Lazy Load Scheduling for Mixed-criticality Applications in Heterogeneous MPSoCs

Tomasz Kloda, Giovani Gracioli, Rohan Tabish, Reza Mirosanlou, Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo

Lazy Load Scheduling for Mixed-Criticality Applications in Heterogeneous MPSoCs

TOMASZ KLODA, LAAS-CNRS, Université de Toulouse, INSA, France
GIOVANI GRACIOLI, Federal University of Santa Catarina, Brazil
ROHAN TABISH, University of Illinois at Urbana-Champaign, USA
REZA MIROSANLOU, University of Waterloo, Canada
RENATO MANCUSO, Boston University, USA
RODOLFO PELLIZZONI, University of Waterloo, Canada
MARCO CACCAMO, Technical University of Munich, Germany

Newly emerging multiprocessor system-on-a-chip (MPSoC) platforms provide hard processing cores with programmable logic (PL) for high-performance computing applications. In this paper, we take a deep look into these commercially available heterogeneous platforms and show how to design mixed-criticality applications such that different processing components can be isolated to avoid contention on the shared resources such as last-level cache and main memory.

Our approach involves software/hardware co-design to achieve isolation between the different criticality domains. At the hardware level, we use a scratchpad memory (SPM) with dedicated interfaces inside the PL to avoid conflicts in the main memory. Whereas, at the software level, we employ a hypervisor to support cache-coloring such that conflicts at the shared L2 cache can be avoided. In order to move the tasks in/out of the SPM memory, we rely on a DMA engine and propose a new CPU-DMA co-scheduling policy, called *Lazy Load*, for which we also derive the response time analysis. The results of a case study on image processing demonstrate that the contention on the shared memory subsystem can be avoided when running with our proposed architecture. Moreover, comprehensive schedulability evaluations show that the newly proposed *Lazy Load* policy outperforms the existing CPU-DMA scheduling approaches and is effective in mitigating the main memory interference in our proposed architecture.

CCS Concepts: • **Computer systems organization** → **Real-time systems**; **Other architectures**; **Embedded systems**; **System on a chip**;

Additional Key Words and Phrases: Mixed-criticality real-time systems, heterogeneous multiprocessor systems-on-chip, schedulability analysis.

ACM Reference format:
Tomasz Kloda, Giovani Gracioli, Rohan Tabish, Reza Mirosanlou, Renato Mancuso, Rodolfo Pellizzoni, and Marco Caccamo. 2021. Lazy Load Scheduling for Mixed-Criticality Applications in Heterogeneous MPSoCs. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2021), 26 pages.
https://doi.org/DOI

1539-9087/2021/1-ART1 $15.00
https://doi.org/DOI

1 INTRODUCTION

New emerging technologies like autonomous driving, unmanned aerial vehicles, cube satellites, or smart manufacturing are significant examples of modern real-time systems. Unlike the past CPU-intensive tasks, the workloads in today's mission- and safety-critical systems are characterized by much higher memory and I/O performance demands [10].

Hardware manufacturers have anticipated this shift by extending the multiprocessor systems-on-chip (MPSoC) feature set, including hardware support for virtualization, the presence of multiple, potentially heterogeneous processing elements, a rich ecosystem of high-bandwidth I/O devices and communication channels, and more recently, the co-location of traditional CPUs and programmable logic (PL) implemented using Field Programmable Gate Array (FPGA) technology. This new class of platforms offers the unprecedented ability to define new hardware components that can bring determinism and tight latency bounds to real-time memory-intensive applications, closing the gap between performance and real-time guarantees [17].

Our previous work in [17] demonstrated how to leverage the latest generation of partially re-configurable MPSoCs to design high-performance embedded systems with strict real-time requirements. We showed that it is possible to instantiate a critical set of PL-defined components to (i) relieve interference on the shared memory hierarchy and achieve temporal isolation among criticality domains; (ii) support efficient inter-domain communication; (iii) co-locate a traditional task execution model with a multi-phase execution model; and (iv) overcome typical limitations of traditional memory partitioning techniques.

