoning. The second mitigation option is inspired by Scheler et al. [34]: provision more than one core within the tile and handle low-priority TileMux interrupts on a core separate from the high-priority application. Whenever the high-priority application is running, the interrupt core will only update message counters in TileMux, but not deschedule or otherwise influence the high-priority execution. The interrupt core can also run low-priority work in its user mode, if desired. Messages intended for the high-priority application can still deliver interrupts directly to the application core.

We considered a solution using a single core and masking interrupt delivery. But we identified corner cases and therefore leave this investigation to future work.

3) Buffer Space Management: Finally, we want to discuss the question, how communication channels can be controlled and restricted. In contrast to other approaches, applications on M^3 communicate directly via DTU without involving the OS kernel (which would allow such control). Thus, without further measures, applications can mount denial-of-service (DoS) attacks on other applications by, for example, exhausting buffer space on the receiving end. Other means for DoS attacks like overwhelming the receiver's processing capacity

or overwhelming the NoC bandwidth are discussed later.

Message passing via DTU is performed between multiple senders, each owning a send endpoint, and a single recipient, owning a receive endpoint. Each pair of send endpoint and receive endpoint is called communication channel. The receive endpoint refers to a receive buffer in memory, which is split into multiple same-sized message slots. Thus, activities that send to the same receive endpoint share its receive buffer. To ensure that each activity gets a usable share of the receive buffer, we use a credit system. The recipient hands out credits to its senders, stored in each of the send endpoints. Sending a message requires at least one credit in the send endpoint and the DTU reduces the number of credits by one. In return, message delivery into a free buffer slot is guaranteed. If no credits are available, the DTU denies an attempt to send a message with an error (without blocking). Having received a message allows the recipient to reply on this message, similar to *reply capabilities* in L4 [19], [20], [40] and EROS [35]. When receiving a reply on a previously sent message, the number of credits at the sender is increased again. This scheme enables the recipient to control how much space each sender can occupy and to tell senders when more messages can be sent. Since owning one credit guarantees the successful delivery of one message, the credit system avoids useless traffic over the NoC. To prevent starvation, the DTU fetches messages in round-robin fashion from the receive buffer.

C. Services and the Kernel

All applications on M^3 are sharing the same M^3 kernel and multiple applications may share an OS service. However, in contrast to traditional OSes where interrupts and exceptions can occur at any time and lead to a mode switch from user mode to kernel mode, both OS services and the kernel run on dedicated tiles and are only called explicitly. For that reason, applications can control when they are subject to interference from these components, simplifying timing analysis. To streamline terminology, we use *server* to refer to both the kernel and OS services in this section.

Using a server requires including this remote execution in the application's timing analysis. However, such a remote call is equivalent to a non-preemptive critical section in traditional systems, for which analysis methods are available. The pattern applies recursively.

1) Client Priorities: An important feature for real-time systems is that servers can assign different priorities to their clients. The server can then prioritize the requests from more important clients to allow meeting their real-time requirements even in the presence of other potentially malicious clients. However, avoiding interference from low-priority to high-priority clients is challenging even in modern real-time-capable microkernels such as seL4-MCS [24] as shown by Mergendahl et al. [27]. Although seL4-MCS uses priority-sorted IPC endpoints, attackers can still cause unbounded delays for high-priority clients by triggering long-running and non-preemptive work within the microkernel. For example, attackers can trigger long insertion times into the sorted IPC-endpoint queue or trigger long scheduler activities when replenishing budgets. While these problems can probably be fixed by adding more complexity to the microkernel, M^3 sidesteps these problems altogether with its different system design.

The budget attacks described by Mergendahl et al. [27] are based on the shared scheduler by attacker and victim. This shared state does not exist on M^3, because each tile has its dedicated TileMux instance, which only knows and schedules the activities on its own tile. As tiles in M^3's hardware platform are not cache coherent and no shared software is running across multiple tiles, interference between activities can be avoided by simply placing them on different tiles.

The remaining attacks are based on a shared IPC endpoint. While multiple send endpoints can send to the same receive endpoint in M^3, message sending and receiving is done by the DTU in hardware and thus in parallel to the execution of software. Furthermore, servers are free to employ multiple receive endpoints and assign clients to them based on their priority. In contrast to existing approaches, this does neither need additional complexity in the kernel nor in applications if multi-threading is required to block on multiple IPC endpoints. Instead, per-priority endpoints are naturally supported, because applications communicate via DTU without involving the kernel and can either check for new messages on a specific endpoint or wait until the next message arrives (see Section IV-A). Our evaluation (see Section V-B) confirms that the worst-case request latency for high-priority clients is indeed bounded on M^3, independent of requests from low-priority clients. Note that the server can learn about client priorities from the system configuration, which we describe in Section IV-C, preventing clients from lying about their priority.

2) CPU-time Restrictions: Besides this infrastructure for servers to assign different priorities to clients, we see one open challenge to further improve the real-time usability M^3 can offer for applications. In case of the M^3 kernel or a shared OS services, the CPU time that these servers require to handle requests from their clients needs to be shared among all these applications. Although the DTU's credit system prevents DoS attacks and the DTU fetches messages in round-robin order to prevent starvation, there are currently no upper bounds enforced on the spent CPU time. For example, a malicious client could send requests that take a long time to process on the server side, preventing other clients from being serviced during that time. One solution would be to let servers track the CPU time spent for each client and limit the allowed time per request. The server can then check for each request whether it can be handled within the time limit and deny the request otherwise. If the request can be handled within the limit, but the remaining time budget of the client does not suffice, the server can delay the request.

However, this problem of client cross-talk occurs in any system with shared resources. The established solution is to consider during timing analysis the maximum time the shared resource could be serving another client. Consequently, real-time applications should only invoke services which can guarantee bounded service time. Current M^3 service implementations do not always guarantee such a bound. Most prominently, also the kernel does not. Contrary to other systems, real-time applications on M^3 can communicate without kernel involvement. Therefore, applications can be constructed to use the kernel only during setup

and consequently can ignore the kernel for their timing analysis.

