MCTI: Mixed-Criticality Task-based Isolation

Abstract

The ever-increasing demand for high performance in the time-critical, low-power embedded domain drives the adoption of powerful but unpredictable, heterogeneous Systems-on-Chip. On these platforms, the main source of unpredictability—the shared memory subsystem—has been widely studied, and several approaches to mitigate undesired effects have been proposed over the years. Among them, performance-counter-based regulation methods have proved particularly successful. Unfortunately, such regulation methods require precise knowledge of each task's memory consumption and cannot be extended to isolate mixed-criticality tasks running on the same core as the regulation budget is shared. Moreover, the desirable combination of these methodologies with well-known time-isolation techniques—such as server-based reservations—is still an uncharted territory and lacks a precise characterization of possible benefits and limitations. Recognizing the importance of such consolidation for designing predictable real-time systems, we introduce MCTI (Mixed-Criticality Task-based Isolation) as a first initial step in this direction. MCTI is a hardware/software co-design architecture that aims to improve both CPU and memory isolations among tasks with different criticalities even when they share the same CPU. In order to ascertain the correct behavior and distill the benefits of MCTI, we implemented and tested the proposed prototype architecture on a widely available off-the-shelf platform. The evaluation of our prototype shows that (1) MCTI helps shield critical tasks from concurrent non-critical tasks sharing the same memory budget, with only a limited increase in response time being observed, and (2) critical tasks running under memory stress exhibit an average response time close to that achieved when running without memory stress.

1 Introduction

Time-critical embedded systems are at the center of a transformational paradigm shift. Traditional embedded systems characterized by simple microarchitectures with well-defined and predictable application workloads are being phased out at an accelerating rate. Complex and architecturally demanding applications are taking their place, supported by sophisticated Multi-Processor System-on-Chip (MPSoC) with advanced microarchitectures,

heterogeneous memory subsystems, and general- and special-purpose accelerators. Notable examples of MPSoCs pushing the boundaries of horizontal scaling to support high-performance embedded applications include the NVIDIA Drive AGX Orin Anandtech (2019) and the Xilinx Versal Xilinx (2023). They integrate up to 12 high-performance ARM cores, GPUs, accelerators, and a large FPGA on a single low-power chip.

On these platforms, it is extremely challenging to ensure timing guarantees, high utilization, and the absence of interference among safety-critical workloads where applications have different levels of assurance. In fact, to optimize size and power consumption on such platforms, the memory subsystem (DRAM, memory controllers, and interconnect) is shared among cores and accelerators, and while accessing memory, high-critical applications are exposed to unpredictable interference from low-critical applications executing on different cores.

Partitioning hypervisors in mixed-criticality systems. In the context of mixed-critical systems, the need to integrate real-time workloads on a single MPSoC, such as robotic applications with stringent real-time demands, alongside best-effort applications like Linux-based logging or communication systems, has prompted the adoption of hypervisors. Hypervisors have been successfully used in industrial safety-critical contexts to isolate independent workloads with different criticalities (*e.g.*, SYSGO (2023)), as well as in the research community due to their ability to enable heterogeneous Quality-of-Service (QoS) and to seamlessly enforce common real-time policies across its guest OSs (*e.g.*, Martins et al (2020)).

Memory regulation via performance counters. Among these policies are performance-counter-based memory regulation techniques (PMC-regulation), which have been proposed over the last decade to control (or at least mitigate) the degree of inter-core interference and shield high-critical applications (tasks) from less critical ones. PMC-regulation implements *core-level* regulation and limits the maximum bandwidth a CPU can request at a millisecond-scale granularity and up to a microsecond-scale (Zuepke et al (2023)). *Memguard* by Yun et al (2016) has received considerable attention as it can be fully implemented in software and only relies on widely available standard performance counters (PMC) (*e.g.*, Sohal et al (2020); Yun et al (2017); Modica et al (2018); Schwaericke et al (2021); Zuepke et al (2023)). Hypervisor-based PMC-regulation has been implemented at both research and industrial levels (Modica et al (2018); Dagieu et al (2016); Green Hills Software (2023)) to extend hypervisors' isolation capabilities to MPSoC's memory subsystem. Implementing PMC-regulation at the hypervisor level is a logical choice as it makes PMC-regulation transparent to the OS level, allowing the use of potentially different operating systems (*e.g.*, general purpose and real-time) while ensuring proper control of memory bandwidth.

Bandwidth-based CPU provisioning. At the same time, when consolidating complex applications with mixed-criticality requirements onto high-performance MPSoCs with light Real-Time Operating Systems (RTOS) and

rich Operating Systems (OS)—such as Linux—CPU provisioning remains a fundamental aspect. Here, server abstractions (*e.g.*, the Constant Bandwidth Server, or CBS for short) (Abeni and Buttazzo (1998)) are well-known and widely used, with the `SCHED_DEADLINE` (Lelli et al (2016)) policy being the most popular example among researchers. Note that even if this work considers only CBS, thanks to the hypervisor, our architecture would allow different OSs to use different types of CPU server regulation.

The challenge of joint CPU and memory budgeting. Combining OS-transparent memory regulation with CBS-based task isolation is desirable. Thus, we envision a mixed-criticality real-time system where server-based provisioning is enacted at the OS level while PMC-regulation is employed to mitigate main memory contention across multiple CPUs at the Hypervisor level. However, the interplay between server-based CPU scheduling strategies and PMC-regulation has received little attention. As we discovered, the lack of coordination between the two mechanisms leads to poor handling of *memory overload* conditions. These conditions correspond to all scenarios where, despite being eligible for scheduling at the OS level, high-criticality tasks are blocked by the memory bandwidth regulation implemented in the hypervisor.

This article explores this issue in detail and presents a prototype architecture called "Mixed-Criticality Task-based Isolation" (MCTI) that aims to address the research gap related to memory overload conditions. The architecture has been developed to leverage existing techniques and is specifically designed to ensure the isolation of high-criticality tasks in mixed-criticality soft real-time systems. In summary, our main contributions are:

- The characterization of the interplay between OS-level CPU regulation and hypervisor-level PMC-regulation.
- The identification of problematic memory overload conditions that might prevent critical tasks from being executed despite having a sufficient CPU budget.
- The proposal of the MCTI protocol to overcome such scenarios and the design of a practical architecture to evaluate the protocol's behavior.
- The implementation of MCTI architecture on a commercial platform.
- An extensive evaluation of the proposed architecture with a detailed analysis of its pros and cons. Notably, the evaluation shows that the prototype provides, on average, the same response time as if the task is running in isolation.
- A detailed discussion on the difficulty of adequately configuring the prototype. Using the existing tools available to the community, we outline its current limitations.

In the remainder of the paper, Sec. 2 and Sec. 3 introduce the necessary background and propose a motivating example. Sec. 4 presents the system model. Sec.5 presents MCTI architecture and discusses the challenges of the analysis. Sec. 6 describes a concrete implementation of MCTI architecture, Sec. 7 and

Sec. 8 evaluate the implementation and discuss benefits and trade-offs. Sec. 9 presents related works, while Sec. 10 concludes the paper.

2 Background

MCTI architecture leverages an integrated HW/SW design whose main concepts are briefly presented below.

Server-based reservation Servers abstractions are well-studied reservation mechanisms to ensure isolation among tasks with different criticalities in the time domain. In this paper, we focus on the Constant Bandwidth Server (CBS) as formulated by Abeni and Buttazzo (1998) and use its Linux Kernel implementation by Lelli et al (2016) (`SCHED_DEADLINE` policy). This policy guarantees that the contribution of each server to the total utilization of the system is constrained by the fraction of CPU time assigned to each server, even under the presence of (time) overloads.

PMC-regulation Likewise, PMC-regulation ensures isolation among CPU cores[1] in the memory domain. We focus on software-based techniques originating from Memguard Yun et al (2016) that have been successfully evaluated in previous studies by Yun et al (2017) and Kim and Rajkumar (2016). These techniques rely on broadly available performance counters to regulate the bandwidth generated by each CPU. MCTI leverages CPU-level PMC-based isolation realized at the hypervisor level and implemented within Jailhouse. Specifically, MCTI relies on a publicly available prototype implementation of Jailhouse[2] that integrates a Memguard-based regulation Yun et al (2016), and that has been adopted in several previous works from Schwaericke et al (2021); Sohal et al (2020); Tabish et al (2021).

Cache-Partitioning Isolation of workloads deployed on CPUs sharing a last-level cache (LLC) can be achieved using cache-partitioning techniques. The objective is to ensure that addresses of independent tasks (or CPUs) are assigned to different cache sets and cannot interfere by evicting one another's cache lines. *Cache-coloring* is a well-studied software-based methodology that realizes cache-partitioning at the operating system (OS) or hypervisor level via manipulation of the virtual to physical address translation (*e.g.*, Mancuso et al (2013); Kim and Rajkumar (2016); Kloda et al (2019)). MCTI leverages the cache-coloring implementation available in the prototype implementation of Jailhouse used for PMC-regulation.

PS/PL Architectures The increasing commercial availability of heterogeneous MPSoCs (such as Xilinx's UltraScale+ Xilinx (2022), Intel's Stratix Intel, Corp. (2016), Microsemi's PolarFire Microsemi — Microchip Technology Inc. (2020)) that tightly integrate traditional Processing Systems (PS) with a Programmable FPGA-based Logic (PL) has led to novel paradigms in the management of the interconnect between CPUs and main memory. In the Programmable Logic in the Middle (PLIM) introduced by Roozkhosh

[1] In this article, we use the terms "CPU" and "CPU core" interchangeably.
[2] https://github.com/rntmancuso/jailhouse-rt

and Mancuso (2020), the PL-side is not simply used as a recipient for hardware accelerators but as an intermediate step on the data path linking the CPUs and DRAM, enabling fine-grained inspection and control on every single memory transactions. Scheduler In the Middle (SchIM) by Hoornaert et al (2021) follows a similar approach to re-order CPU-originated transactions and enforce a given memory-transaction scheduling policy. As discussed in Sec. 5, MCTI extends SchIM by enabling a criticality-triggered dynamic control of the memory-transaction scheduling policies.

3 Interplay of CBS and PMC-regulation

Under PMC-regulation, each CPU is regulated by a server-like mechanism and is assigned a maximum memory *budget* that is periodically replenished. If the budget is exhausted, the CPU remains idle until the memory budget is recharged. The budget value is determined using PMCs that monitor (directly or indirectly) the memory transactions performed by the CPU. For example, the number of last-level cache refills performed by the CPU is often used as a proxy for the extracted main memory bandwidth.

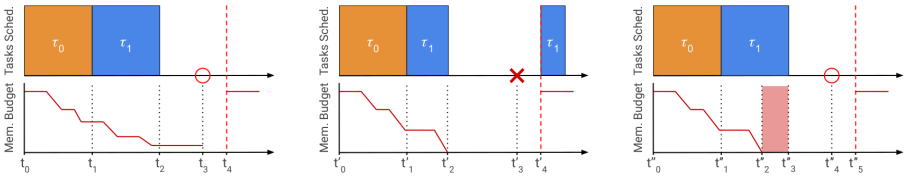
Unfortunately, the desirable isolation properties of PMC-regulation cannot be extended to applications with different criticalities running on the *same* CPU since only *one single* bandwidth threshold can be defined for each CPU. For PMC-regulation, this limitation is unavoidable and directly rooted in the capability of current performance counters (PMCs). Worse, this limitation adds to the technical difficulties of precisely characterizing the memory behavior of complex (preemptive) tasks executing on RTOSs.

