# On the Interplay of Computation and Memory Regulation in Multicore Real-Time Systems

Denis Hoornaert*, Golsana Ghaemi†, Andrea Bastoni*, Renato Mancuso†, Marco Caccamo*, and Giulio Corradi‡
*Technical University of Munich †Boston University ‡Xilinx
*{denis.hoornaert, andrea.bastoni, mcaccamo}@tum.de, †{golsana, rmancuso}@bu.edu, ‡giulioc@xilinx.com

***Abstract*—The ever-increasing demand for high-performance in the time-critical embedded domain has pushed the adoption of powerful yet unpredictable heterogeneous Systems-on-a-Chip. The shared memory subsystem, which is known to be a major source of unpredictability, has been extensively studied, and many mitigation techniques have been proposed. Among them, performance-counter-based regulation techniques have seen widespread adoption. However, combining performance-based regulation with time-domain isolation is a relatively unexplored approach that requires the assessment of limitations and benefits.**

**In this article, we outline our current work-in-progress on SHCReg (Software Hardware Co-design Regulator), a full-stack hardware/software co-design architecture that aims to improve the interplay between CPU and memory isolation for mixed-criticality tasks running on the same core.**

***Index Terms*—Criticalities, Real-time, Hypervisor, Budget-based Regulation**

## I. Introduction

The real-time community has proposed many successful techniques to mitigate the impact of inter-core memory interference (*e.g.*, [6], [7]). Notably, performance counter (PMC) based techniques such as *Memguard* [7] have received significant attention due to their practicality. In fact, PMC-regulation techniques are used to establish *temporal isolation* by mitigating the problem of non-arbitrated memory bandwidth sharing between cores. In the embedded and real-time domain, these techniques are often implemented within a partitioning hypervisor (*e.g.*, Jailhouse [2]) when consolidation of multiple RTOSs onto the same multicore system-on-a-chip (MPSoC) is required. At the same time, when consolidating complex applications with mixed-criticality requirements onto MPSoC with rich OSs like Linux, CPU provisioning still remains a fundamental dimension. Here, server abstractions –*e.g.*, the Constant Bandwidth Server (CBS) [1]– are well known and widely used, with the `SCHED_DEADLINE` [5] policy being the most popular example.

Despite combining CBS-based CPU scheduling and PMC-regulation to achieve isolation in *both* time and memory domains being a logical choice, effective integration proves to be challenging. The need to enact the CBS at the task level (*i.e.*, in the OS) and PMC-regulation at the CPU level (and hence in the hypervisor) results in a lack of coordination between the two mechanisms. This leaves the system incapable of handling what we refer to as *memory overload conditions*. These correspond to all the cases where high-critical tasks are still eligible for scheduling in the OS, but unable to use the necessary memory bandwidth because throttled via PMC-regulation at the hypervisor level.

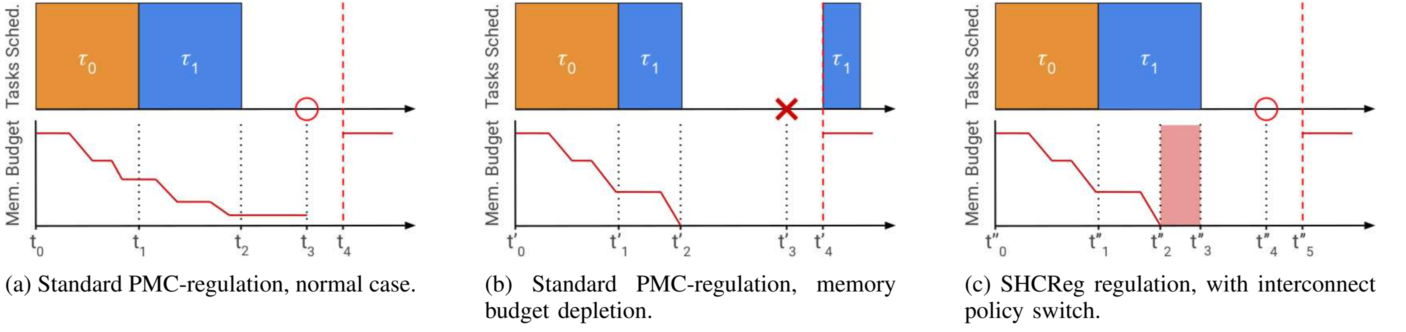
Fig. 1 illustrates a run-time scenario where such overload can happen. The considered system is composed of one high and one low criticality task (respectively $\tau_1$ and $\tau_0$) scheduled using CBS to absorb execution variations. The common PMC-budget assigned is determined beforehand via profiling and the addition of a fixed *safety margins*, a common practice in industrial applications. While in Fig. 1a, $\tau_1$ is able to complete on time, in Fig. 1b it experience extra blocking due to the lack of sufficient *memory budget* caused by an increased memory consumption of $\tau_0$. Such an increase can be due to changing computational needs that require additional memory accesses (for example, consider the case of object detection or object tracking in an almost empty street or at a very crowded intersection). We note that such an increase in memory consumption cannot be determined apriori without resorting to very pessimistic over-estimations.