IV. Enhancements to M³'s Real-time Guarantees

After the discussion on how M³'s current state can be used to run real-time workloads, we now describe our concrete technical improvements. We start with a contribution towards lower communication latency, but without sacrificing energy efficiency. The subsequent two subsections about network-on-chip and resource limits describe our changes to enable or improve the analysability and predictability of M³.

A. Fast and Energy-efficient Communication

Besides low-latency communication, energy efficiency is important for IoT devices and CPSes. These goals can be contradicting, though. For example, low latency can be achieved by communicating via shared memory and polling, but only at the expense of energy efficiency. Conversely, communicating via blocking system calls enables the OS kernel to put the core to sleep, but increases the overall communication latency.

To achieve both goals, we extended the DTU and M³. Our approach is comparable to `umonitor`/`umwait` on x86-64, but applied to the DTU. The idea is to put the core to sleep while waiting for a message to arrive. To that end, the DTU offers a sleep command that puts the core to sleep and wakes the core up again as soon as a message arrived. A similar concept exists on the MIPS 34K processor family, where computation can be blocked in the load/store unit when accessing a special memory mapped device used for IPC.

However, without further measures, the DTU could have already received a message and may therefore sleep forever in case no further message arrives. The DTU solves this race condition by atomically checking whether new messages exist for the current activity and only if this is not the case, putting the core to sleep. Like `umonitor`/`umwait`, the sleep command can be used by unprivileged and potentially untrusted applications. To prevent indefinite sleeps in case no message is received, the DTU also provides a simple timer that is programmed by TileMux to fire at the end of the current activity's time slice.

In summary, the DTU sleep command avoids kernel entry and exit, because it is usable by unprivileged applications. With this command, applications can wait for new messages in an energy-efficient way without increasing the communication latency. Adding this feature via the DTU avoids modifications to the core. However, we did not yet investigate the general trade-off between short wakeup times and deep sleep states. The current implementation in our gem5-based prototype uses clock gating to put the core to sleep.

B. Network on Chip

The NoC constitutes the only hardware resource globally shared by all tiles whenever they communicate. As such, it is a point for undesired cross-application disturbance, which must be restricted to be analysable. Apart from the aforementioned credit system, which is designed to manage buffer space, but not NoC bandwidth, such restrictions were absent in M³ and we added them in this work.

1) Methods for NoC Analysis: In order to make the minimal changes necessary to enable real-time NoC analysis, we first review existing analysis methods. A very coarse-grained NoC delay bound can be obtained using the lumped link model [46]. Independent of actual topology, the NoC is modeled as an exclusive bus. All direct and indirect contention is treated the same and is over-approximated, so while this analysis is lightweight, it leads to pessimistic bounds.

The M³ NoC can preempt message transfers at the granularity of 512-byte chunks and delivers them using deadlock-free wormhole routing [30], for which tighter analysis methods are available. Rahmati et al. [33] have shown for such NoCs, that even best-effort networks without any traffic regulation have non-infinite delay bounds. Intuitively, since the network is deadlock-free, every message will arrive eventually. However, considerably lower bounds can be obtained when traffic regulation is in place at the sender side. The typical regulation primitive [33], [12], [25] is a token-bucket traffic shaper at the point(s) where NoC clients insert traffic into the network.

M³ has the structural advantage that all NoC clients are always connected to the NoC via a DTU. Thus, the DTU is the natural place to add such regulation for ingress traffic. Each token-bucket shaper is parameterized by a size limit and a fill rate. A message can be emitted into the NoC, when an amount of tokens equivalent to the size of the message in bytes is available in the bucket. Since a timer facility (needed to refill tokens) is already available within the DTUs to implement other functionality, the token bucket regulation does not add a lot of complexity to the DTUs.

For a concrete application scenario, the bucket parameters have to be configured such that the NoC delay bounds obtained using existing analysis methods result in an overall schedulable system. Finding the right parameters and performing the analysis is a problem orthogonal to the M³ architectural details discussed here. We therefore refer to other works like the analysis by Boyer et al. of the Kalray MPPA2 NoC [12].

Finally, we note that using the NoC link capacity, an upper bound for message delay can be converted to a lower bound for NoC bandwidth allocated to a client. Thus, the simple addition of token-bucket regulation to the DTUs leads to tighter delay bounds as well as guaranteed bandwidth.

2) Implementation of Token-Bucket Regulation: As outlined earlier, the credit system is enforced by the DTU on a per-endpoint basis, because the receive buffer exists per receive endpoint and serves as the shared resource for all send endpoints that refer to it. However, the NoC as a global resource is shared by all endpoints, allowing a more coarse-grained regulation. Existing analysis methods restrict entire tiles, but we decided to regulate traffic on a per-activity basis. Regulating individual activities can be useful when the schedule of multiple activities on different tiles is known.

To support per-activity NoC regulation, we equip the DTU with multiple *token-bucket registers* and refer to one of these registers via specific bits in the register holding the currently running activity (`CUR_ACT`). The token-bucket registers are only accessible by the M³ kernel, whereas `CUR_ACT` is set by TileMux when switching between activities on its tile. This design allows differ-

ent parameters for different activities, but TileMux is only trusted regarding activity selection. TileMux itself runs with a special activity ID and is thus itself affected by the NoC regulation.

Each token-bucket register consists of a `rate`, an `amount`, and a `limit`. `amount` represents the number of bytes that can immediately be sent without being delayed. A NoC message is delayed if `amount` is smaller than the number of bytes being sent. Otherwise, `amount` is reduced by that number of bytes. And finally, `amount` is refilled according to the configured `rate` until `limit` is reached. We evaluate the effectiveness of this approach in Section V.

3) Memory Bandwidth: In M^3, the NoC is also the only means by which tiles can access off-tile storage like main memory (DRAM). All memory accesses are based on memory endpoints, which are either used explicitly or implicitly. Applications can use memory endpoints explicitly to issue DMA requests to the memory the endpoint provides access to. Applications use memory endpoints implicitly when accessing memory via load and store instructions. These implicit accesses are part of the physical-memory protection (PMP) feature that the DTU already provides [7]. PMP defines the accessible physical memory for a tile with a set of memory endpoints. This allows the software to, for example, provide each tile access to a separate part of the physical memory. Last-level cache misses are routed through the DTU and are only passed through to the NoC if they are permitted according to PMP.