However, no scheduling mechanism was integrated into the system model proposed in [17]. In this work, we present a new scheduling technique for the proposed mixed-criticality architecture based on a multi-phase task model to close the gap between the system design and theory. The PL-based scratchpad that we employ can reduce memory inter-core interference but cannot guarantee the same level of latency reduction as the standard, located close to the processor, scratchpad memories, or caches that were used in the previous works implementing the multi-phase model [42, 48]. Therefore, we propose a new scheduling technique that induces less low-priority task blocking when compared with state-of-the-art approaches proposed in [45, 49], and can take full advantage of our architecture. To summarize, the main contributions are:

(1) We extend our previous work [17] by proposing a new scheduling policy, called *Lazy Load*, as well as a scheduler design and a schedulability analysis for real-time tasks running on top of modern MPSoC platforms using a multi-phase execution.
(2) Compared to previous schedulability results in [45, 46, 48], the scheduling techniques proposed in this work improve the schedulability performance for event-triggered mixed-criticality applications (even 50% of improvement in terms of schedulability ratio). We evaluate the proposed scheduling policy and contrast it with existing scheduling policies for multi-phase task sets using synthetic task sets and hardware overheads that were measured.
(3) Differently from the previous three-phase models [48], which used TDMA arbitration with fixed slot sizes, we propose a TDMA mechanism with a finer granularity that allows splitting long memory transactions over multiple TDMA slots.
(4) We present an overview of the implementation, evaluation, and main results from our previous paper [17], including an overview for the design and implementation of a hardware block, named address translator, that prevents memory waste when cache partitioning based on page coloring is used.

The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 introduces the adopted system model and assumptions. Section 4 presents the response time analysis for the new scheduling policy, *Lazy Load*. Section 5 discusses the design principles and

overviews the implementation. Section 6 compares previous implementation results and shows the evaluation of the new schedulability analysis. Finally, Section 7 concludes the paper.

2 RELATED WORK

Shared resource handling. Several recent works have proposed techniques to deal with shared resources in multicore real-time systems at both OS and hypervisor levels. Cache partitioning based on page coloring was used by several works to improve the predictability of multicore real-time systems [16, 21, 52]. Page coloring together with cache locking was proposed in [28]. Similarly, some other works focused on making DRAM accesses more predictable [20, 23, 62, 63].

Regarding the use of hypervisors in multicore real-time systems, Modica *et al.* [31] proposed a hypervisor-based architecture targeting critical systems similar to ours [17], including cache partitioning for spatial isolation and DRAM bandwidth reservation for temporal isolation. The techniques were implemented in the *XVISOR* open-source hypervisor and tested in a quad-core ARM A7 processor [33]. Our hypervisor-based architecture, instead, explores the existence of PL to handle data transfers between the processing system and programmable logic and data prefetching. PL together with a processing system was first introduced in [27] to reduce interference of mixed-criticality applications in uniprocessors without shared caches.

Other approaches used features available on modern multicore processors to handle contention among the cores. *MARACAS* [61], for instance, used hardware performance counters (HPCs) information to regulate the memory bandwidth of threads. Crespo *et al.* also used HPCs together with control theory to regulate the memory bandwidth of critical and non-critical cores. Awan *et al.* [2] proposed a memory regulation mechanism for mixed-criticality applications. *vCAT* used the Intel's *Cache Allocation Technology* (*CAT*) to provide cache partitioning for the hypervisor and virtual machines [58]. However, this approach depends on a specific hardware feature and uses non-real-time basic software support (Linux and Xen). *vLLC* and *vColoring* were two hypervisor techniques proposed to enable cache-aware memory allocation for individual tasks running in a virtual machine [22]. *CHIPS-AHOy* integrates hardware isolation mechanisms, such as memory partitioning, with an observe-decide-adapt loop to achieve predictability, energy and thermal management in a holistic hypervisor [32].

PRedictable Execution Model. Other research works proposed different task execution model to bound or eliminate the contention for shared resources. The PRedictable Execution Model (PREM) [3, 7, 34, 36, 53] splits the task execution into two separate phases, one dedicated for memory transactions and another one for pure computation. During the memory phase, the data required by a task is fetched from the shared main memory to a fast local memory (either a cache or a scratchpad memory - SPM). During the computation phase, a task used the prefetched data without the need to access the main memory. A memory scheduler is responsible for ensuring that tasks do not overlap their memory phases. Several works [54–56] leverage the fact that the time of memory fetches carried out together is less than the combined cost of individual cache misses. The PREM's loading phase takes the same advantage. However, as explained below, it also goes one step further by allowing the cost of the load operations to be hidden.