To date, to cope with this limitation, architects of safety-critical systems have adopted designs that statically isolate criticalities across CPUs. Although beneficial from an analysis and certification point of view, these designs cannot leverage the full potential of MPSoC platforms as they must strictly separate high-critical and low-critical tasks. This makes partitioning and priority assignment more difficult and amplifies memory bottleneck problems for low-critical tasks forced to share the same CPUs.

Combining CBS-based CPU scheduling and PMC-regulation to achieve isolation in *both* time and memory domains is a logical choice. Enacting the former at the OS level and the latter at the hypervisor level aims to reap the benefits of a multi-layered architecture[3]. However, this approach results in a lack of coordination between the two mechanisms. This leaves the system incapable of handling *memory overload conditions* where the early depletion of CPU-bound memory budget prevents a (critical) task from completing its execution despite still having CBS-computation budget (see Sec. 4).

To better understand the key issues that MCTI addresses, consider the conditions depicted in Fig. 1. Task τ_0 is a low-criticality task, while τ_1 is a high-criticality one. In the time domain, both tasks are scheduled using CBS

[3] see Section 1



(a) Standard PMC-regulation, normal case.

(b) Standard PMC-regulation, memory budget depletion.

(c) MCTI regulation, with interconnect policy switch.

Fig. 1: Example scenario of a PMC-regulated CPU, where an increased memory consumption causes τ_1 to miss its deadline.

regulation that absorbs variations of the execution time of τ_0 without impacting τ_1 (Abeni and Buttazzo (1998)). In the memory domain, the bandwidth is regulated with PMC-regulation. Being a scarcer resource than CPU time, memory budgets are assigned based on the standard memory behavior (*e.g.*, obtained via profiling) of the tasks executing on a CPU to avoid under-utilizing memory. Deviations from the normal behavior are accounted for using fixed *safety margins*, which is a common practice in industrial applications.

Under normal conditions (Fig. 1a), τ_0 completes its execution at t_1, and τ_1 receives sufficient memory budget in the interval $[t_1, t_2]$ to meet its deadline at t_3. Instead, in Fig. 1b, in response to a change in the input type, τ_0 consumes more memory budget.[4] Note that the deadline of τ_0 is earlier than the one of τ_1, and τ_1 starts executing at t'_1. Although the time interval $[t'_1, t'_3]$ would be sufficient for τ_1 to complete on time, at t'_2 the memory budget of the CPU is depleted, and τ_1 must wait until t'_4 to resume execution, thus missing its deadline.

The key idea behind MCTI is to prevent τ_1 from being suspended if it still has sufficient time budget to complete its execution, even if the memory budget of the CPU has already been depleted. Indeed, this condition corresponds to a *memory overload*. As depicted in Fig. 1c, at time t''_2, MCTI detects that a high-criticality task is running and switches (at hardware level) the fairness-based default memory policy of the interconnect to prioritize memory traffic coming from the CPU where τ_1 is running. Thus, τ_1 can complete its execution in $[t''_2, t''_3]$ and meet its deadline at t''_4.

Although Fig. 1 represents a pathological case, due to the difficulties in precisely estimating the time and memory behavior of applications, these conditions can occur in practice for seemingly well-understood workloads. For example, this is the case for vision-based algorithms (Venkata et al (2009)) operating on input vectors with identical sizes but different content semantics. As shown in Fig. 8 and discussed in Sec. 7, selecting a single and safe regulation bound is impossible without severely under-utilizing the system.

[4] To simplify the explanation, we ignore that τ_0 might also take longer to complete its execution.

Table 1: Notation summary

Category	Notation	Description
Architecture	m	Amount of general purpose CPUs available
	CPU_k	k^{th} CPU of the system ($k \in [1, ..., m]$)
	l_{max}	Highest level of criticality handled by the system
Task	n	Amount of sporadic mixed-criticality real-time tasks
	Γ	Set of sporadic mixed-criticality real-time tasks
	τ_i	i^{th} task of the system's task set Γ ($\tau_i \in \Gamma$, $i \in [1, ..., n]$)
	C_i	τ_i worst case execution time
	D_i	τ_i relative deadline
	T_i	τ_i period
	l_i	τ_i criticality ($l_i \in [1, \ldots, l_{max}]$)
Server	S_i	i^{th} task CBS server ($i \in [1, \ldots, n]$)
	Qc_i	S_i computation budget
	P_i	S_i computation period
	U_i	S_i utilization ($U_i = Qc_i/P_i$)
PMC-reg.	Qm_k	CPU_k memory budget (memory transactions; $k \in [1, ..., m]$)
	M	Memory regulation period
	B_k	CPU_k memory bandwidth ($B_k = \frac{Qm_k}{M}, k \in [1, ..., m]$)
Interconnect	π	Run-time configurable interconnect policy ($\pi \in \{Fair, FP\}$)
	R_k	Bus priority of CPU_k when $\pi = FP$

Interestingly, when considered alone, the individual regulation mechanisms employed by MCTI are not sufficient to achieve the same degree of isolation and flexibility. (1) Perhaps the most straightforward solution would be to over-provision the per-CPU memory bandwidth. (2) On the other hand, statically prioritizing CPUs when they access main memory (*e.g.*, Hoornaert et al (2021)) might lead to starvation for the low-priority CPUs and prevent them from running non-critical, memory intensive tasks entirely. (3) Dynamically switching the bus priority depending on the criticality level of the running tasks defeats the isolation properties of PMC-regulation and might prevent low-critical tasks from running when the system is not subject to memory overload.

4 System Model and Regulation Policy

Overall System. The system comprises m general-purpose CPUs. Each CPU_k ($k \in \{1, \ldots, m\}$) has a private L1 instruction and data cache and shares a unified L2 cache (last-level cache—LLC) with all other CPUs. Cacheable memory is managed by the LLC using a write-back, write-allocate policy, and a pseudo-random replacement policy. The main memory features a single DRAM controller with an interleaved multi-bank configuration. Any access to the LLC resulting in a miss creates a read transaction toward the DRAM controller and the attached DRAM.

Tasks, Task set, and Partitions. We consider a set Γ of sporadic mixed-criticality real-time tasks. Each task $\tau_i \in \Gamma, i \in \{1, \ldots, n\}$ is defined by a tuple $\langle C_i, D_i, T_i, l_i \rangle$, where C_i is the worst-case execution time, T_i is the minimum inter-arrival time, D_i is the (arbitrary) deadline, and l_i is the criticality level. l_i conveys how critical a task is in terms of, for example, certification-related assurance level (RTCA Inc. (2011)). We assume $l_i \in \{0, \ldots, l_{max}\}$, where l_{max} is the highest criticality level and 0 means the task is not critical. At any instant t, the deadline of the task running on CPU_k is given by the function $D_k(t)$ ($D_k : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$), while the criticality of the task running on CPU_k at time t is given by the function $L_k(t)$ ($\mathbb{R}_0^+ \rightarrow \{0, \ldots, l_{max}\}$). At time t, a *critical task* τ_i has $L_k(t) > 0$. The tasks (Γ) are *partitioned* among CPUs in task sets noted Γ_k, and their execution on each CPU is controlled by a CBS server. Each task τ_i is associated with a server S_i. Each server $S_i, i \in \{1, \ldots, n\}$ is characterized by a tuple $\langle Qc_i, P_i \rangle$, where Qc_i is the computation budget and P_i the period. The CBS policy ensures that each server's utilization (time bandwidth) $U_i = Qc_i/P_i$ remains constant over time. The function $G_k(t)$ ($\mathbb{R}_0^+ \rightarrow \{true, false\}$) indicates whether a CBS server on CPU_k is eligible for execution at time t.

PMC-regulation. The system features a PMC-based regulator to monitor and limit the memory bandwidth that a CPU_k can consume. PMC-regulation assigns each CPU_k a memory bandwidth B_k, which is enforced by allowing at most Qm_k read transactions within a period M. The memory budget depletes while CPU_k performs memory transactions. The function $A_k(t)$ ($\mathbb{R}_0^+ \rightarrow \mathbb{N}_0$) associates a CPU_k with its instantaneous memory budget at time t. In this paper, we assume that all CPUs have the same replenishment period M. In other words, the CPUs' replenishment periods are assumed (1) to have the same duration and (2) to be synchronously aligned (as presented in the original Memguard article by Yun et al (2013)). PMC regulators divide the system life-cycle in two categories: *regulated* and *stalled*. When regulated, a CPU_k runs and consumes its memory budget ($A_k(t) > 0$). When $A_k(t) = 0$, CPU_k is stalled. Regardless of the CPUs' phase, at the start of each regulation period M all memory budgets are restored ($\forall p \in \mathbb{N}_0 : A_k(pM) = Qm_k$) and stalled CPUs become again regulated.

Programmable Interconnect. The CPUs are connected to main memory via a *run-time configurable* interconnect. The interconnect can discriminate and arbitrate CPU's memory transactions using a policy π, which can be either *Fair* or *Fixed Priority* (FP). The $Fair$ policy aims to balance each CPU's bandwidth, while the FP policy assigns each CPU_k a unique bus priority R_k ($k \in \{0, \ldots, m\}$) and schedules memory transactions accordingly. The function $MaxR(t)$ ($MaxR(t) : \mathbb{R}_0^+ \rightarrow \{0, \ldots, m\}$) indicates the CPU with the highest bus priority at instant t, while $MinR(t)$ ($MinR(t) : \mathbb{R}_0^+ \rightarrow \{0, \ldots, m\}$) indicates the CPU with the lowest bus priority at instant t.

Memory overload. A *memory overload* identifies those situations where, due to the depletion of the CPU memory budget, critical tasks are stalled despite being still eligible for execution. We note that, although a CBS server is used in

this paper, the definition of a *memory overload* does not depend on a specific choice of server regulation. Specifically:

Definition 1 For a critical τ_i ($l_i > 0$), a *memory overload* occurs at $t^{overload}$ if $A_k(t^{overload}) = 0$ and $G_k(t^{overload}) = true$.

MCTI protocol. To enforce the regulation of the system, MCTI's protocol relies on the following rules:

1. The system's life cycle is divided into a succession of memory regulation periods M.
2. At the start of each M:
   - each CPU_k in the system has its memory budget replenished ($\forall k \in \{0, \ldots, m\}, \forall p \in \mathbb{N}_0 : A_k(pM) = Qm_k$) and
   - the interconnect policy is set to $Fair$ ($\pi = Fair$).
3. While $G_k(t) \ \wedge \ A_k(t) > 0$, CPU_k runs regulated and consumes its memory budget.
4. If $G_k(t) \ \wedge \ A_k(t) = 0 \ \wedge \ L_k(t) = 0$, CPU_k is stalled until the start of the next memory regulation period.
5. (Memory Overload) If $G_k(t^{overload}) \wedge A_k(t^{overload}) = 0 \wedge L_k(t^{overload}) > 0$, the interconnect policy is set to fixed-priority ($\pi = FP$), R_k is set according to the following property (Prop. 1), and CPU_k can continue to execute the task scheduled at time t with CBS regulation.
6. If $\neg G_k(t)$, CPU_k is idle.