In summary, both DMA requests and ordinary memory accesses involve the DTU, which automatically divides these transfers into chunks. Each chunk is subject to the bandwidth restriction given by the current activity's token-bucket register before being emitted to the NoC.

C. Resource Limits

Another important aspect in IoT devices and CPSes is the management and restriction of resources. The available resources of the platform such as memory or CPU time have to be distributed to applications and services, depending on their requirements. In particular, the system needs to be able to enforce limits on the use of resources to prevent an application from mounting a DoS attack on another application by allocating all available resources. The existing M^3 prototype uses capabilities to exchange access permissions, but had no facility to distribute initial capabilities and no way to enforce limits (e.g., all applications could allocate an arbitrary amount of physical memory). This section describes our extensions to integrate these resource limits with M^3's existing capability system and its delegation mechanisms.

We use a layered approach to control the access to resources. The first layer is a policy layer that manages and distributes resources to services and applications. As a second layer, these resources are translated into capabilities, so there can be a fine-grained exchange of resources between services and applications. The third layer is responsible for enforcing the restrictions that have been set for individual applications. For performance reasons, this enforcement is performed locally at the component responsible for the resource.

1) Configuration Files and Resource Managers: In more detail, the first layer uses XML-based *configuration files* to describe the distribution of resources and *resource managers* that read this configuration file and manage the resources accordingly. The overall idea is to use one configuration file per use case of the system (each boot expects exactly one configuration file), which describes the services and applications that should be started including their accessible resources and limits.

There is always at least one resource manager, but nesting of resource managers to create subsystems is supported. Each resource manager receives the subset of the configuration file it's responsible for and starts the therein requested services and applications. Each child additionally receives a communication channel to its resource manager. Children use this channel to request access to resources, which the resource manager checks against the configuration file. In case access can be granted, it creates a capability for the resource and passes it to the child.

2) Capabilities: These capabilities are handled by the second layer. That is, capabilities are managed in software by the M^3 kernel, which offers system calls to create, exchange, and revoke capabilities. After receiving an initial capability from the resource manager, applications can pass on the capability to other applications, provided that communication channels to the applications exist.

We extended the capability system to support *derivation* of capabilities. Derivation creates a new capability with a specified subset of a quota or permissions of an existing one. For example, the tile capability holds multiple quotas for tile-local resources such as CPU time and NoC bandwidth. Starting an activity on a tile requires a tile capability and uses the attached quotas for this activity. For that reason, if activity A has a tile capability with 1s CPU time and 4 GB/s NoC bandwidth, activity A can derive from this tile capability and thereby move 0.5s of CPU time and 3 GB/s of NoC bandwidth to a new tile capability. Afterwards, a new activity B can be started with this derived tile capability, which will be constrained to these resource limits. The tile capability of activity A has been reduced by the derive to 0.5s CPU time and 1 GB/s NoC bandwidth.

3) Enforcement of Restrictions: The third layer is responsible for enforcing restrictions on resource accesses. These restrictions are based on quotas and are enforced locally by the component that embodies the resource. Local enforcement is important to achieve good performance, because it avoids the involvement of other components during access time. For that reason, resource limits are enforced by different components, depending on the resource. For example, TileMux is responsible for enforcing limits on CPU time, whereas the DTU enforces limits on the NoC bandwidth via the token-bucket registers. Despite the local enforcement by different components, all these limits are specified in the configuration files and passed to applications in the form of capabilities, which enables a fine-granular exchange and refinement of these limits.

V. Evaluation

Our evaluation[2] attempts to answer the question of whether M³ is suitable for real-time scenarios. We start with a comparison of different communication primitives to demonstrate the advantages of M³'s hardware-supported communication mechanism. Afterwards we study the effectiveness of per-priority endpoints on M³. Furthermore, we analyse the ability of local reasoning with the current M³, followed by an evaluation of how this is improved by our NoC-regulation addition. Finally, we measure the latency of the DTU's sleep feature. Note that we do not repeat application-level benchmarks performed with previous M³ prototypes [9], [8], [7] (showing that M³'s performance is competitive to Linux) as our additions do not change the results.

A. Communication

Our goal is to keep the advantages of M³ in terms of heterogeneity and security and extend it to be also suitable for real-time scenarios. For that reason, applications are typically placed on different cores for either security benefits (preventing side-channel attacks) or efficiency benefits (two cores that have been specialized for different workloads). Suitability for real-time scenarios additionally requires communication primitives with low latency and in particular low jitter. Therefore, we first study these properties for M³'s cross-core communication mechanisms in comparison to other such mechanisms. Note that we do not consider polling-based communication mechanisms due to their low energy efficiency.

Since M³ is a hardware/software co-design, it needs specific hardware and currently only runs in simulation and on an FPGA-based platform. As the tiles of the FPGA platform are not cache coherent, no operating system other than M³ currently runs on multiple tiles on this platform. Thus, a comparison with other communication primitives on hardware is difficult. We therefore start with M³ on the FPGA platform and move to other hardware platforms by using gem5 as an intermediate step. Note also that we compare entire hardware/software platforms and not only OSes and thus run each OS in its natural way on the corresponding hardware platform.

On all platforms, we measure the round-trip latency of the corresponding cross-core mechanism available on the platform. Both the message that is sent from sender to receiver and the reply that is sent from receiver to sender already exists in memory. Both messages contain 1 byte of payload. Note that we studied larger message sizes as well, but found that the performance of all communication primitives scales linearly. Therefore, we only show results for 1-byte messages to focus on latency. We perform 1000 runs without warmup and show the average latency and standard deviation in Figure 2 after removing outliers to study the typical runtime performance. We determine outliers as in box plots: results that fall below $Q1-1.5IQR$ or above $Q3+1.5IQR$. Additionally, we study the communication jitter with Table I, which includes outliers and shows the average, minimum, and maximum latency and the standard deviation. To abstract over different clock frequencies on different platforms, we show latency in cycles.