Three-phase model. The original PREM model was later extended by the Acquisition Execution Restitution (AER) [12] and three-phase [5, 48] models. Both models consist of a load phase, in which code/data is loaded from main memory to the scratchpad (SPM), before a task starts, an execution phase, and an unload phase in which code/data of the task is unloaded from the SPM to main memory. A DMA component is responsible for the loading and unloading. The SPM is divided in two halves, allowing one task to execute in one half, while DMA is active on the another one, thus hiding the latency of loading and unloading phases. Due to its ability to avoid contention

at the memory level and the applicability to platforms that have SPM memories, we use the three-phase model in this work.

Scheduling approaches in the three-phase model. Several works have implemented different scheduling approaches within the *AER* or the *three-phase* models, ranging from round-robin [14] or TDMA [17, 48, 53, 59] arbitration among processors, to static [1, 3, 12, 29, 40, 41] or priority-based [30, 36, 60] schedule among tasks.

The SPM-centric scheduling policies considered in the previous works load the data for the next task to be scheduled on a CPU either at the beginning of the current task's CPU computation phase [46, 53] or when an SPM partition becomes free [48, 49]. This can result in the blocking from the low priority tasks. Our scheduling policy reduces the blocking from low priority tasks by postponing the load decision until the current task enters the final part of its execution long enough to overlap the loading phase that is going to be scheduled.

Recently, in [9], the authors addressed the problem of reducing the priority inversion introduced in the multi-phased task scheduling policies. When a latency-sensitive task is released, an ongoing lower priority task loading phase is aborted, and the processor prefetches the newly released task data. This is orthogonal to our approach, where the low-priority task blocking is reduced by postponing the scheduling decisions until the last time instant when the memory transaction can be hidden with the remaining computation. The schedulability analysis in [9] is formulated as a mixed-integer linear programming optimization problem.

In [41], the authors proposed an offline scheduling optimization technique to hide the communication delay for parallel periodic real-time tasks in the *three-phase* model. The scheduling technique selects the SPM contents offline to hide the cost of SPM loading/unloading. Our work focuses on run-time scheduling instead. Similarly, [42] proposes a memory-centric scheduler for *PREM*-compliant tasks that do not rely on any hardware support. The work used fixed-priority scheduling and proposed a global memory preemption scheme to improve the system schedulability. Although the proposed work has some similarities to ours (such as the use of a hypervisor), our work targets the three-phase model and leverages a hardware with programmable logic.

An extension to the three-phase model to support streaming tasks that allows overlapping the memory and computation phases of segments of the same task is presented in [45]. The approach is implemented at the compiler level (using *LLVM*) together with an RTOS API to handle load/unload requests.

3 SYSTEM MODEL AND ASSUMPTIONS

3.1 Criticality Domains

Our goal is to implement multiple *criticality domains* on a single multicore SoC. We consider a system with up to C criticality domains, in which C is also the total number of cores in the SoC. Thus, each core can have its own static criticality domain, isolated from each other, both in time and space [8].

We consider three types of criticality domains: (i) a *low-criticality domain* running a general-purpose operating system (OS) – *e.g.,* Linux – responsible for handling I/O with complex devices, processing large amounts of data, and using general-purpose libraries and applications. No strong temporal guarantees can be expressed due to the best-effort nature of the software stack; a *high-criticality domain* responsible for running hard real-time tasks with simple I/O devices; and (iii) a *mid-criticality domain* responsible for running tasks with intermediate criticality. Within this domain, and unlike the low-criticality domain, temporal guarantees for real-time tasks are still provided; however, the degree of hardware resource isolation offered to the mid-criticality domain

is lower when compared to the high-criticality one. The number of cores allocated to high- and mid-criticality domains is M ($M \leq C$).

3.2 Processor and Programmable Logic

We consider an embedded MPSoC platform with two main subsystems, the processor subsystem (PS) and the programmable logic (PL), and a communication engine, as detailed below.

Processor Subsystem (PS): The PS has a multicore embedded processor with C cores. Each core has a private Level-1 (L1) cache, and all the cores share a Level-2 (L2) cache, which is also the last level cache (LLC). We adopt a widespread model in modern multicore embedded systems, although other memory hierarchy organizations are possible. Because our goal is to define strongly isolated criticality domains, we assume that hardware support for virtualization exists in the PS.