Property 1 (**Bus priority assignment**). *When any CPU_k is in a memory overload, bus priorities are assigned according to the criticality and deadline of the critical tasks.*

$$\forall t \in \{\mathbb{R}_0^+ \mid G_k(t) \wedge A_k(t) = 0 \wedge L_k(t) > 0\}, \forall k \in \{0, \ldots, m\}, \forall z \in \{0, ..., m \mid (L_z(t) > L_k(t)) \vee ((L_z(t) = L_k(t)) \wedge (D_z(t) < D_k(t)))\} : R_z > R_k \tag{1}$$

Note that rules 1 to 4 describe the budget accounting of a typical PMC regulator such as Memguard. It is the introduction of rule 5 that enables the handling of *memory overload* situations. Whenever $A_k(t) > 0$ (*e.g.*, every M) or $L_k(t) = 0$, CPU_k is not in a *memory overload* situation anymore, and it falls back to the usual PMC regulation mechanism (rules 1 to 4).

In the example illustrated by Fig. 1c, rules 1 to 4 are being used to regulate the system's memory bandwidth from t_0'' to t_2''. Because of τ_0's increased memory consumption, a memory overload occurs at t_2''. Henceforth, rule 5 is applied until the start of the subsequent memory regulation period at t_6'' (rule 2).

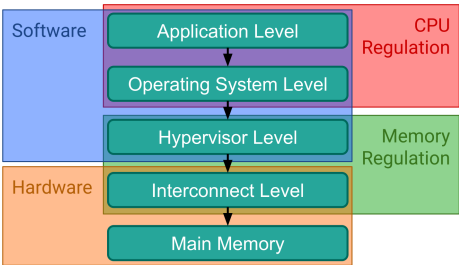


Fig. 2: Layered architecture of MCTI.

5 Architecture

As depicted in Fig. 2, MCTI adopts a layered architecture with five layers ranging from application software level to hardware control of the main memory. The CPU regulation is completely implemented in software at the OS level, while memory regulation implementation is distributed across the hypervisor level and the hardware-based control of the data link to the main memory. Furthermore, lightweight communication between layers is required to propagate, for example, information on the criticality of the currently executing tasks.

5.1 CPU Regulation

Real-time tasks execute at the application level on top of an OS with real-time capabilities. The OS supports a server-based scheduling policy (*e.g.*, Buttazzo (2011)) that provides isolation among the tasks. We use Linux as OS to prototype our architecture as it has been successfully employed in many soft real-time contexts (*e.g.*, Cinque et al (2022)) and constitutes a solid prototyping platform due to its widespread adoption. In Linux, the `SCHED_DEADLINE` scheduling policy realizes a CBS regulation that fulfills the requirements of our architecture. We associate each task τ_i to a server S_i and define its maximum utilization U_i. Each S_i is statically assigned to a CPU_k.

5.2 Memory Regulation

Memory regulation is the most complex part of our architecture and consists of two layers, one implemented at the hypervisor level and one implemented at the hardware level (Fig. 2).

5.2.1 PMC-regulation and Memory Overload Detection

The hypervisor implements a PMC-regulation mechanism that limits the maximum number of memory transactions that the CPUs can issue to the main memory. The choice of a hypervisor to realize PMC-regulation is natural given the widespread adoption of hypervisors in safety-critical contexts to isolate independent workloads with different criticalities. Implementing PMC-regulation at the hypervisor level makes the PMC-regulation transparent to the

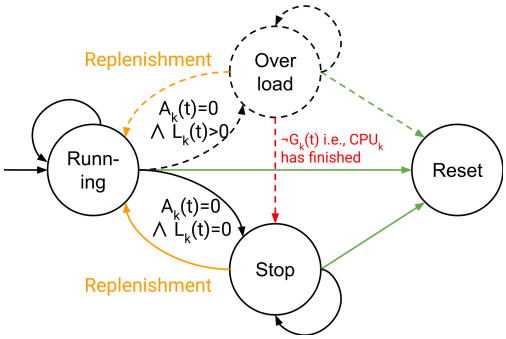


Fig. 3: Overload-aware Memguard Finite State Machine of CPU_k running τ_i. Additions to the standard Memguard FSM are drawn with dashed contours.

OS level, and it allows using different OSs while ensuring memory bandwidth control. Furthermore, even if this work considers only CBS, our architecture would allow different OSs to use different types of CPU server regulation. Hence, separating the PMC-regulation level from the CPU regulation level is a clean architectural choice.

We consider *Memguard* by Yun et al (2016) as PMC-regulation to enforce a target maximum bandwidth B_k. The latter controls the amount of transactions (up to a maximum Qm_k) emitted by a CPU_k within a time frame M. The bandwidth B_k is enforced by stalling CPU_k until the next M whenever Qm_k is depleted. Figure 3 illustrates the default state (*Running*) of the Memguard state machine and the transition to *Stop* when the memory budget Qm_k is depleted.

MCTI's rules (see Sec. 4) are accommodated into Memguard's finite state machine by adding a new state (*Overload*) that captures *memory-overloads* situations. CPU_k enters the *Overload* state if its budget is depleted ($Qm_k = 0$) and its currently running task is critical ($l_i > 0$). Otherwise, it enters the *Stop* state. When one of the CPU enters the *Overload* state, the shared interconnect policy is switched to fixed-priority ($\pi = FP$). The bus priority of each CPU is determined based on the criticality of its running task: the higher the $L_k(t)$, the higher the R_k. If multiple CPUs run a task with the same criticality level, higher R_k is given to the CPU_k whose critical task has a closer deadline (Prop. 1). This strategy facilitates the completion of the most urgent and critical tasks, potentially penalizing other critical tasks running in parallel. We note that without intervention, critical tasks will miss their deadlines when a memory overload occurs. When the (synchronous) replenishment period (M) is reached, budget Qm_k is replenished, and CPUs return to *Running* state. If the *Running* state is re-entered from the *Overload* state upon replenishment, π is switched back to $Fair$. Note that switching the policy to fixed priority does not cause other CPUs to transition to an *Overload* state, meaning that Memguard rules still apply for such CPUs.

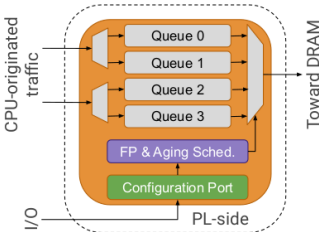


Fig. 4: Abstract overview of the SchIM design

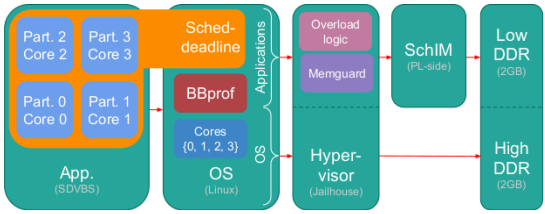


Fig. 5: MCTI with *memory overload*

The *Reset* state in Fig. 3 does not belong to the regulation and is entered asynchronously when the system is subject to a reboot to restore standard unregulated parameters.

5.2.2 Dynamic FP/Fair Interconnect Policy

The lowest part of the memory regulation realized by MCTI is implemented in hardware leveraging the architecture of SchIM Hoornaert et al (2021).

As in the original article, the SchIM module is implemented on the PL side and acts as an intermediate step on the data path between CPUs and DRAM. As shown in Figure 4, each CPU is associated with a queue storing the memory transactions directed to DRAM. Under heavy traffic, the queues are being progressively filled, creating contention within the module and allowing SchIM to schedule the transactions as desired by the system. Scheduling is enacted by deciding which queue's content is forwarded to the target memory and is orchestrated by the hardware transaction schedulers (depicted as *FP & Aging Sched.* and *multiplexer* modules in Fig. 4). The scheduler module defines a set of hardware schedulers (*e.g.*, Fixed-Priority, TDMA) implemented at design time and statically available on the PL at system boot.

This work extends the original SchIM by enabling the dynamic choice of a specific scheduler at run-time and by adding the *Fair* scheduling policy. Specifically, a scheduler can be selected by operating on a set of registers accessible by the whole system through a memory-mapped configuration port (*Configuration Port* in Fig. 4). In addition to this configuration link, our SchIM implementation features two input links for CPU-originated transactions (each one being shared by two CPUs) and one output link to the DRAM.

It should be noted that to-date SchIM-like approaches are the only viable way to enable fine granular scheduling of memory transactions on a COTS platform. In fact, it's unclear whether even advanced—and not yet fully available—QoS solutions such as MPAM ARM (2022) will be able to provide the same granularity and configurability levels.

5.3 Open Challenges

Schedulability analysis of the presented architecture with respect to its system model (Sec. 4) poses significant challenges. Given the desired (by design) independence of OS, hypervisor, and interconnect layers, an overhead-aware schedulability analysis that considers the combined effects of all three layers is challenging. In particular, to the best of our knowledge, the following three main sources of overhead cannot be easily factored in existing overhead-aware schedulability analysis.

Hypervisor-based PMC-regulation overheads. At the OS level, standard techniques (Brandenburg (2011); Buttazzo and Bini (2006)) can be adopted to account for OS, caches, and interrupt overheads in CBS-schedulability analysis. Unfortunately, these techniques cannot be easily extended to consider the impact of PMC-regulation overheads generated at the hypervisor level. Hypervisor-based PMC-regulation is *by design*, transparent to the OS layers. An analysis of PMC-regulation overheads must therefore be conducted at both hypervisor *and* OS level and must consider the combined impact of both CPU-based and task-based overheads. We are unaware of an overhead-based schedulability analysis that could be directly applied to our MCTI architecture.

Interconnect-based overheads. When a critical task enters a memory overload, MCTI updates the priority of the interconnect to privileged memory transactions issued by the "most critical" CPU (Prop. 1). While the interconnect operates with $\pi = FP$, the tasks executing on the CPUs inevitably experience slowdowns that depend on the assigned interconnect priorities and their memory consumption. Integrating such overheads is challenging even for a non-tight analysis where all $\{CPU_k \ \forall k \in \{0, \ldots, m\}\} \setminus \{CPU_{MaxR(t)}\}$ are assumed to be as penalized as $CPU_{MinR(t)}$.

Techniques such as Yun et al (2015) do not consider priority-aware interconnects but could nonetheless be used as a starting point for the analysis of MCTI. Similarly to hypervisor-level PMC-regulation overheads, integrating interconnect-based overheads into a schedulability analysis will be part of our future work.

Criticality-inversion. In addition to interconnect- and hypervisor-PMC-overheads, our MCTI architecture includes another source of pessimism rooted in the lack of fast communication between OS and hypervisor levels. In fact, MCTI does not have an expensive OS-to-hypervisor communication channel (*e.g.*, hypercalls, Siemens AG (2023); Martins et al (2020)) to signal the completion of critical tasks that entered a memory overload. This choice helps reduce the high cost of hypercalls and improves the (common) case where memory overloads occur close to a memory replenishment period.

Note that this source of pessimism is an artifact specific to MCTI's architecture. The implementation-agnostic rules listed in Section 4 do not lead to this condition.

Nonetheless, when considering worst-case schedulability analysis, the effect of criticality inversion at the interconnect must be accounted for the complete

duration of a replenishment period M, and their impact cannot be tightly limited to the duration of a memory overload. We refer to this indirectly-induced overhead as *criticality-inversion*, since, after a memory overload occurs, the actual interconnect priorities and policy can only be restored at the next replenishment period.

6 Implementation

Given the architecture requirements (see Sec. 5), the target platform of this work is a System-on-Chip featuring a tightly integrated FPGA. The selected platform instance is Xilinx's UltraScale+ ZCU102 Xilinx (2022) that features four ARM Cortex-A53 CPUs with a shared 1 MB last-level cache, 4 GB DRAM, and a tightly coupled FPGA. The operating system that realizes the CPU regulation mechanism is a Linux system with a modified kernel.[5] We extend the Jailhouse hypervisor[6] to integrate Memguard with our *memory overload* logic and to interact with our dynamic hardware memory scheduler. The overview of the detailed architecture of MCTI is presented in Fig. 5. This section presents 1) the software and hardware modifications required to enforce the regulation on the system, 2) the memory organization and layout, and 3) the benchmark framework used for the evaluations.