[2]The artifact is available on Zenodo: `10.5281/zenodo.10798062`.

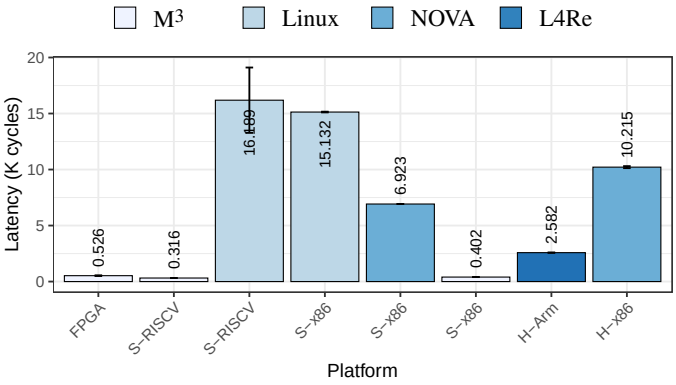


Fig. 2: Round-trip latency for cross-core messaging on different hardware platforms and OSes (excluding outliers).

OS	Platform	avg	P99	min	max	σ
M³	FPGA	537	675	484	3571	107
M³	S-RISCV	319	316	316	3460	99
Linux	S-RISCV	16234	24824	11152	36578	3042
Linux	S-x86	15317	22773	10529	35112	1335
NOVA	S-x86	7058	7017	6919	130181	3899
M³	S-x86	405	416	377	3347	93
L4Re	H-Arm	2605	2639	1622	22739	644
NOVA	H-x86	10261	10442	9958	55724	1408

TABLE I: Round-trip latency for cross-core messaging on different hardware platforms and OSes (including outliers).

1) M³ on FPGA Platform: The FPGA platform of M³ is implemented on the Xilinx Virtex UltraScale+ FPGA (VCU118 board), contains eight processing tiles with a single RISC-V core each and two memory tiles with interfaces to external DDR4 DRAM. All tiles are connected by a NoC using a 2x2 star-mesh topology. The platform uses Rocket cores [6] and BOOM cores [45] as the RISC-V cores. 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. The clock frequencies of the Rocket and BOOM cores are set to 100 MHz and 80 MHz, respectively, to fully meet timing requirements during FPGA synthesis and place-and-route. The M³ kernel runs on a Rocket core, whereas all benchmarks in this evaluation run on BOOM cores.

The benchmark runs the sender and receiver on two different tiles with BOOM cores and both leverage the DTU to exchange messages. As shown in Figure 2 (excluding outliers), the average latency (526 cycles) and standard deviation (32 cycles) are comparatively low. This stems from the fact that applications on M³ can directly access a dedicated hardware component to exchange messages without requiring hardware interrupts or involving the operating system. Table I shows that the standard deviation is slightly higher when including outliers (107 cycles) and the maximum latency measured is 3571 cycles (in the first communication due to cache misses).

2) Comparison with Linux on gem5: To put these results into perspective, we now evaluate the cross-core communication performance on other OSes. We start with Linux (version 5.11 with side-channel mitigations disabled), because it is well optimized, runs on RISC-V cores and is in widespread use even in the real-time community. Since Linux cannot leverage multiple tiles on the FPGA platform, we compare M³ and Linux on gem5 [10]

using RISC-V cores. We configure gem5 to use a CPU clock frequency of 2 GHz, a memory frequency of 1 GHz, and the out-of-order CPU model. M^3 uses a single core per tile, each containing 32 KiB L1 instruction cache, 32 KiB L1 data cache and 512 KiB L2 cache. Linux uses two cores with dedicated L1 caches in the same size and a shared 1 MiB L2 cache. Note that we use 32 KiB for the L1 caches instead of 16 KiB as on the FPGA, because it is more typical in real hardware and benefits Linux. All cores can access a shared DRAM using gem5's DDR3 1600 8x8 model.

On Linux, we use a message queue to send 1-byte messages back and forth between two processes (we also tried pipes, but obtained worse performance). On M^3, the messages are exchanged via DTU, exactly as on the FPGA. As shown in Figure 2 for the 'S-RISCV' platform ('S' for simulator), M^3 achieves an average latency of 316 cycles, whereas Linux achieves 16189 cycles on average. The large difference stems not only from the complexity of Linux, but also from the two required IPIs and kernel involvements. The results for M^3 differ from the results on the FPGA platform due to the different core implementations, but are still within the same order of magnitude. The results including outliers in Table I show that Linux exhibits more jitter than M^3 and has a high latency during the first run (36578 cycles). Furthermore, even when ignoring outliers as done in Figure 2, we experience a rather large standard deviation (2920 cycles) for Linux on RISC-V.

3) Comparison with NOVA on gem5: Since Linux is not specifically optimized for message passing, we now compare M^3 to NOVA [40], one of the fastest microkernels. This comparison is done on x86-64, because NOVA does not run on RISC-V. We therefore leverage the described gem5 platform again with the same settings, but with the x86-64 instruction set architecture (ISA). As a comparison, we also run Linux again on gem5 with the x86-64 ISA. Figure 2 and Table I show that M^3 exhibits comparable results for 'S-x86' as for 'S-RISCV'. On 'S-x86', NOVA achieves a latency of 6920 cycles on average, whereas Linux achieves 15132 cycles on average. Both NOVA and Linux have a high latency for the first communication of 130181 cycles and 35112 cycles, respectively. Therefore, since NOVA is optimized for message-passing performance, the latency is significantly lower than on Linux, but NOVA still suffers from the same problems: IPIs and kernel involvement.

4) Comparison with Existing Hardware Platforms: After the measurements with gem5 as an intermediate step, we now perform the benchmark on real hardware. At first, we run the benchmark on NOVA again using an Intel Core i3-8100 CPU with 3.6 GHz, four physical cores without hyper-threading, and frequency scaling disabled. The results are shown as 'H-x86' ('H' for hardware). As can be seen in Figure 2 and Table I, the performance and jitter is comparable with NOVA on 'S-x86', but with slightly worse performance on hardware, similar to M^3 on 'FPGA' versus 'S-RISCV'/'S-x86'.