Our proposed Software Hardware Co-design Regulator architecture (SHCReg) tackles exactly this issue. Under SHCReg (Fig. 1c), when an overload is detected, the critical memory accesses of $\tau_1$ are prioritized *at hardware level* by switching the policy of the interconnect to main memory from a fair round-robin to a priority-based one. Consequently, $\tau_1$ can further execute and meet its deadline. Priorities are assigned depending on the criticality of the tasks running on each core. The idea is to facilitate $\tau_1$'s completion (possibly at the expanses of other cores) and quickly restore the standard isolation property of the system.

We envision implementing SHCReg on the Xilinx ZCU102 development board leveraging available tools including Linux, Memguard on Jailhouse,[1] and SchIM [4]. The envisioned hardware/software co-design architecture and the strategy employed are outlined in this work-in-progress.

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant number CCF-2008799. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF. Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research.

[1]`https://github.com/rntmancuso/jailhouse-rt`




(a) Standard PMC-regulation, normal case.

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

(c) SHCReg regulation, with interconnect policy switch.

Fig. 1: Example scenario of a PMC-regulated core, where an increased memory consumption causes $\tau_1$ to miss its deadline.

## II. Proposed Regulation Policy

Due to an early depletion of the memory budget $A_k$ on core $CPU_k$ (see Fig. 1b), $CPU_k$ could experience a *memory overload*. Formally, a *memory overload* occurs if a critical task $\tau_i$, running on $CPU_k$ under server- and PMC-regulation, is unexpectedly stalled because its memory budget $A_k$ has depleted, while its associated server $S_i$ is still eligible for execution.

When one or more $CPU_k$ experience a memory overload, SHCReg reacts by prioritizing their memory transactions over those of other cores. The following rules control the regulation:

1) By default, the interconnect is configured to use a fair policy (*i.e.*, similar to round-robin), and each $CPU_k$ memory budget accounting is done following standard PMC-regulation rules.
2) When $CPU_k$ exhausts its memory budget $A_k$, it is stalled until $A_k$ is replenished unless: i) the criticality of the running task $\tau_i$ is high, or ii) a high critical task is released while $CPU_k$ is stalled (*i.e.*, before the next replenishment period). In these cases, the core experiences a ***memory overload***.
3) Upon a *memory overload*, the interconnect policy $\pi$ is switched to fixed-priority ($FP$), and each $CPU_k$'s bus priority is set according to the criticality of the executed task. (We assume a finite set of task's criticalities.)
4) A $CPU_k$ leaves the *memory overload* state when the high critical task has completed or when the memory budget $A_k$ is replenished. When all CPUs have left a *memory overload*, the interconnect policy is set to $Fair$.
5) Each $CPU_k$ with leftover memory budget always continues its memory budget accounting until budget depletion is reached. Only critical tasks $\tau_i$ can execute even when the memory budget is depleted thanks to the *memory overload* policy.

Interestingly, when considered alone, the individual regulation mechanisms employed by SHCReg are insufficient 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. But unfortunately, the safe (conservative) usage of PMC-regulation alone inevitably leads to under-utilization of the already scarce memory bandwidth. 2) On the other hand, statically prioritizing CPUs when they access main memory (*e.g.*, [4]) 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.

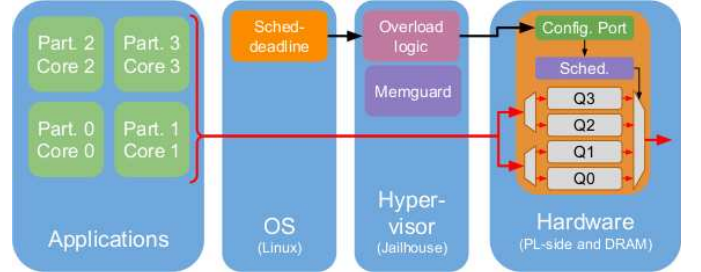



Fig. 2: SHCReg layered architecture

## III. Architecture

We target systems that consolidate different RTOSs on top of (lightweight) partitioning hypervisors. SHCReg implements therefore a layered architecture such as the one depicted in Fig. 2. CPU regulation is completely implemented in software at the operating system level, while memory regulation requires a hardware/software co-design, and its implementation is distributed across the hypervisor software level and the hardware-based control of the data link to the main memory (see red arrows in Fig. 2). Furthermore, lightweight communication between layers is required to propagate, for example, information on the criticality of the currently executing tasks (see black arrows in Fig. 2).