Programmable Logic (PL): The PL is an on-chip block of Field Programmable Gate Array (FPGA) cells that coexists with the embedded PS cores. We consider systems where high-bandwidth, low-latency memory interfaces connect the PS to the PL and vice-versa. While we assume that one or more PS-PL interfaces exist, it cannot be assumed that at least C interfaces are available. The number and capacity, in terms of memory throughput, of the PL-PS interfaces directly impact the performance and degree of temporal isolation that can be enforced among criticality domains. The FPGA can also provide different memory blocks, such as scratchpad (SPM) and PL-side DRAM. Examples of existing MPSoC platforms that fit into our system model are the Intel Stratix 10 SoC FPGA, Intel Arria 10 SoC FPGA, Intel Cyclone SoC FPGA, Xilinx Ultrascale+ ZCU102, and Xilinx Zynq-7000.

Communication Engine: We assume that a Direct Memory Access (DMA) component is available in either the PS or the PL, and it can act as the communication engine to transfer memory from/to PL and PS memories. Differently from the previously implemented three-phase solution in [48], which used TDMA arbitration with fixed slot sizes, we propose a TDMA mechanism with finer granularity and per-core slots of different sizes. In this scheme, each real-time core j is assigned a slot size σ_j, with $\mathcal{T} = \sum_{j=1}^{M} \sigma_j$ being the length of the TDMA round. We do not require the slots to be sized based on the SPM dimension; instead, if a DMA phase cannot finish within a slot, we break it down into multiple transfers and perform them over multiple TDMA rounds. The price we pay is extra overhead: since it takes some time to re-program the DMA controller, during each slot we can only perform DMA transfers for a maximum of $\bar{\sigma}_j$ time. Hence, $(\sigma_j - \bar{\sigma}_j)$ represents the DMA overhead. Assume that two consecutive (un)load phases require k TDMA slots. Then it is easy to see that the total transfer time Δ is upper bounded by:

$$\Delta = k \cdot \mathcal{T} + \sigma_j; \tag{1}$$

the core receives one slot every $\mathcal{T}$ time, but its initial slot can be wasted if the first memory phase arrives just after the beginning of the slot.

3.3 Application Model

We make no assumption on the behavior of applications operating in low-criticality domains. They can perform complex I/O operations, and they can be arbitrarily memory intensive. Mid-/high-criticality applications are structured as real-time tasks: a sequence of jobs whose activation is time- (periodic) or event-triggered (sporadic). Mid-/high-criticality applications are also statically assigned to cores, and locally scheduled using fixed-priority non-preemptive scheduling. Inter-task communication is performed via message passing. Only input data —from other tasks or devices— available by a given job's activation instant are used by the job itself. Similarly, output data are produced by a job only at its completion. We formalize the scheduling model in the next subsection.

We assume that the memory footprint of mid-/high-criticality tasks is limited. On the one hand, this allows to place code and data of real-time applications onto local memories of constrained size. On the other hand, it allows to load and unload applications in and out of local memories —following scheduling decisions— without incurring high overheads. Tasks follow the *three-phase* model, as introduced in Section 2.

3.4 Scheduling Model

A system consists of a finite set of sporadic real-time tasks statically allocated to single processors. Each task gives rise to a potentially infinite sequence of jobs released sporadically after some minimum inter-arrival time T_i, and each job of τ_i must complete within a fixed time interval from its release given by a relative deadline $D_i \leq T_i$ (*i.e.,* constrained deadlines). Each task τ_i follows a three-phase model and is hence composed of three consecutive non-overlapping phases: a load phase (L-phase), a computation phase (C-phase) and an unload phase (U-phase). The DMA performs the load and the unload phases and the processor performs the computation phases. The task's code and data are first loaded into the scratchpad during its L-phase. Then, the task is executed on the processor during its C-phase. Eventually, after the end of the task's computation phase, the task's final results are unloaded from the scratchpad back to the main memory during its U-phase. Both DMA and processor can handle only one task at a time. We denote by C_i the worst-case execution time (*WCET*) of τ_i computation phase, by L the longest time needed by any task to load its code, private and input data into the scratchpad using DMA, and by U the longest time needed by any task to unload its computation results from the scratchpad to the main memory using DMA. We assume that load and unload phase execution times already include the DMA access delays related to the shared memory bus arbitration (*e.g.,* see Equation (1) for TDMA-arbitrated access). All of the aforementioned parameters are positive integers. We assume that a scratchpad is large enough to accommodate the code and data of any two tasks at a time. Since the DMA operations do not involve the processor, a task load or unload phase can overlap with another task's computation phase. The task τ_i worst-case response time R_i (*WCRT*) is the longest response time from task release to completion of its unload phase for any of its jobs. A task set is said to be schedulable if all jobs of all tasks always complete unload phases before their respective deadlines, *i.e.,* $R_i \leq D_i$.