Finally, we run the benchmark on a heterogeneous R-Car Gen 4 Arm-based SoC, which provides hardware for interrupt delivery with payload, making it similar to the DTU's message passing feature. The SoC contains two different kind of Arm systems, an Arm Cortex-A cluster and an Arm Cortex-R cluster, connected

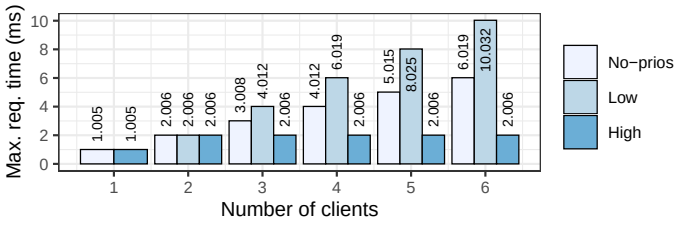


Fig. 3: Worst-case request latency with and without usage of priorities, depending on the number of clients.

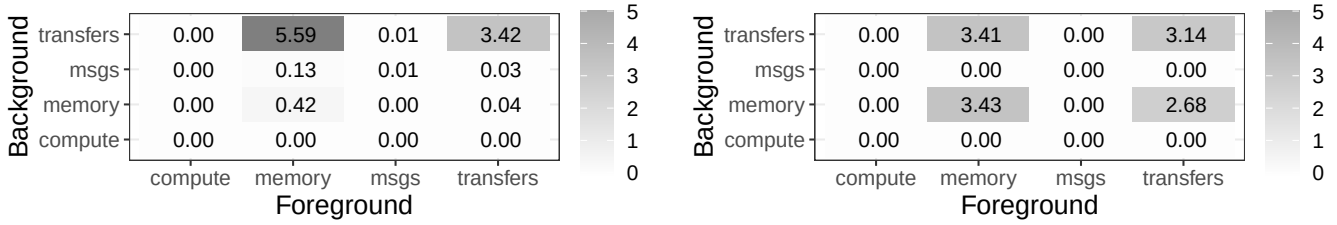
through the SoC's interconnect. The Cortex-A cores use a 64bit MMU-based architecture, running at a clock speed of 1.8 GHz, while the Cortex-R cores use a 32bit MPU-based architecture, running at 1.0 GHz. We run L4Re [20] on this device because L4Re can run on both architectures. For communication and notification we use the SoC's MFIS hardware allowing to send IRQ notifications from one side to the other as well as transferring data of up to 32 bytes. The communication latency for a round-trip, originating from a Cortex-A core, and using IRQ-based wakeup on both sides, is depicted in Figure 2 and Table I. Messages that are larger than 32 bytes can be transferred via shared memory.

Note that we do not compare to FreeRTOS [1], because it does not support cross-core communication out of the box. Extensions exist [4], but are based on shared memory and interrupts, similar to L4Re above. We therefore expect slightly better results than with L4Re, because FreeRTOS requires no kernel entry/exit, but at the expense of weaker isolation.

5) Summary: Since there is no hardware platform that runs M^3, NOVA, L4Re, and Linux, a direct comparison of their communication mechanisms is difficult, although partially possible in simulation. However, as the differences between these systems is about one order of magnitude, we still conclude that M^3's DTU-based communication mechanism is faster and has lower jitter. These advantages are primarily caused by the DTU as a dedicated hardware component that allows to exchange messages between tiles without requiring OS involvement. That raises the question of whether Linux can be adapted to make use of the DTU for inter-process communication. However, as this requires significant changes to the Linux kernel, we leave this study for future work.

B. Communication Latency with Per-Priority Endpoints

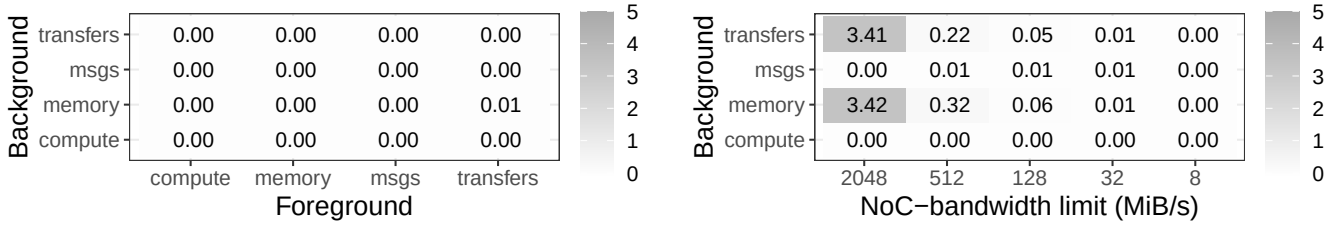
As explained in Section III-C1, M^3 sidesteps the challenges experienced for communication by other microkernel-based systems by using a per-tile scheduler (TileMux) and communicating via DTU. To verify whether high-priority clients can indeed be serviced in a bounded time, we run a server on a dedicated tile and one or more clients on the remaining tiles. The server uses two receive endpoints (for low and high-priority clients) and requires a fixed amount of time (1ms) to handle every request. In the first configuration, all clients get assigned the same priority, which we use as a baseline. The second configuration assigns the first client a high priority and the remaining clients a low priority. We run this benchmark on the FPGA with up to 6 clients, occupying all tiles.



(a) Disturbance on FPGA prototype

(b) Disturbance on gem5 simulator

Fig. 4: Disturbance of different foreground workloads on the FPGA by different background workloads on other tiles, shown as relative slowdown, without any NoC regulation.



(a) Disturbance with a NoC limit of 8 MiB/s

(b) Disturbance for memory accesses

Fig. 5: Disturbance of different foreground workloads on gem5 with NoC regulation. The left shows the slowdown for different foreground workloads and a NoC-bandwidth limit of 8 MiB/s, whereas the right shows the slowdown for memory accesses and different NoC limits.