6.1 CPU and Memory Regulation

Because the regulation is enforced at three distinct levels, appropriate communication mechanisms have been defined to exchange the states controlling the regulation. Specifically, the Memguard logic (at the hypervisor level) plays a central role: it monitors the memory budgets of the CPUs, detects possible memory overloads, reads from the OS the criticality and deadline of the currently running task, and drives the bus policy via SchIM when required.

6.1.1 `SCHED_DEADLINE`

As mentioned in Sec. 5, we associate each task τ_i to a CBS server and define its maximum utilization via the `runtime` (Qc_i) and `period` (P_i) parameters. The implementation of CBS in Linux makes tasks not eligible for execution as soon as their budget has depleted, even when the CPU would otherwise be idle. This behavior has practical implications since assigning a large server period P_i would cause τ_i to be suspended for a long time. We selected $P_i = 1\ ms$, which provides a good compromise between the granularity of the regulation and blocking time (the workload under analysis—Sec. 7—has runtime in the range of seconds) and matches the period value of Memguard (see Section 6.1.2). The implementation of CBS being flexible w.r.t. the period, we set this value as it is reasonable for both the CBS with the PMC regulation (see Section 6.1.2). We extended the structure `sched_attr` to accommodate the criticality l_i of

[5] Version 5.4, from https://github.com/Xilinx/linux-xlnx.git
[6] Omitted for review

the task (implicitly 0, *i.e.*, non-critical) and disabled the "`rt_throttling`" to statically pin tasks to CPUs.

The communication with the hypervisor level is realized via a cached, per-CPU shared memory page written by `SCHED_DEADLINE` and read by the hypervisor. When a `SCHED_DEADLINE` task is selected (de-selected) for scheduling, its criticality and current deadline are stored (cleared) in the page.

6.1.2 Overload-aware PMC Regulation

The PMC Regulation (Memguard) implementation in Jailhouse has been extended with the memory overload logic presented in Sec. 5.

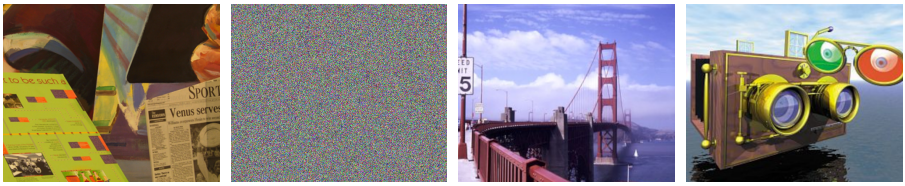
Specifically, when the memory budget of a CPU is depleted and the CPU should be stalled, the overload-aware logic reads the criticality and deadline of the current task on the CPU as propagated by Linux. If the task is critical, the *Overload* state (see Fig.3) is entered, and a change in the bus policy is communicated to SchIM. The priorities on each CPU are determined by checking (for all CPUs) the criticality of each running task and breaking same-criticality chains using the deadlines of the tasks. Upon reception of the synchronous replenishment PMC-regulation interrupt, the memory budget of each CPU is restored, and the bus policy is switched back to $Fair$. The implementation overhead *w.r.t.* the standard Memguard implementation is minimal, and it only consists of reading m criticality and deadline values. In particular, by using a synchronous replenishment period, no additional interrupts or hypercalls are required. Previous studies have shown that selecting very short replenishment periods might cause excessive overheads (*e.g.*, Schwaericke et al (2021); Zuepke et al (2023)). Other studies by Saeed et al (2022) and Yun et al (2013) have used a regulation period of 1 ms. Hence, we used a regulation period $M = P_i = 1\ ms$.

6.1.3 SchIM

The SchIM design from Hoornaert et al (2021) has been extended to add the features discussed in Sec. 5 and to achieve better raw performance. Specifically: (1) the frequency has been set to 300 MHz; (2) the amount of pipeline stages in the architecture has been reduced; and (3) the supported memory scheduling policies have been extended with the *Aging* policy that realizes our $Fair$ default scheduling policy.

The *Aging* policy schedules transactions in a *fair* way by giving priority to the longest stalled transaction while under contention. The scheduler keeps track of the *age* of the queues' head and considers the oldest head for scheduling. The *age* of a queue's head is maintained by a counter increasing for each clock cycle where a transaction stored in the queue's head is stalled (*i.e.*, larger counter values mean older transactions). The counter is reset to zero when the queue's head transaction is scheduled or if the queue is empty.

Switching the bus policy for already set priorities is a fast activity that requires around 40 clock cycles. The bus policy switch is triggered by writing



(a) Input `def1` (b) Input `deg1` (c) Input `nor1` (d) Input `nor2`

Fig. 6: Examples of input used for the SD-VBS suite

the desired scheduling policy and priorities (if required) on the mapped registers exposed by SchIM. As in Hoornaert et al (2021), only one bus policy can be set at a time. In particular, a combination of *Fair* and *FP* for different groups of CPUs is not supported.

6.2 Memory Organization and Layout

MCTI targets systems that isolate memory regions accessed by tasks of different criticalities and avoid sharing memory between such tasks. Ensuring such desirable isolation properties throughout the MCTI stack is not trivial as it involves (1) per CPU memory range allocation, (2) address coloring, and (3) DRAM partitioning. These properties are enforced and required by different layers of the stack. For instance, within SchIM, transactions belonging to a specific CPU are logically identified using the physical address of the memory region that they target. Therefore, a precise mappings of critical tasks to specific memory address ranges has to be enforced to ensure that they will target the appropriate memory regions. Appendix B describes the implementation details of the mechanisms and provides an exhaustive technical description of the several mappings and address translations.

6.3 Benchmarks

The natural targets for the proposed framework are memory-intensive tasks. Hence, in all the experiments displayed in Section 7, memory-intensive benchmarks from the San-Diego Vision Benchmark Suite (SD-VBS) Venkata et al (2009) are used. Specifically, the RT-Bench (Nicolella et al (2022)) adapted version of SD-VBS has been used to simplify the acquisition of performance metrics.

As previously discussed, MCTI is helpful in scenarios where selecting specific memory-regulation levels can lead to real-time constraint violations or to excessive under-utilization of the system. In order to generate such scenarios, we consider multiple different inputs (`def1`, `deg1`, `deg2`, `nor1`, and `nor2`) for the algorithms of the SD-VBS. Fig. 6 showcases a subset of the inputs considered. The key intuition is that when provided with different –but *equally sized*– inputs, vision algorithms behave differently and can generate different

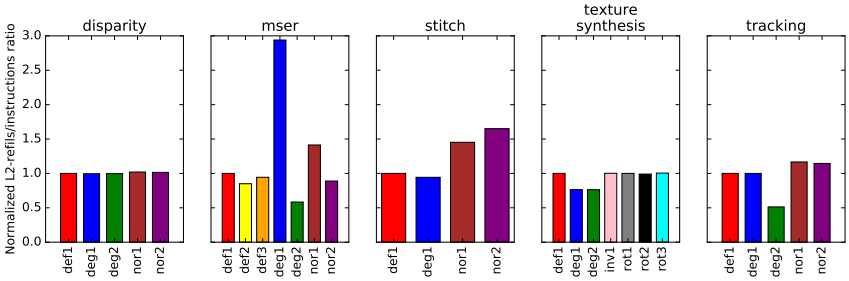


Fig. 7: Measured Bandwidth (L2-refills over instructions retired) for each benchmark using various inputs. Each bar is normalized over the `def1` input.

amounts of memory transactions. All experiments presented in Section 7 have been carried out using this framework.

7 Evaluation

The evaluation of the MCTI architecture presented in Sections 5 and 6 is divided into three phases. In the first phase (Sec. 7.1), we use the PMCs to produce performance profiles of SD-VBS benchmarks under various inputs and highlight their behavioral variations. In the second phase (Sec. 7.2), using the insights gathered in the first phase, we identify benefits, limitations, and trade-offs of the architecture in simplified scenarios where tasks contend for a shared memory budget. Finally, in the third phase (Sec. 7.2.2), the proposed architecture is further tested in high contention scenarios.

7.1 Benchmark Profiling

When dealing with systems using PMC-based memory regulation, system designers must understand the exact memory requirements of each task in order to assign an adequate memory budget. In this section, to facilitate the analysis and profiling of each benchmark "in isolation", we do not enforce any PMC-regulation for the task under analysis (*TUA*). Moreover, we do not re-route the memory accesses through SchIM. Nonetheless, each *TUA* runs in isolation on a dedicated CPU and targets its pre-defined memory partition.

7.1.1 Behavioral variations

As previously discussed, we argue that the execution and the main-memory bandwidth requirements of a benchmark vary depending on the input density. To support this intuition, we run a set of experiments that measure the amount of L2 refills and instructions retired for several benchmark-input pairs. These PMC values respectively relate to the memory bandwidth and to the execution time of a task.

Fig. 7 displays the ratio of variations in the amount of L2 refills, and instructions retired that a given benchmark experiences. The results show the

normalized ratio for different benchmarks *w.r.t.* the `def1` input (leftmost, red-bar). In the plots, each inset presents a benchmark, and the available inputs are indicated on the x-axis.

Observation 1. *For the selected set of benchmarks, variations in the input-density have a direct and difficult-to-predict impact on the memory activity and CPU activity.*

In `mser`, both the instructions retired and the L2 refills vary considerably for different inputs. This is especially the case for the `deg1` input, where the amount of L2 refills triples while the instructions retired increases by only a marginal extent, resulting in a significant increase in the ratio. Conversely, `tracking` experiences small L2 refill variations for different inputs but a high variation of instructions executed, leading to lower ratios as represented by `deg2`. Finally, benchmarks such as `disparity` and `texture_synthesis` show little-to-no variation.

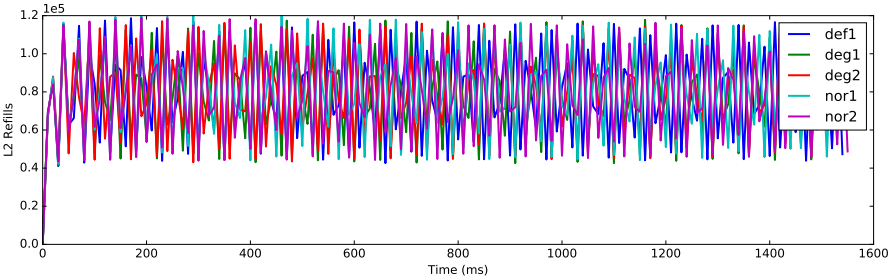
7.1.2 Run-time memory requirements

Unless a task is known to have a constant memory utilization, calculating its memory budget based on *e.g.*, the total amount of L2 refills is bound to incur over- or under-estimations. Thus, a careful investigation of the memory accesses at run time is required to gain a better understanding. In the experiments reported in this section, we measure the number of L2 refills within a period of 10 *ms* throughout the execution of the *TUA*.