The target platform for SHCReg is the Xilinx ZCU102 UltraScale+ development board,[2] a Linux and Hypervisor capable embedded platform associating a tightly integrated programmable fabric (*i.e.*, FPGA) with a traditional Processing System composed of a CPU cluster.

### A. 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

[2]Any PS-PL platform being hypervisor and Linux capable is eligible.

scheduling policy (*e.g.*, [3]) that provides isolation among the tasks. We use Linux as OS to prototype our architecture. In Linux, the `SCHED_DEADLINE` scheduling policy [5] realizes a Constant Bandwidth Server. We associate each task to a server and define its maximum utilization. Each server is statically assigned to one of the CPUs.

### B. Memory Regulation

The 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, as depicted in Fig. 2.

*1) PMC-regulation and Memory Overload Detection:* The hypervisor implements a PMC-regulation mechanism limiting the maximum number of memory transactions towards the main memory the cores can issue. Implementing PMC-regulation at the hypervisor level makes the PMC-regulation transparent to the OS level, and it allows using potentially different OSs while ensuring adequate memory bandwidth control. In addition, the proposed architecture allows different OSs to use different types of CPU server regulation. The belief is that separating the PMC-regulation level from the CPU regulation level is a clean and sensible architectural choice.

*2) Dynamic FP/Fair Interconnect Policy:* The lowest part of the memory regulation realized by SHCReg is implemented in hardware extending the architecture of SchIM [4].[3]

The SchIM module is implemented on the PL-side and acts as an intermediate step on the data path between cores and DRAM. Similarly to [4], all CPU-originated memory transactions are redirected to the PL-side and to SchIM. As shown in Fig. 2, each core is associated with a queue storing the memory transactions directed to DRAM. Fig. 2 illustrates that CPU-originated transactions are split into two input links, each being shared by two CPUs. Under heavy traffic, the queuing of the transactions enables SchIM to schedule them 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. 2). 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. A scheduler can be selected by operating on a set of registers accessible by the whole system through a memory-mapped configuration port. We extended the original SchIM by enabling the dynamic choice of a specific scheduler at run-time and by adding the $Fair$ scheduling policy.

### C. Design Choices

The synchronization and the communication between the layers constitute a critical performance hurdle of our architecture. It is particularly the case regarding the interplay between the memory budget and the CPU budget (respectively enforced at the hypervisor- level and OS-level).

Considering rules (2) and (4) in Sec. II, the release of a critical task while a CPU is stalled and dynamically switching the priority of the interconnect when a critical task is completed requires careful synchronization between the OS and the hypervisor. For example, while a hypercall can be used by the operating system to signal the completion of critical task, synchronizing the complex high-resolution timers in Linux with the PMC-regulation logic (at hypervisor level) to detect the release of a high-critical task during a memory- regulation phase is more complex. Either direction introduce foreseeable run-time overheads that might limit the potential benefits of the proposed architecture. Furthermore, the implementation of PMC-regulation in hypervisors such as Jailhouse-rt[4] is realized with interrupt-nesting in hypervisor-context. Its synchronization with the expiration of Linux's `hrtimers` is therefore particularly challenging and might require considerable hypervisor changes. Nonetheless, while slow, hypercalls completion times can be bounded. On the other hand, not using a hypervisor in the middle would automatically prevent the conception of consolidated systems using various OSs or RTOSs.

## IV. Conclusion and Future Work

We presented our work-in-progress on SHCReg, a hardware/software co-design that aims to solve the *memory overload* problems of real-time workloads with variable memory requirements on architectures that feature both CPU and PMC-based regulations. Our targets are real-world systems that consolidate multiple different RTOSs on a single MPSoC leveraging on hypervisor technologies.

Despite the technical challenges on the design side, we believe that, by exploiting the capability of dynamically change the policy of the interconnect, SHCReg could provide real-time and performance benefits for a wide class of workloads.

## References


[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 4–13, 1998.

[2] Siemens AG. Jailhouse hypervisor. https://github.com/siemens/. Accessed: 2021-02-08.

[3] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag, 2011.

[4] Denis Hoornaert, Shahin Roozkhosh, and Renato Mancuso. A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:22, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2021/13933.

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

[6] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 45–54, 2013.

[7] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2016.


[3] `https://github.com/denishoornaert/MemorEDF`

[4] `https://github.com/rntmancuso/jailhouse-rt`