Tasks are individually scheduled on each processor (*i.e.*, partitioned scheduling) by a fixed-priority non-preemptive scheduler. Task priorities are unique. We introduce notation $hp(i)$ and $lp(i)$ for the set of tasks with priorities, respectively, higher than, and lower than the priority of task τ_i assigned to the same processor as τ_i. Furthermore, we introduce notation $hep(i) = hp(i) \cup \{\tau_i\}$ for the set of tasks with priorities higher than or equal to the priority of task τ_i that are assigned to the same processor as τ_i.

The scheduler selects the jobs for the execution on CPU and DMA. While the CPU executes a computation phase of the task with its code and data stored in one scratchpad partition, the DMA engine can reload another partition (*i.e.,* unload the results of the completed task and load the code and input data for the next task). The scheduling decisions are made as late as possible: L time units before the end of the current task computation phase, the DMA is programmed to load the task with the highest priority (*Lazy Load*). The unload operations are programmed immediately after the end of the task computation phase. If there is no active task on the CPU, the scheduler is invoked at the first task release. If the task execution time is shorter than the time needed to reload the scratchpad partition, we inflate the task execution time to the end of the scratchpad reload and consider as still running. Section 5.7 describes our *Lazy Load* policy in more detail.

Compared to [48, 53], where the scheduling decisions are made earlier (*i.e.*, the next task load phase starts when the current task computation phase starts, *Eager Load*), our approach reduces

the low-priority task blocking and, as shown by our experiments in Section 6.3, improves the system schedulability. On the downside, our scheduling algorithm requires the knowledge of the worst-case execution times of the particular tasks and might result in the increase of the average response times (*e.g.,* if a computation phase executes for L time units less than its worst-case execution time, the next load phase cannot overlap with the computation phase).

Figure 1 shows two schedules for the same sequence of tasks' releases: one for our *Lazy Load*, shown in the lower part, and one for the standard *Eager Load*, shown in the upper part. It should be noted that in this example, we assume that the tasks execute with their worst-case execution times and can have different load and unload phases lengths. The DMA and CPU activities are shown in the respective axes, separately for *Eager* and *Lazy Load*, and the scheduling events (*e.g.,* task release, task completion, task load, etc.) are shown in the *Sched*-axis. Three real-time tasks, high-priority τ_1, mid-priority τ_2, and low-priority τ_3, are released, respectively, at time instants, t_3, t_1, and t_0.