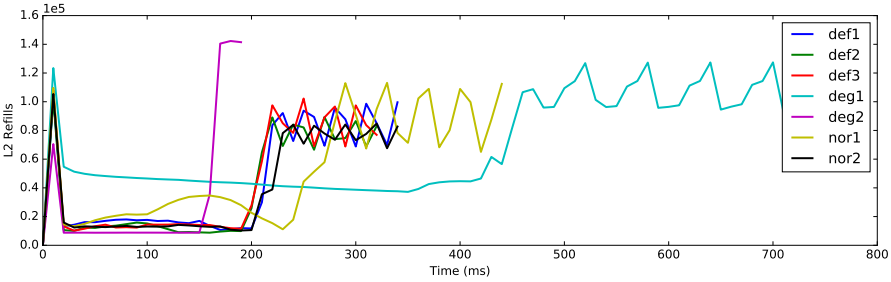
The outcome of this set of experiments for `disparity`, `mser`, and `tracking` are displayed in Fig. 8a, Fig. 8b, and Fig. 8c, respectively. For each of these figures, we report on the y-axis the amount of L2 refills measured every 10 *ms* during the execution of the *TUA* (x-axis). The process is repeated for all inputs available for the benchmark under test (`mser` has two additional inputs: `def2` and `def3`).

Observation 2. *The benchmarks do not display a linear temporal memory access pattern (or constant memory consumption), making the problem of assigning a tight and sufficient memory bandwidth difficult.*

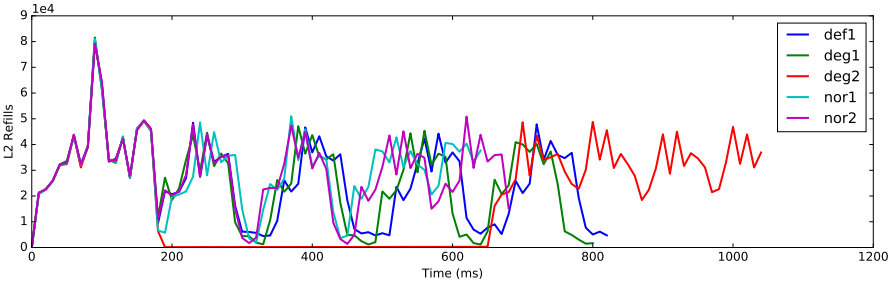
As already suggested by the experiments discussed in Sec. 7.1.1, the three benchmarks display considerably different memory access patterns. On the one hand, `disparity` has a relatively constant memory consumption despite frequent oscillations. On the other hand, `tracking` and `mser` display higher variations. This is especially the case for `mser`, which features three distinct phases. A short but intense memory phase from 0 *ms* to 20 *ms*, followed by a quieter phase until 200 *ms*, and finally, a new memory-intensive phase until task completion. In addition, under certain inputs such as `deg1`, `deg2`, and `nor1`, the duration and intensity of the phases drastically change. Likewise,



(a) `disparity`



(b) `mser`



(c) `tracking`

Fig. 8: Progression of the memory consumption of `disparity`, `mser`, and `tracking` for various inputs.

`tracking` behaves differently depending on the input, with shifted phases and considerably different memory consumption (*e.g.*, `deg2`).

Observation 3. *The hard-to-predict impact of input density on benchmarks complicates the assignment of bandwidth, leading to over- or under-provisioning.*

Fig. 8b showcases the challenges of setting a (single) static memory budget. For example, a conservative budget of 100,000 transactions per 10 *ms* would

Table 2: Summary of the scenarios considered for the evaluation.

Abbrev.	Scenario	TUA	TUA criticality	Other cores
ScSt	Single-core Single-task	Runs alone	Not critical	
ScStC	Single-core Single-task Critical	Runs alone	Critical	
ScMt	Single-core Multi-task	W/ co-runner	Not critical	
ScMtC	Single-core Multi-task Critical	W/ co-runner	Critical	
McSt	Multi-core Single-task	Runs alone	Not critical	Run non-critical memory bombs
McStC	Multi-core Single-task Critical	Runs alone	Critical	Run non-critical memory bombs
McMt	Multi-core Multi-task	W/ co-runner	Not critical	Run non-critical memory bombs
McMtC	Multi-core Multi-task Critical	W/ co-runner	Critical	Run non-critical memory bombs

not prevent regulation during intense memory phases while still causing over-provisioning for more than half of the execution time. The situation is even worse if we consider the special case of `deg2`, as defining a proper regulation for this input would lead to over-provisioning in most cases. Such challenges present system designers with a hard choice between over-provisioning at the expense of reduced bandwidth for the other CPUs or risking delays and possibly deadline misses in the case of memory overloads.

7.2 MCTI Assessment

For the evaluation of MCTI, we use the prototype implementation described in Sections 5 and 6. Specifically, memory transactions from the *TUA* are re-routed through the PL side, and we enforce memory regulation via our modified Memguard and CPU domain isolation via CBS. As previously mentioned, both the Memguard regulation period and the CBS period are set to 1 ms. In addition to the benchmarks (*TUA*), we also consider a *co-runner* stress task, which generates pressure on the memory sub-system by purposely creating LLC cache-line misses. The *TUA* and its co-runner run on the same CPU, thus sharing a common memory budget but targeting different (isolated) cache partitions.

In this Section, experiments will either focus on *Single-* or *Multi-core* scenarios listed in Table 2. In total, we consider up to eight different scenarios whose traits are displayed in Table 3.

The *SVM* benchmark is particularly time-expensive and has been excluded from the evaluation presented in this section. Furthermore, other benchmarks (`sift` and `multi_ncut`) either cause runtime errors or do not apply to different inputs and have similarly been excluded from the results presented in this section. We used the data from the experiments presented in Sec. 7.1.2 to guide the selection of Memguard budgets and evaluate different CBS budgets (as % of available CPU) to assign to the *TUA*.

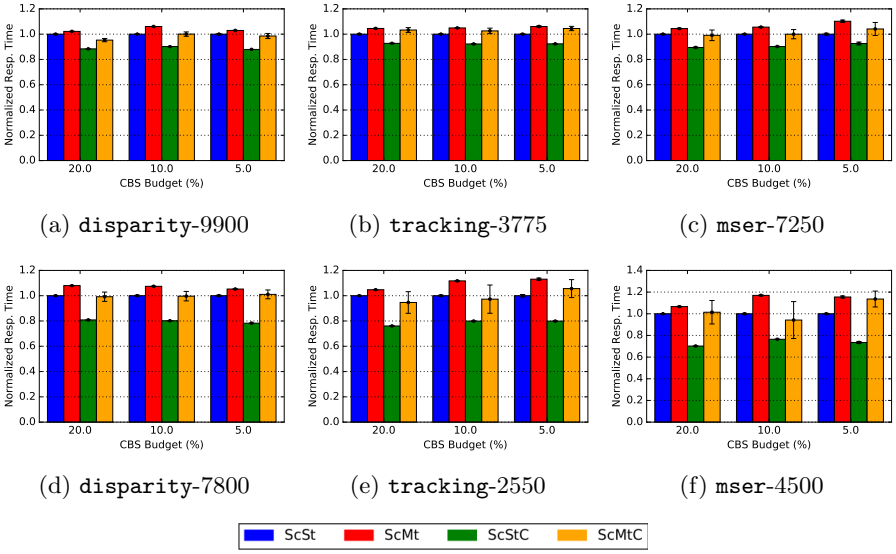
In this section, we present the most interesting trends of the evaluation. Thus, for `disparity`, `mser`, and `tracking` benchmarks, we focus on two Memguard budgets (Table 4) that intercept: 1) the average memory consumption of the benchmark (*Intermediate*) and 2) an average with a safety-margin corresponding to $(max - average)/2$ extra memory transactions (*Full*). We will

Table 3: Description of the experimental setups used in Section 7.2.

		CPU_0		CPU_1	CPU_2	CPU_3
ScSt	Task	τ_0	×	×	×	×
	Criticality	$l_0 = 0$	×	×	×	×
	CBS Per.	1 ms	×	×	×	×
	CBS Util.	U_0	×	×	×	×
ScStC	Task	τ_0	×	×	×	×
	Criticality	$l_0 = 1$	×	×	×	×
	CBS Per.	1 ms	×	×	×	×
	CBS Util.	U_0	×	×	×	×
ScMt	Task	τ_0	τ_1	×	×	×
	Criticality	$l_0 = 0$	$l_1 = 0$	×	×	×
	CBS Per.	1 ms	1 ms	×	×	×
	CBS Util.	U_0	$1 - U_0$	×	×	×
ScMtC	Task	τ_0	τ_1	×	×	×
	Criticality	$l_0 = 1$	$l_1 = 0$	×	×	×
	CBS Per.	1 ms	1 ms	×	×	×
	CBS Util.	U_0	$1 - U_0$	×	×	×
McSt	Task	τ_0	×	τ_2	τ_3	τ_4
	Criticality	$l_0 = 0$	×	$l_2 = 0$	$l_3 = 0$	$l_3 = 0$
	CBS Per.	1 ms	×	1 ms	1 ms	1 ms
	CBS Util.	U_0	×	U_2	U_3	U_4
McStC	Task	τ_0	×	τ_2	τ_3	τ_4
	Criticality	$l_0 = 1$	×	$l_2 = 0$	$l_3 = 0$	$l_3 = 0$
	CBS Per.	1 ms	×	1 ms	1 ms	1 ms
	CBS Util.	U_0	×	U_2	U_3	U_4
McMt	Task	τ_0	τ_1	τ_2	τ_3	τ_4
	Criticality	$l_0 = 0$	$l_1 = 0$	$l_2 = 0$	$l_3 = 0$	$l_3 = 0$
	CBS Per.	1 ms	1 ms	1 ms	1 ms	1 ms
	CBS Util.	U_0	$1 - U_0$	U_2	U_3	U_4
McMtC	Task	τ_0	τ_1	τ_2	τ_3	τ_4
	Criticality	$l_0 = 1$	$l_1 = 0$	$l_2 = 0$	$l_3 = 0$	$l_3 = 0$
	CBS Per.	1 ms	1 ms	1 ms	1 ms	1 ms
	CBS Util.	U_0	$1 - U_0$	U_2	U_3	U_4
PMC	PMC Per.	1 ms	1 ms	1 ms	1 ms	1 ms
	PMC Bud.	See Table 4			1.500.000	

Table 4: Summary of the benchmarks' bandwidths.

Benchmark	`disparity`	`mser`	`tracking`
Full (Qm_k)	9900	7250	3775
Intermediate	7800	4500	2250



(a) `disparity`-9900 (b) `tracking`-3775 (c) `mser`-7250

(d) `disparity`-7800 (e) `tracking`-2550 (f) `mser`-4500

Fig. 9: Response time distribution for multiple scenarios with `def1` input.

refer to the association of a benchmark and a PMC budget as `benchmark`-budget. For instance, `tracking` with an associated memory budget of 2550 transactions is referred to as `tracking`-2550. In the multi-core category, the co-running CPUs have been assigned a large Memguard budget to ensure that they are able to pressure the bus as much as possible without being regulated. For each benchmark, we attribute a CPU budget of either 20%, 10%, or 5% to the *TUA* in *Single-core* scenarios and CPU budget of either 90%, 70%, 50%, 30% 20%, or 10% in *Multi-core*. In all Multi-task scenarios (i.e., *ScMt*, *ScMtC*, *McMt*, and *McMtC*), the co-runner is assigned the remaining CPU budget (*i.e.*, the total budget of the two equals 100%) as shown in Table 3.

We focus the discussion on the most representative subsets of the benchmark-input-budget configurations. Such configurations have been selected among the full list of performed experiments (see Appendix A).