The results are depicted in Figure 3, which shows the worst-case request latency observed by the clients during 1000 runs after 10 warmup runs. With just a single client the maximum time is about 1ms. As expected, without priorites ("No-prios") every additional client increases the maximum request time by 1ms. When using priorites, the latency for low-priority clients ("Low") increases with the number of clients. In contrast, the high-priority client ("High") observes request times of at most 2ms independent of the number of other clients, because it needs to wait for at most one already running client request. Thus, M^3 does indeed provide an upper bound for high-priority clients.

C. Local Reasoning without NoC Regulation

Although the results for high-priority clients are encouraging, we were wondering what other means low-priority clients have to influence high-priority clients on other tiles. We start with the FPGA platform, which does not support NoC regulation, to study the current state of cross-tile influence with M^3. Afterwards, we evaluate the ability of the introduced NoC regulation to reduce this influence based on the gem5 platform. We consider four different workloads: 1) computation, 2) memory accesses, 3) cross-tile message passing, and 4) DMA-based memory transfers. The computation workload is not performing memory accesses or other cross-tile interactions and should therefore have no influence on other tiles. The memory-access workload repeatedly copies data between two locations ensuring that the data does not fit into the tile-local cache (causing NoC traffic). The cross-tile message-passing workload exchanges messages with the maximum size supported by the DTU (2 KiB) with another activity on another tile. And finally, the memory-transfer workload issues read and write DMA requests to a 8 KiB large memory region in DRAM and is therefore causing a lot of NoC traffic as well.

We run all combinations of these workloads to study their influence on each other. That is, for each pair of workloads, we measure the time for the 'foreground workload' while running as many 'background workloads' on the remaining tiles as possible. Additionally, we run the foreground workload without background workloads to get a baseline. The results in Figure 4a show the slowdown of the foreground workload when running background workloads relative to the case when running the foreground workload alone. We always show the maximum slowdown experienced over 100 runs after 1 warmup run. As expected, the computation workload causes no slowdown for other workloads. Similarly, message passing has only a small effect on other workloads. Memory accesses have a rather small effect on other workloads as well, interestingly. However, memory transfers have a large effect on other workloads that use the NoC such as memory accesses, which are slowed down by a factor of 5.59.

To prepare for the study of our added NoC-regulation mechanism, we also run the same benchmark on the gem5 prototype platform without any NoC regulation. As can be seen in Figure 4b, the effects are similar, but not the same. Most importantly, memory accesses have a much larger influence on other tiles than on the FPGA platform. We suspect that the cause is the low clock frequency of the soft-cores (80 MHz and 100 MHz) on the FPGA, which does not suffice to put significant stress on the DRAM. This is in contrast to the gem5 cores, which are clocked with 2 GHz. Additionally, we observe no interference between message passing and other workloads. This is because the NoC in gem5 is currently modeled as a crossbar, leading to a dedicated path between each pair of tiles. However, this only affects the message-passing workload since both transfers and memory accesses compete for the shared DRAM.

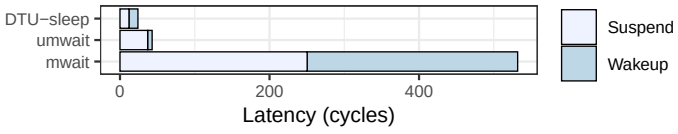


Fig. 6: Latency of DTU sleep, `umwait`, and `mwait`.

D. Local Reasoning with NoC Regulation

After analysing the cross-tile influence with M³'s current state, we now study the token-bucket regulation for the NoC that we add with this paper (see Section IV-B2). We implemented the NoC regulation in the gem5-based prototype with four token-bucket registers. Therefore, each DTU allows up to four different NoC regulations. To evaluate the effectiveness of the token-bucket regulation, we run the same benchmark as in Section V-C again, but enable the NoC regulation with different NoC-bandwidth limitations for the background workloads (using a single token-bucket register per tile and therefore per background workload).

As shown in Figure 5a, reducing the available NoC bandwidth for the background workloads to 8 MiB/s suffices to limit the slowdown of the foreground workload to at most 1 percent. To evaluate the influence depending on the NoC-bandwidth limit, we focus on the memory-accesses foreground workload, because it experienced the most severe slowdown in Section V-C. Figure 5b shows how the NoC bandwidth of the background workloads can be tuned to achieve the desired trade-off between less disturbance for the foreground workload and more available NoC bandwidth for the background workloads.

In summary, the added NoC regulation allows M³ to limit the usage of the remaining shared resources (e.g., NoC and DRAM). This enables tile-local reasoning, because low and guaranteed latency bounds for a given tile can be achieved even if other tiles perform message passing, memory accesses, or memory transfers.

E. Application-Controlled Core Sleep

Finally, we evaluate the DTU's sleep command, which allows applications to put the core into a sleep mode while waiting for incoming messages. This feature is important to avoid OS involvement during communication and still be energy efficient. We therefore evaluate its latency to suspend and wakeup the core in comparison to `umonitor/umwait` on x86-64 due to their similarities (see Section IV-A). Additionally, we compare to the use of `monitor/mwait` via a syscall as the best case for an OS-based sleep. We evaluate the DTU's sleep command on M³ and the other on Linux, both with the x86-64 version of gem5 using the same configuration as before. Since gem5 only supports `monitor/mwait`, we simulate `umonitor/umwait` by also executing it within a syscall but excluding the entry and exit costs of the syscall.

Figure 6 shows the average latencies of 100 runs after 20 warmup runs. As can be seen, the wakeup and suspend latency of both the DTU sleep and `umwait` is negligible in contrast to the syscall latency required for `mwait`. Note however that gem5 does not simulate different sleep states, which typically have different wakeup latencies depending on their depth.

VI. Related Work

With the presented results, M³ explores the intersection of real time, security, and heterogeneity. This paper focusses on real time, so we present related work in this area from the software and the hardware domain.