7.2.1 Impact on response time in single-core scenario

In this section, we study the impact of the proposed memory overload handling mechanism on the response time of the *TUA*. The set of experiments presented uses the setup and rules used in the previous section. For the sake of clarity, the benchmarks considered here only focus on the `def1` input.

The measured response times of the *TUA* under the experimented configurations and scenarios are displayed in Fig. 9. The figure is divided into six sub-figures, each focusing on a benchmark-PMC-budget combination. The average response times measured for the selected scenarios are grouped in bar clusters. Each figure has three of them; one for each *TUA*'s CBS utilization

considered (x-axis). Within each cluster, the reported average response times are normalized over the *ScSt* scenario to ease comparison.

Observation 4. *For all tested configurations and regardless of the benchmark under analysis, the* ScStC *scenario displays the lowest average response time (Fig. 9).*

In the *ScStC* scenario, the *TUA* is the only task running and, hence, it is the only task that can deplete the memory budget. The *TUA* being critical, whenever a memory budget depletion occurs, an overload situation also occurs, and our overload handling mechanism is triggered. Because triggering the overload handling mechanism means that the regulation rules are bypassed, in *ScStC*, the *TUA* always bypasses the regulation, resulting in response times shorter than the regulated baseline.

Observation 5. *Under competition for memory budget, MCTI successfully shields the* TUA *from the co-runners. On average,* ScMtC *response times are equivalent to the* TUA *running in isolation. The extent of improvements varies according to the benchmark (Fig. 9).*

Fig. 9 shows that, in the majority of the *ScMtC* cases, the response times are, on average, equivalent to *ScSt*, our baseline. `tracking`-3775, `tracking`-2550, `mser`-7250, and `mser`-4500 are exceptions to this trend and show marginal increases in average response times. Such trends occur in constrained scenarios where the CBS utilization is set to 5%. Finally, we note that *ScMtC* is the only scenario incurring fluctuations as shown by the standard deviations in Fig. 9. We suspect that the cause of such fluctuations is the lack of high-precision synchronization between the start of memory-regulation and CPU-regulation periods between hypervisor and OS layers. Depending on the workload, the misalignment between memory and CPU regulation can cause fluctuations in the budget-depletion instants for both CPU *or* memory. In turn, this can result in faster (slower) response times depending on the periodicity and memory requirements of the workloads. We discuss possible tradeoffs and solutions in Section 8.

7.2.2 Impact on the response time in multi-core scenario

This section assesses the response time under multi-core workload scenarios. The setup for this evaluation is identical to the one used previously, except that the other CPUs are active. In order to put pressure on the memory subsystem, these co-running CPUs emit large read memory request sequences in the direction of the PL side and, indirectly, towards the main memory. These *memory bombs* are marked as non-critical and hence, should obtain a low priority on the shared interconnect when memory overload occurs.

Observation 6. *In multi-core scenarios and well-provisioned memory budget configurations, MCTI isolates the* TUA *from co-runners' disturbances (Fig. 10).*

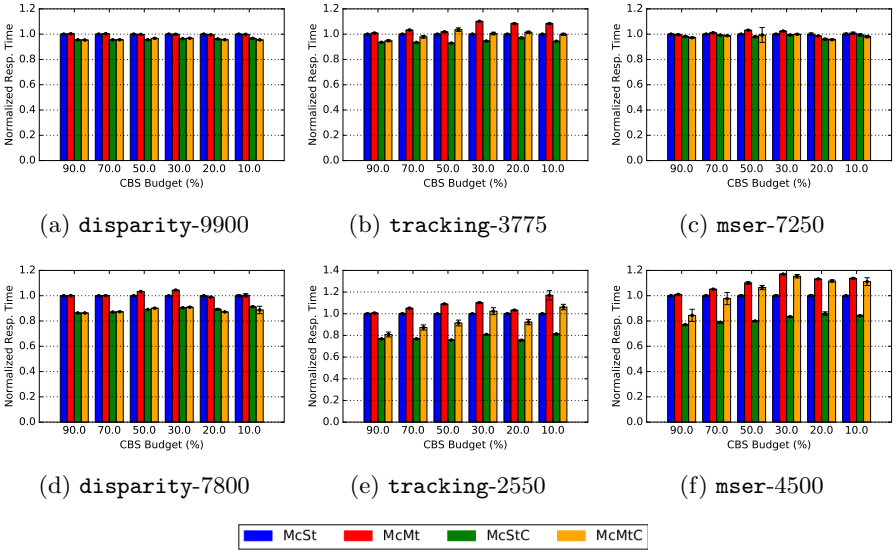
For the *Full* memory configurations (upper row in Fig. 10), we observe that the measured response times for the *McMtC* scenario are equivalent to *McSt*, our baseline. There are only two exceptions: (1) an outlier in `tracking`-3775 with a CBS utilization of 50% and (2) a larger standard deviation in `mser`-4500 with a CBS utilization of 50%. We can even observe that for `disparity`-9900 and high CBS utilizations for `tracking`-3775 and `mser`-4500, the reported average response times are equivalent to *McStC*.

Observation 7. *In multi-core scenarios with constrained memory budget configurations, the average response times measured vary (Fig. 10).*

For constrained memory budget configurations (lower row in Fig. 10), a majority of CBS utilizations yield average response times contained between *McSt* (our target) and *McStC* (the best case scenario). On the other hand, for low CBS utilizations, the average response times recorded for *McMtC* increase. For `tracking`-2550, *McMtC*'s response times remain underneath *McSt* levels (our target) but reach or slightly exceed for 30% and 10% CBS utilizations. Likewise, `mser`-4500 has a *McMtC* response time in par with *McStC* (the best case scenario) for a CBS utilization of 90% before equaling *McSt* (our target) for 70% CBS utilization and, finally, reaching the levels of *McMt* (the worst-case scenario) for the lower CBS utilizations.

Observation 8. *In scenarios with constrained memory budgets, it is difficult to predict the impact of MCTI on the response time, since it depends on the benchmark and the CBS utilization.*

Under multi-core scenarios, as emphasized by Observations 6 and 8, predicting the exact response times of the *TUA* is challenging due to three influencing factors: the memory budget, the CBS utilization, and the benchmark's nature (w.r.t. the memory load). Observation 8 suggests that a system with a constrained memory budget is more sensitive to the settings of other configuration parameters. Due to the unexpected influence of input density on memory requirements, constrained memory budget configurations are not unlikely, mandating a careful profiling and characterization of benchmarks and applications behavior for multiple parameters. Moreover, in conjunction with the constrained memory budget, the configuration of the CBS utilization also affects the response times in *McMtC* scenarios in a non-linear manner. The inflection point where the *McMtC* response time exceeds the target for a given CBS utilization depends on the benchmark being considered. As a result, determining the appropriate CBS utilization requires profiling and analysis of the benchmark's behavior.



(a) `disparity`-9900 (b) `tracking`-3775 (c) `mser`-7250

(d) `disparity`-7800 (e) `tracking`-2550 (f) `mser`-4500

Fig. 10: Normalized response time for multiple scenarios using the `def1` input.

8 Discussion

Memory and runtime behavior may considerably change depending on the type of inputs that applications are processing (Sections 7.1.1, 7.1.2). The mismatches between the desired flexibility in CPU scheduling, isolation of mixed-criticality workloads, and inflexibility in the per-CPU memory management may result in memory overloads that prevent high-criticality tasks from running in their allocated CPU reservations. The architecture of MCTI targets all such dimensions by integrating CBS-reservations atop a partitioning hypervisor and by relaxing the inflexibility of the per-CPU memory management with a more flexible interconnect policy.

MCTI can successfully protect the critical *TUA* from external disturbances coming from the concurrent non-critical co-runners (Sections 7.2.1, 7.2.2). In both single- and multi-core scenarios, the average response time of the critical *TUA* is lower than the worst-case scenario (*i.e.*, *ScMt*) and, on average, equal to the ideal isolated scenario (*i.e.*, *McSt*). Nonetheless, in all scenarios where the *TUA* is critical (*i.e.*, *ScMtC* and *McMtC*), it can be observed that variations in the response time are present; hence, the exact shielding effect of MCTI's regulation on the bandwidth varies as a function of the CBS utilization and the benchmarks themselves. However, such deviations are contained and—as verified from the raw measurements—no *TUA*'s response time exceeds the worst-case scenario (*i.e.*, *ScMt* or *McMt*). This indicates that even in the unlikely worst-case scenario, MCTI never exceeds the worst response time while providing, in most cases, response times similar to the ideal isolated

target. Considering these results, we believe MCTI is an attractive choice for mixed-criticality soft real-time systems.

The reported fluctuations are a direct indication of the difficulty of configuring systems using budget-based regulation. Although MCTI helps isolate critical tasks, it does not exempt from a careful profiling of the tasks considered. Specifically, both CBS utilization, input density, and the memory-access patterns of benchmarks affect the behavior of tasks regulated by MCTI.

One of the most complex configuration aspects of MCTI is the lack of low-overhead synchronization between the hypervisor and the OS layers (*e.g.*, to ensure aligned memory and CBS periods). Synchronization *hypercalls* are expensive, and sharing read and *write* memory between the hypervisor and the OS violates the separation of different privilege levels. On the other hand, directly realizing memory regulation within the OS lacks the isolation properties provided by *e.g.*, partitioning hypervisors.

Even by sacrificing the isolation capabilities of a hypervisor, the interface offered by the hardware to control memory is far from flexible and from providing low overheads. In fact, despite logically belonging to the hardware, PMC regulators must be periodically replenished by the software. In addition, one must consider the large constraints imposed by *e.g.*, SchIM as the conjunction of the FPGA routing and the back-pressure mechanism feedback considerably adds to the complexity of implementation.

9 Related Work

Memory bandwidth partitioning has found wide adoption for the consolidation of real-time applications on multicore platforms. In particular, budget-based bandwidth regulation initially proposed in Yun et al (2016) has received significant attention owing to its practicality. Several works have proposed schedulability results for systems under static Yun et al (2016); Awan et al (2018b); Mancuso et al (2017) and dynamic Awan et al (2018a); Agrawal et al (2017, 2018) bandwidth regulation. In a way that is closely related to this paper, the interplay between CPU-level scheduling and budget-based memory bandwidth regulation has been explored in the case of fixed-priority Yun et al (2016); Mancuso et al (2017); Agrawal et al (2018), mixed-criticality Awan et al (2018a), and multi-frame task models Awan et al (2019). In the derivation of the aforementioned results, the lack of coordination between the CPU scheduler and bandwidth regulators is either prevented—i.e., task context-switches can only occur at the boundary of regulation periods—or accounted for in the analysis. Furthermore, these works consider applications whose worst-case memory bandwidth demand can be either statically derived or experimentally bounded. The proposed MCTI differentiates itself from these works because 1) it considers a realistic implementation of CPU-level scheduling and memory bandwidth regulation enacted at two different layers of the software stack; 2) postulates that applications with input-dependent memory access patterns require a reactive approach for joint CPU and memory management; and 3)

describes a possible hardware/software co-design to add runtime elasticity to bandwidth regulation.

In light of the limitations and additional analysis complexity caused by budget-based bandwidth regulation, a number of researchers have investigated variations and alternative approaches to enact inter-core bandwidth partitioning. First, implementation at the hypervisor level was proposed in Modica et al (2018); Dagieu et al (2016); Martins et al (2020) as a way to significantly lower the regulation overhead and to make it transparent *w.r.t.* CPU scheduling. Similarly, we adapted support for PMC-based regulation implemented in the Jailhouse hypervisor. A second line of work has attacked the problem of implementing bandwidth partitioning directly in hardware. The first work in this direction was by Zhou and Wentzlaff (2016), while a generalization of the regulation strategy that could be applied at multiple levels of the memory hierarchy was studied in Farshchi et al (2020). In the same spirit, other works have investigated the use of bandwidth regulation primitives already available in commercial platforms Sohal et al (2020); Serrano-Cases et al (2021); Houdek et al (2017). Compared to these works, our MCTI is substantially different in scope because its goal is to augment budget-based regulation—regardless of its implementation—with the ability to handle transient overload conditions.

The fundamental problem of unarbitrated memory bandwidth contention has also been attacked by devising adaptations at the level of the main memory interconnect and controller. In particular, the works in Mirosanlou et al (2020); Hassan et al (2017); Valsan and Yun (2015); Jalle et al (2014) focus on modifications to the DRAM controller logic to drastically reduce the worst-case latency of main memory requests in the presence of multicore contention. On a parallel track, enforcing Time Division Multiplexing (TDM) at the level of interconnect has been explored in Hebbache et al (2018); Jun et al (2007); Li et al (2016); Kostrzewa et al (2016). For instance, the work in Kostrzewa et al (2016) proposes a slack-based bus arbitration scheme where the per-transaction slack is static and computed offline for all the critical tasks, while the authors in Hebbache et al (2018) propose a strategy to compute a safe lower-bound on the slack of memory requests at runtime. Although research on predictable memory interconnects and controllers have achieved important milestones, the inability to efficiently carry out system-level implementation and evaluation has traditionally hindered their practicality. The work proposed in Hoornaert et al (2021), which represents one of the building blocks of MCTI, demonstrated that implementing transaction-level memory scheduling is possible in multicore systems with on-chip programmable logic. In the context of the literature surveyed above, MCTI is the first work to propose the integration of task- and transaction-level memory scheduling strategies to handle unpredictable overload conditions while delivering a full-stack implementation on a commercial system.

10 Conclusion

In this paper, we discussed the difficulties of appropriately setting CPU and memory budgets for real-time tasks with memory needs dependent on input density (*e.g.*, vision or AI-based applications). Furthermore, we have shown how—in the worst-case—such misconfigurations might lead to *memory overloads* where critical tasks are not eligible for scheduling due to a premature depletion of the memory budget.

In order to solve these issues while preserving isolation among mixed-criticality taks, we proposed MCTI, a layered architecture integrating OS-based CBS-regulation and hypervisor-based memory management with a flexible management of hardware interconnect priorities. To the best of our knowledge, MCTI is the first architecture that attempts to holistically address the needs of a) CPU- and memory isolation, and b) strong isolation of mixed-criticality workloads, in the face of inflexible management of the interconnect.

The prototype builds on established systems such as the Linux kernel, CBS, and Memguard. We have proposed, described, implemented, and assessed a full-stack architecture capable of handling and taming the effects of memory overloads in most cases. The implementation is evaluated on a widely available out-of-the-shelf platform.

Our results indicate that MCTI is effective in protecting critical tasks from external interference and avoiding memory-overload issues. Nonetheless, the results also indicate that achieving CPU isolation and flexible memory management while preserving strong partitioning of mixed-criticality workloads is a non-trivial task, and several improvements at both software and hardware levels are needed. We intend to progressively devise such improvements in future works.

Appendix A List of experiments

For all experiments, we run the benchmark using multiple benchmark-input configurations. Each table is populated using the following code:

✓ indicates that the benchmark-input has been run and is shown
∼ indicates that the benchmark-input has been run but is not shown
× indicates that the benchmark-input has not been run
– indicates that the benchmark-input pair does not exist.

Benchmark-input pairs run for experiments carried in Section 7.1.1.

Benchmark	def1	def2	def3	deg1	deg2	nor1	nor2	inv1	rot1	rot2	rot3
`disparity`	✓	–	–	✓	✓	✓	✓	–	–	–	–
`mser`	✓	✓	✓	✓	✓	✓	✓	–	–	–	–
`stitch`	✓	–	–	✓	–	✓	✓	–	–	–	–
`texture.`	✓	–	–	✓	✓	–	–	✓	✓	✓	✓
`tracking`	✓	–	–	✓	✓	✓	✓	–	–	–	–
`stitch`	∼	–	–	∼	–	∼	∼	–	–	–	–
`localization`	∼	–	–	–	–	–	–	–	–	–	–

Benchmark-input pairs run for experiments carried in Section 7.1.2.

Benchmark	def1	def2	def3	deg1	deg2	nor1	nor2	inv1	rot1	rot2	rot3
`disparity`	✓	–	–	✓	✓	✓	✓	–	–	–	–
`mser`	✓	✓	✓	✓	✓	✓	✓	–	–	–	–
`stitch`	∼	–	–	∼	–	∼	∼	–	–	–	–
`texture.`	∼	–	–	∼	∼	–	–	∼	∼	∼	∼
`tracking`	✓	–	–	✓	✓	✓	✓	–	–	–	–
`stitch`	∼	–	–	∼	–	∼	∼	–	–	–	–
`localization`	∼	–	–	–	–	–	–	–	–	–	–

Benchmark-input pairs run for experiments carried in Sections 7.2.1 and 7.2.2.

Benchmark	def1	def2	def3	deg1	deg2	nor1	nor2	inv1	rot1	rot2	rot3
`disparity`	✓	–	–	×	×	×	×	–	–	–	–
`mser`	✓	×	×	×	×	×	×	–	–	–	–
`stitch`	×	–	–	×	–	×	×	–	–	–	–
`texture.`	×	–	–	×	×	–	–	×	×	×	×
`tracking`	✓	–	–	×	×	×	×	–	–	–	–
`stitch`	×	–	–	×	–	×	×	–	–	–	–
`localization`	×	–	–	–	–	–	–	–	–	–	–

Appendix B Memory organization and Layout

The MCTI architecture stack relies on multiple levels of address mappings and translations. Each of these levels have their own requirements and the mappings presented this section are directly linked to the modules used.

B.1 VA to IPA Translation

Our MCTI architecture (see Fig. 5) uses two levels of address translations: at the OS level, virtual addresses (VA) are translated to intermediate physical addresses (IPA), and at the hypervisor level, IPAs are translated to physical addresses (PA).

The standard user-level allocators in Linux (*i.e.*, those used to serve *e.g.*, `malloc()` requests) do not allow precise control of which IPAs will be used to back the memory request. To overcome this limitation, we extended an existing profiling tool-chain by Ghaemi et al (2021) to ensure that allocations of memory pages in the OS for selected tasks will always be served from predefined memory regions with well-known contiguous IPAs.

Our modifications allow augmenting the ELF of the binary target programs (corresponding to our compiled target tasks) with additional metadata. We modified the ELF-loader in the Linux kernel to tag the process descriptors accordingly when these metadata are detected at binary load time. In particular, the metadata encodes a CPU ID that helps identify which of the predefined memory regions/partitions should be used to serve the tasks' memory allocation. The CPU ID (from 0 to $m-1$) is read by the loader when the ELF is parsed and added to the `mm_struct` of the corresponding process. Additionally, an identifier (`VM_ALLOC_PVT_CORE`) is added to the Virtual Memory Area (VMA) flags (`vm_flag`) in all the VMA descriptors (`vm_area_struct`) of the process. For a tagged process, we also ensured that any new VMA created at runtime (*e.g.*, via `mmap` or `sbrk`) inherits the `VM_ALLOC_PVT_CORE` identifier.

Upon (intermediate) physical page allocation as a result of demand paging, the Linux kernel uses the Buddy System to allocate a new page from the global free list. We modified the allocation path to redirect allocation towards our per-core predefined memory regions whenever the `VM_ALLOC_PVT_CORE` is detected in the descriptor of the VMA where the allocation is being performed. The predefined memory regions are reserved at boot time using the kernel Device Tree Blob (see the full list in Table B1). The regions are marked as "reserved" and are thus excluded by the default Linux kernel allocator. Instead, our modified allocator remaps the reserved areas and creates new allocator pools, one per CPU and corresponding to the reserved regions, to serve allocation and deallocation requests. The changes in the page allocation path and ELF loader amount to approximately 350 lines of code.

B.2 IPA to PA Translation and PL-Routing

After ensuring at the OS level that memory allocations of target tasks are served from pre-configured memory partitions, the hypervisor (see Fig. 5) is responsible for translating IPAs into appropriate physical addresses. This translation step also takes care of additional requirements of the MCTI architecture.

Under MCTI, accesses to the pre-configured memory partitions must be re-routed through SchIM (on the PL-side), while standard Linux accesses continue to directly target the main memory (without PL-side indirection). Such re-routing is configured during the IPA to PA translation (see Table B1), thus creating two separate routes to the main memory.

Additionally, the IPA to PA translation also prevents last-level cache interference among logically independent partitions. By using cache-coloring (Kloda et al (2019)) , contiguous IPAs are translated into *colored* PAs within the configured partition pools. Note that this step de facto reduces the maximum size of a contiguous IPA range by a factor that depends on the number of independent colors (in our case four, one per CPU partition). Within the SchIM module, the memory transactions targeting the colored physical addresses are converted back to contiguous ranges following a process similar to Roozkhosh and Mancuso (2020).

Table B1: Summary of the different address spaces

	Colored	Cached	IPA	PA	SchIM
Partition 0	Yes (0xF000)	Yes	0x00_1000_0000	0x10_4000_0000	0x00_1000_0000
Partition 1	Yes (0x0F00)	Yes	0x00_2000_0000	0x10_8000_0000	0x00_2000_0000
Partition 2	Yes (0x00F0)	Yes	0x00_3000_0000	0x48_C000_0000	0x00_3000_0000
Partition 3	Yes (0x000F)	Yes	0x00_4000_0000	0x49_0000_0000	0x00_4000_0000
Linux	No	Yes	0x08_0000_0000	0x08_0000_0000	—
SchIM conf.	No	No	0x00_8000_0000	0x00_8000_0000	—
Shared Mem.	No	Yes	0x00_FFFC_0000	0x00_FFFC_0000	—

Finally, since SchIM has full control of each memory transaction, the bank interleaving configuration (*i.e.*, row-bank-column) has been disabled (*i.e.*, changed into bank-row-column), thus enabling a simpler bank partitioning for the regions of interest.

References

Abeni L, Buttazzo G (1998) Integrating multimedia applications in hard real-time systems. In: Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279), pp 4–13, https://doi.org/10.1109/REAL.1998.739726

Agrawal A, Fohler G, Freitag J, et al (2017) Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study. In: Bertogna M (ed) 29th Euromicro Conference on Real-Time Systems (ECRTS 2017), Leibniz International Proceedings in Informatics (LIPIcs), vol 76. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp 2:1–2:22, https://doi.org/10.4230/LIPIcs.ECRTS.2017.2, URL http://drops.dagstuhl.de/opus/volltexte/2017/7174

Agrawal A, Mancuso R, Pellizzoni R, et al (2018) Analysis of dynamic memory bandwidth regulation in multi-core real-time systems. In: 2018 IEEE Real-Time Systems Symposium (RTSS), pp 230–241, https://doi.org/10.1109/RTSS.2018.00040

Anandtech (2019) NVIDIA Drive AGX Orin. https://www.anandtech.com/show/15245/nvidia-details-drive-agx-orin-a-herculean-arm-automotive-soc-for-2022, accessed: 2021-10-13.