A. Real-Time System Software

System-level software deployed on commodity hardware architectures ranges from low-level real-time executives over microkernels up to complex solutions based on Linux. Executives such as FreeRTOS [1] achieve high predictability by deliberately not using CPU privilege separation between kernel and user modes. Instead, applications run with close to bare-metal access to the hardware. However, this comes at the price of losing inter-component protection based on address spaces. On its tiles, M³ also enables bare-metal computation without kernel mode, but does not need to sacrifice on security, because isolation is implemented outside the tile without impacting the on-tile computation.

Microkernels proficiently use address-space isolation to structure the system software into isolated components. M³ can optionally use address spaces within individual tiles, but the primary isolation is provided by the DTUs. M³ shares many traits with microkernel systems such as the philosophy of reducing the system TCB. However, different microkernels offer added value for real-time systems: seL4 offers formally verified implementation correctness and spatial isolation [19], as well as a timing analysis of all code paths through the kernel [11]. Fiasco.OC introduced the separation of execution and scheduling contexts [41] to run services on time budget provided by clients, a feature that M³ currently lacks. Fiasco.OC also features a system configuration mechanism by Lua scripts, which influenced our design for M³ resource configuration. The hierarchical resource refinement was inspired by the recursive system structure of the Genode OS [15].

Composite [32] and TOROS [13] are two microkernel approaches with a strong focus on predictability and security. Composite features user-level scheduling for maximum flexibility regarding the scheduling policy. M³ allows to customize scheduling per tile by deploying custom instances of TileMux. TOROS is intended to offer only analysable primitives at its application development interface. The existing M³ service implementations have yet to be surveyed for analysability. Similar to the hardware-based NoC regulation in M³, Chaos [16] proposes rate limiting for inter-processor interrupts in software based on user-level proxy servers.

Linux can also be used as foundation for real-time systems, especially with its deadline-based scheduler [22] that allows for convenient application development [37]. Decoupling of threads [21] can combine the convenience of Linux with the predictability of an underlying microkernel.

B. Real-Time Hardware Solutions

The DTU acts like a message-passing accelerator, which is directly accessible by applications from user mode. The Shrimp multi-computer [5] similarly proposes message passing from user mode, but uses address spaces instead of physical tiles for isolation. Tilera is a tile-based manycore architecture with processor extensions for tile-to-tile messaging [43]. DLibOS [26]

takes advantage of this feature by placing a network stack and applications on separate tiles and using the low-latency messaging extension to significantly improve network latency and throughput. Instead of extending the hardware of the cores, the DTU in M³ is core-independent, which is beneficial for security (no trust in the core implementation) and heterogeneity (accelerators can be equipped with a DTU as well).

For NoC delay predictability, we employ token-bucket regulation like other NoC implementations [12], [33] and works on real-time Ethernet [25]. In comparison to the Kalray MPPA2 NoC [12], the M³'s hardware platform allows to isolate different tiles from each other, allowing for example to integrate untrusted cores or accelerators. That is, even if a core is broken, the architecture can protect other cores from the broken core. The NoC regulation in M³ also applies to memory requests, making it similar in spirit to MemGuard [44]. However, while MemGuard implements coarse-grained regulation based on observing application's memory bandwidth using performance counters and descheduling them when their budget is depleted, limiting memory requests in hardware within the DTU allows for more fine-grained accounting down to single-byte granularity. Modifying the memory controller [42] is an alternative, but in M³, this would add another layer of regulation after the traffic already traversed the NoC. Instead, BRU [14] proposes to add new hardware between cores and DRAM to implement memory bandwidth regulation and Arm system interconnects feature quality-of-service regulation. These solutions are topologically most similar to M³'s DTU-based approach, but lack the security isolation of the DTU.

VII. Conclusion

We have analysed the real-time properties of the M³ hardware/software co-design platform. Resulting from the lack of hard-to-control globally shared resources, we have identified *local reasoning* about execution behavior as a key benefit. In order to enable analysis of the network-on-chip, we have added token-bucket based traffic regulation to the data transfer units (DTUs) of M³. Our FPGA and gem5-based evaluation has shown that M³ has competitively low message latency and jitter when communicating across cores within separate tiles. The traffic regulation limits cross-application disturbance from messages and memory accesses. With these results, we claim that M³ is a suitable cyber-physical platform, combining security, heterogeneity, and real-time operation.

VIII. Acknowledgements

This research is funded by the European Union's Horizon Europe research and innovation program under grant agreement No. 101092598 (COREnext). We thank the anonymous reviewers of RTAS'23, ECRTS'23, and RTAS'24.

References

[1] Freertos. https://www.freertos.org. viewed October 24, 2022.

[2] R-car-s4: Automotive system-on-chip (SoC) for car server/communication gateway. https://www.renesas.com/us/en/products/automotive-products/automotive-system-chips-socs/r-car-s4-automotive-system-chip-soc-car-servercommunication-gateway. viewed August 8, 2022.

[3] S32g2 processors for vehicle networking. https://www.nxp.com/products/processors-and-microcontrollers/s32-automotive-platform/s32g-vehicle-network-processors/s32g2-processors-for-vehicle-networking:S32G2. viewed August 8, 2022.

[4] Simple multicore core to core communication using freertos message buffers. https://www.freertos.org/2020/02/simple-multicore-core-to-core-communication-using-freertos-message-buffers.html. viewed October 24, 2022.

[5] R. Alpert, C. Dubnicki, E.W. Felten, and K. Li. Design and implementation of NX message passing using Shrimp virtual memory mapped communication. In *Proceedings of the 1996 International Conference on Parallel Processing*, volume 1 of *ICPP'96*, pages 111–119, Aug 1996. doi:10.1109/ICPP.1996.537151.

[6] 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. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[7] Nils Asmussen, Sebastian Haas, Carsten Weinhold, Till Miemietz, and Michael Roitzsch. Efficient and scalable core multiplexing with M³v. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 452–466. ACM, February 2022. doi:10.1145/3503222.3507741.

[8] Nils Asmussen, Michael Roitzsch, and Hermann Härtig. M³x: Autonomous accelerators via context-enabled fast-path communication. In *USENIX Annual Technical Conference (ATC)*, pages 617–632. USENIX, July 2019.

[9] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 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)*, pages 189–203. ACM, March 2016. doi:10.1145/2872362.2872371.