ARM (2022) Arm Architecture Reference Manual Supplement. Memory System Resource Partitioning and Monitoring (MPAM) for Armv8-A. https://developer.arm.com/docs/ddi0598/latest Accessed: 2021-02-08

Awan MA, Bletsas K, Souto PF, et al (2018a) Mixed-criticality scheduling with dynamic memory bandwidth regulation. In: 2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and

Applications (RTCSA), pp 111–117, https://doi.org/10.1109/RTCSA.2018.00022

Awan MA, Souto PF, Bletsas K, et al (2018b) Worst-case Stall Analysis for Multicore Architectures with Two Memory Controllers. In: Altmeyer S (ed) 30th Euromicro Conference on Real-Time Systems (ECRTS 2018), Leibniz International Proceedings in Informatics (LIPIcs), vol 106. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp 2:1–2:22, https://doi.org/10.4230/LIPIcs.ECRTS.2018.2, URL http://drops.dagstuhl.de/opus/volltexte/2018/9002

Awan MA, Souto PF, Bletsas K, et al (2019) Memory bandwidth regulation for multiframe task sets. In: 2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp 1–11, https://doi.org/10.1109/RTCSA.2019.8864563

Brandenburg BB (2011) Scheduling and locking in multiprocessor real-time operating systems. PhD thesis, The University of North Carolina at Chapel Hill

Buttazzo G, Bini E (2006) Optimal dimensioning of a constant bandwidth server. In: 2006 27th IEEE International Real-Time Systems Symposium (RTSS'06), pp 169–177, https://doi.org/10.1109/RTSS.2006.31

Buttazzo GC (2011) Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series). Springer-Verlag

Cinque M, De Tommasi G, Dubbioso S, et al (2022) Rpuguard: Real-time processing unit virtualization for mixed-criticality applications. In: 2022 18th European Dependable Computing Conference (EDCC), pp 97–104, https://doi.org/10.1109/EDCC57035.2022.00025

Dagieu N, Spyridakis A, Raho D (2016) Memguard: A memory bandwith management in mixed criticality virtualized systems memguard kvm scheduling. In: 10th Int. Conf. on Mobile Ubiquitous Comput., Syst., Services and Technologies (UBICOMM)

Farshchi F, Huang Q, Yun H (2020) Bru: Bandwidth regulation unit for real-time multicore processors. In: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 364–375, https://doi.org/10.1109/RTAS48715.2020.00011

Ghaemi G, Tarapore D, Mancuso R (2021) Governing with Insights: Towards Profile-Driven Cache Management of Black-Box Applications. In: Brandenburg BB (ed) 33rd Euromicro Conference on Real-Time Systems (ECRTS 2021), Leibniz International Proceedings in Informatics (LIPIcs), vol 196.

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp 4:1–4:25, https://doi.org/10.4230/LIPIcs.ECRTS.2021.4, URL https://drops.dagstuhl.de/opus/volltexte/2021/13935

Green Hills Software (2023) GHS Integrity. https://www.ghs.com/products/rtos/integrity_virtualization.html

Hassan M, Patel H, Pellizzoni R (2017) Pmc: A requirement-aware dram controller for multicore mixed criticality systems. ACM Trans Embed Comput Syst 16(4). https://doi.org/10.1145/3019611, URL https://doi.org/10.1145/3019611

Hebbache F, Jan M, Brandner F, et al (2018) Shedding the shackles of time-division multiplexing. In: 2018 IEEE Real-Time Systems Symposium (RTSS), pp 456–468, https://doi.org/10.1109/RTSS.2018.00059

Hoornaert D, Roozkhosh S, Mancuso R (2021) A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic. In: Brandenburg BB (ed) 33rd Euromicro Conference on Real-Time Systems (ECRTS 2021), Leibniz International Proceedings in Informatics (LIPIcs), vol 196. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp 2:1–2:22, https://doi.org/10.4230/LIPIcs.ECRTS.2021.2, URL https://drops.dagstuhl.de/opus/volltexte/2021/13933

Houdek P, Sojka M, Hanzálek Z (2017) Towards predictable execution model on arm-based heterogeneous platforms. In: 2017 IEEE 26th International Symposium on Industrial Electronics (ISIE), pp 1297–1302, https://doi.org/10.1109/ISIE.2017.8001432

Intel, Corp. (2016) Intel's Stratix 10 FPGA: Supporting the smart and connected revolution. URL https://newsroom.intel.com/editorials/intels-stratix-10-fpga-supporting-smart-connected-revolution/, accessed on 2022-01-19

Jalle J, Quiñones E, Abella J, et al (2014) A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study. In: 2014 IEEE Real-Time Systems Symposium, pp 207–217, https://doi.org/10.1109/RTSS.2014.23

Jun M, Bang K, Lee HJ, et al (2007) Slack-based bus arbitration scheme for soft real-time constrained embedded systems. In: 2007 Asia and South Pacific Design Automation Conference, pp 159–164, https://doi.org/10.1109/ASPDAC.2007.357979

Kim H, Rajkumar RR (2016) Real-Time Cache Management for Multi-Core Virtualization. In: Proceedings of the 13th International Conference on Embedded Software. Association for Computing Machinery, New York, NY,

USA, EMSOFT '16

Kloda T, Solieri M, Mancuso R, et al (2019) Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), p 1–14

Kostrzewa A, Saidi S, Ernst R (2016) Slack-based resource arbitration for real-time networks-on-chip. In: 2016 Design, Automation Test in Europe Conference Exhibition (DATE), pp 1012–1017

Lelli J, Scordino C, Abeni L, et al (2016) Deadline scheduling in the Linux kernel. Software: Practice and Experience 46(6):821–839

Li Y, Akesson K, Goossens K (2016) Architecture and analysis of a dynamically-scheduled real-time memory controller. Real-Time Systems 52(5):675–729. https://doi.org/10.1007/s11241-015-9235-y

Mancuso R, Dudko R, Betti E, et al (2013) Real-time cache management framework for multi-core architectures. In: 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), p 45–54

Mancuso R, Pellizzoni R, Tokcan N, et al (2017) WCET Derivation under Single Core Equivalence with Explicit Memory Budget Assignment. In: 29th Euromicro Conference on Real-Time Systems (ECRTS 2017), Leibniz International Proceedings in Informatics (LIPIcs), vol 76. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp 3:1–3:23, https://doi.org/10.4230/LIPIcs.ECRTS.2017.3, URL http://drops.dagstuhl.de/opus/volltexte/2017/7168

Martins J, Tavares A, Solieri M, et al (2020) Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems. In: Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020), OpenAccess Series in Informatics (OASIcs), vol 77. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp 3:1–3:14, https://doi.org/10.4230/OASIcs.NG-RES.2020.3, URL https://drops.dagstuhl.de/opus/volltexte/2020/11779

Microsemi — Microchip Technology Inc. (2020) PolarFire SoC - Lowest Power, Multi-Core RISC-V SoC FPGA. URL https://www.microsemi.com/product-directory/soc-fpgas/5498-polarfire-soc-fpga, accessed on 09.01.2020

Mirosanlou R, Hassan M, Pellizzoni R (2020) Drambulism: Balancing performance and predictability through dynamic pipelining. In: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 82–94, https://doi.org/10.1109/RTAS48715.2020.00-15

Modica P, Biondi A, Buttazzo G, et al (2018) Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms. In: 2018 IEEE International Conference on Industrial Technology (ICIT), pp 1651–1657, https://doi.org/10.1109/ICIT.2018.8352429

Nicolella M, Roozkhosh S, Hoornaert D, et al (2022) Rt-bench: An extensible benchmark framework for the analysis and management of real-time applications. In: Proceedings of the 30th International Conference on Real-Time Networks and Systems. Association for Computing Machinery, New York, NY, USA, RTNS 2022, p 184–195, https://doi.org/10.1145/3534879.3534888, URL https://doi.org/10.1145/3534879.3534888

Roozkhosh S, Mancuso R (2020) The potential of programmable logic in the middle: Cache bleaching. In: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 296–309, https://doi.org/10.1109/RTAS48715.2020.00006

RTCA Inc. (2011) RTCA/DO-178C Software Consideration in Airborne Systems and Equipment Certification

Saeed A, Dasari D, Ziegenbein D, et al (2022) Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores . In: 2022 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), p 133–145

Schwaericke G, Tabish R, Pellizzoni R, et al (2021) A Real-Time virtio-based Framework for Predictable Inter-VM Communication. In: 2021 IEEE International Real-Time Systems Symposium (RTSS)

Serrano-Cases A, Reina JM, Abella J, et al (2021) Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC. In: Brandenburg BB (ed) 33rd Euromicro Conference on Real-Time Systems (ECRTS 2021), Leibniz International Proceedings in Informatics (LIPIcs), vol 196. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp 3:1–3:26, https://doi.org/10.4230/LIPIcs.ECRTS.2021.3, URL https://drops.dagstuhl.de/opus/volltexte/2021/13934

Siemens AG (2023) Jailhouse hypervisor. https://github.com/siemens/jailhouse, accessed: 2023-06-06.

Sohal P, Tabish R, Drepper U, et al (2020) E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management. In: 2020 IEEE Real-Time Systems Symposium (RTSS)

SYSGO G (2023) PikeOS Hypervisor. https://www.sysgo.com

Tabish R, Wen J, Pellizzoni R, et al (2021) An analyzable inter-core communication framework for high-performance multicore embedded systems. Journal of Systems Architecture p 102178. https://doi.org/https://doi.org/10.1016/j.sysarc.2021.102178, URL https://www.sciencedirect.com/science/article/pii/S1383762121001284

Valsan PK, Yun H (2015) Medusa: A predictable and high-performance dram controller for multicore based embedded systems. In: 2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications, pp 86–93, https://doi.org/10.1109/CPSNA.2015.24

Venkata SK, Ahn I, Jeon D, et al (2009) Sd-vbs: The san diego vision benchmark suite. In: IISWC. IEEE Computer Society, pp 55–64, URL http://dblp.uni-trier.de/db/conf/iiswc/iiswc2009.html#VenkataAJGLGBT09

Xilinx (2022) ZCU 102 MPSoC TRM. https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm/Zynq-UltraScale-Device-Technical-Reference-Manual, accessed: 2022-11-08

Xilinx (2023) Xilinx Versal. https://www.xilinx.com/products/silicon-devices/acap/versal.html, accessed: 2021-10-13.

Yun H, Yao G, Pellizzoni R, et al (2013) Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In: 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 55–64, https://doi.org/10.1109/RTAS.2013.6531079

Yun H, Pellizzoni R, Valsan PK (2015) Parallelism-aware memory interference delay analysis for cots multicore systems. In: 2015 27th Euromicro Conference on Real-Time Systems, pp 184–195, https://doi.org/10.1109/ECRTS.2015.24

Yun H, Yao G, Pellizzoni R, et al (2016) Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms. IEEE Transactions on Computers 65(2):562–576

Yun H, Ali W, Gondi S, et al (2017) BWLOCK: A Dynamic Memory Access Control Framework for Soft Real-Time Applications on Multicore Platforms. IEEE Transactions on Computers 66(7):1247–1252

Zhou Y, Wentzlaff D (2016) Mitts: Memory inter-arrival time traffic shaping. In: Proceedings of the 43rd International Symposium on Computer Architecture. IEEE Press, ISCA '16, p 532–544, https://doi.org/10.1109/ISCA.2016.53, URL https://doi.org/10.1109/ISCA.2016.53

Zuepke A, Bastoni A, Chen W, et al (2023) Mempol: Policing core memory bandwidth from outside of the cores. In: 2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 235–248, https://doi.org/10.1109/RTAS58335.2023.00026