[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, u 2011. URL: http://doi.acm.org/10.1145/2024716.2024718, doi:10.1145/2024716.2024718.

[11] Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *32nd Real-Time Systems Symposium (RTSS)*, pages 339–348. IEEE, November 2011.

[12] Marc Boyer, Benoît Dupont de Dinechin, Amaury Graillat, and Lionel Havet. Computing routes and delay bounds for the network-on-chip of the kalray mppa2 processor. In *European Congress on Embedded Real Time Software and Systems (ERTS)*, January 2018. URL: https://hal.archives-ouvertes.fr/hal-01707911.

[13] Björn Brandenburg. The case for an opinionated, theory-oriented real-time operating system. In *1st International Workshop on Next-Generation Operating Systems for Cyber-Physical Systems (NGOSCPS)*, April 2019.

[14] Farzad Farshchi, Qijing Huang, and Heechul Yun. BRU: Bandwidth regulation unit for real-time multicore processors. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 364–375. IEEE, April 2020.

[15] Norman Feske. Introducing Genode. In *Free and Open Source Software Developers' European Meeting. (FOSDEM)*, February 2012.

[16] Phani Kishore Gadepalli, Gregor Peach, Gabriel Parmer, Joseph Espy, and Zach Day. Chaos: a system for criticality-aware, multi-core coordination. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 77–89. IEEE, 2019.

[17] Monowar Hasan, Sibin Mohan, Rakesh B. Bobba, and Rodolfo Pellizzoni. Exploring opportunistic execution for integrating security into legacy hard real-time systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 123–134. IEEE, November 2016. doi:10.1109/RTSS.2016.021.

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

[19] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4:

Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 207–220, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org/10.1145/1629575.1629596, doi:10.1145/1629575.1629596.

[20] Adam Lackorzynski and Alexander Warg. Taming subsystems: Capabilities as universal resource access control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, IIES'09, pages 25–30, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org/10.1145/1519130.1519135, doi:10.1145/1519130.1519135.

[21] Adam Lackorzynski, Carsten Weinhold, and Hermann Härtig. Predictable low-latency interrupt response with general-purpose systems. In *13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pages 19–24, June 2017.

[22] Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline scheduling in the linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016.

[23] Luis E. Leyva-del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 14–23. IEEE, April 2006. doi:10.1109/RTAS.2006.34.

[24] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–16, 2018.

[25] Jork Löser and Hermann Härtig. Low-latency hard real-time communication over switched ethernet. In *16th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 13–22. IEEE, July 2004. doi:10.1109/EMRTS.2004.1310992.

[26] Stephen Mallon, Vincent Gramoli, and Guillaume Jourjon. DLibOS: Performance and protection with a network-on-chip. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'18, pages 737–750, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3173162.3173209, doi:10.1145/3173162.3173209.

[27] Samuel Mergendahl, Samuel Jero, Bryan C Ward, Juliana Furgala, Gabriel Parmer, and Richard Skowyra. The thundering herd: Amplifying kernel interference to attack response times. In *28th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 95–107. IEEE, June 2022.

[28] Sibin Mohan, Man Ki Yoon, Rodolfo Pellizzoni, and Rakesh Bobba. Real-time systems security through scheduler constraints. In *26th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 129–140. IEEE, July 2014. doi:10.1109/ECRTS.2014.28.

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

[30] Sadia Moriam and Gerhard P. Fettweis. Fault tolerant deadlock-free adaptive routing algorithms for hexagonal networks-on-chip. In *Euromicro Conference on Digital System Design (DSD)*, pages 131–137. IEEE, August 2016. doi:10.1109/DSD.2016.71.

[31] Vincent Nollet, Paul Coene, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. 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*, page 174. IEEE Computer Society, 2003. doi:10.1109/IPDPS.2003.1213320.

[32] Gabriel Parmer and Richard West. Predictable interrupt management and scheduling in the Composite component-based system. In *Real-Time Systems Symposium (RTSS)*, pages 232–243. IEEE, December 2008.

[33] Dara Rahmati, Srinivasan Murali, Luca Benini, Federico Angiolini, Giovanni De Micheli, and Hamid Sarbazi-Azad. Computing accurate performance bounds for best effort networks-on-chip. *IEEE Transactions on Computers*, 62(3):452–467, March 2013. doi:10.1109/TC.2011.240.

[34] Fabian Scheler, Wanja Hofer, Benjamin Oechslein, Rudi Pfister, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 167–174. ACM, October 2009.

[35] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. Eros: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, 1999.

[36] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. Gpufs: integrating a file system with gpus. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, pages 485–498. ACM, 2013. doi:10.1145/2451116.2451169.

[37] Till Smejkal, Jan Bierbaum, Manuel von Oltersdorff-Kalettka, and Michael Roitzsch. CABAS: Real-time for the masses. In *15th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, July 2022.

[38] Hayden Kwok-Hay So and Robert W. Brodersen. 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*, pages 567–570. IEEE, 2008. doi:10.1109/FPL.2008.4630010.

[39] S.H. Son, R. Mukkamala, and R. David. Integrating security and real-time requirements using covert channel capacity. *IEEE Transactions on Knowledge and Data Engineering*, 12(6):865–879, 2000. doi:10.1109/69.895799.

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

[41] Udo Steinberg, Jean Wolter, and Hermann Härtig. Fast component interaction for real-time systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 89–97. IEEE, September 2005.

[42] Prathap Kumar Valsan and Heechul Yun. MEDUSA: a predictable and high-performance DRAM controller for multicore based embedded systems. In *3rd IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 86–93. IEEE, August 2015.

[43] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27:15–31, 10 2007. URL: doi.ieeecomputersociety.org/10.1109/MM.2007.89, doi:10.1109/MM.2007.89.

[44] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64. IEEE, April 2013. doi:10.1109/RTAS.2013.6531079.

[45] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. May 2020.

[46] Shobana Balakrishnan Füsun Özgüner. A priority-driven flow control mechanism for real-time traffic in multiprocessor networks. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):664–678, 1998. doi:10.1109/71.707545.