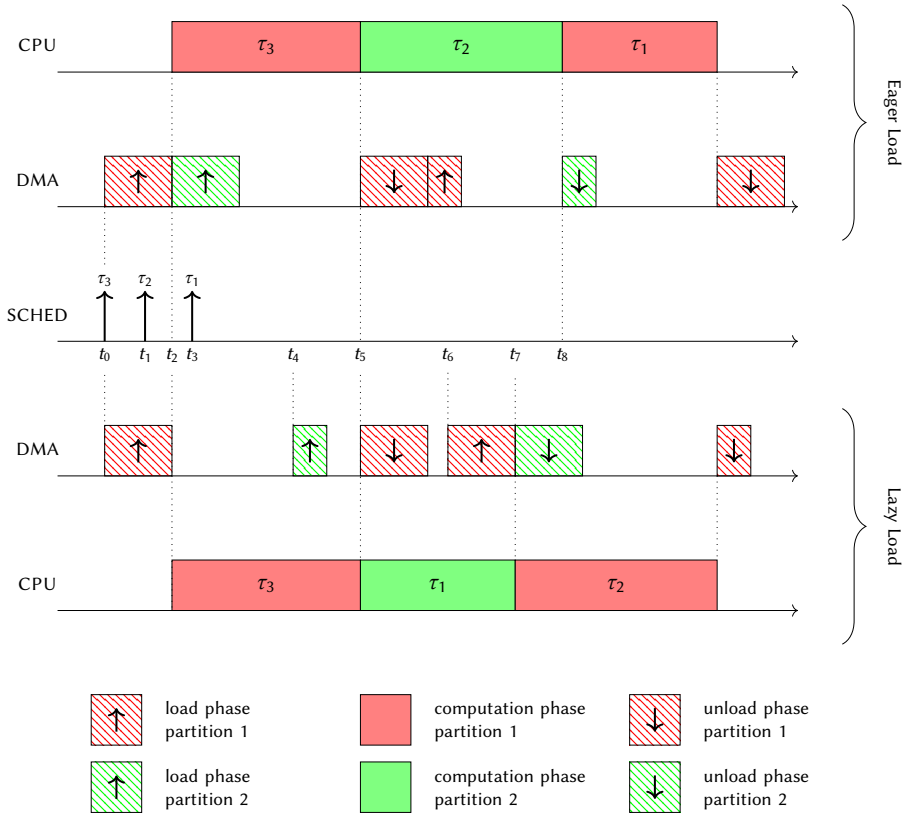


Fig. 1. Scheduling algorithm for three-phase task model under *Eager* and *Lazy Load* approaches.

Upon the first task release at t_0, the system is idle, and both scratchpad partitions are empty, and the scheduler immediately starts loading τ_1's data into the scratchpad. The loading completes at t_2, and the task τ_3 computation phase starts. In the *Eager Load*, since the job of task τ_2 released at time instant t_1 is already pending, the DMA starts loading τ_2's data into the second scratchpad partition. In contrast, under the *Lazy Load* approach, the DMA scheduling decision is postponed

until the time t_4: as the high-priority task τ_1 was released at t_3, its data will be loaded into the scratchpad instead of mid-priority task τ_2, and the job of τ_1 will start at t_5. Under *Eager Load*, the τ_1's jobs must wait for the τ_2 completion and starts at the time t_8. A high-priority job of τ_1 suffers from priority-inversion blocking caused by two jobs (τ_2 and τ_3). The *Lazy Load* reduces the priority-inversion blocking to one lower-priority job (τ_3) and results in a shorter response time of the high-priority job of τ_1. In the next section, we characterize the worst-case blocking for *Lazy Load* and derive a proper response time analysis.

4 SCHEDULABILITY ANALYSIS

We now introduce the response time analysis for the three-phase task model under the *Lazy Load* scheduling policy described in Section 3.4. Since we employ partitioned scheduling for real-time tasks, we focus only on the core executing task under analysis τ_i. We do not consider single-task sets as the task worst-case response time is straightforward to obtain in this case: $R_1 = L + C_1 + U$.

The scheduling problem of the *Lazy Load* policy is similar to non-preemptive fixed-priority scheduling on a single processor. The difference is that the scheduling decisions are made L time units before the end of current task execution, while in the classic non-preemptive fixed-priority scheduling, these decisions are made at task completion. We first derive the bounds on the three-phase task processing and blocking times. Using these bounds, we characterize the busy-period in the context of the three-phase model and derive the upper bounds on task response times.

Processing Time: A computation phase can run in parallel with at most one unload and one load phase. The maximal time that can elapse between the start of task τ_j computation phase and the start of the next task computation phase is given by:

$$\widehat{C}_j = \max(C_j, L + U), \tag{2}$$

where $L + U$ is the maximal time it takes to reload the scratchpad partition content (for the TDMA specific delays, please refer to Equation (1) or [49]).

Blocking: The non-preemptive scheduling policy might introduce blocking due to *priority inversion*. A job must wait for the last L time units of the current job execution to start its loading phase. If the job is released right after the start of the lower priority job loading phase, then the blocking is maximal, and any $hep(i)$ task computation phase start is delayed by no more than:

$$L + B_i, \tag{3}$$

where $B_i = \max\{\widehat{C}_j | \tau_j \in lp(i)\}$ is the longest scheduling interval with a computation phase of the task having a priority lower than τ_i. If τ_i has the lowest priority, then we consider $B_i = L + U$ as the processor can be idle for at most one load and one unload. Consider a task that arrives too late to be loaded (*i.e.,* within the L-window before the current task completion), and the memory time for loading is totally wasted. The next DMA operation is an unload, and only then can the ready task start its load phase. During that time, the processor remains idle.

Critical Instant: With the above-obtained bounds on task execution (Equation (2)) and task blocking (Equation (3)), we can now reduce the schedulability problem of the three-phase model to the non-preemptive fixed-priority scheduling. A *critical instant* for a task is a task arrival instant at which that task has the longest response time [26]. For our transformed model, task τ_i's critical instant is the synchronous release of all $hp(i)$ tasks when the longest low-priority blocking B_i has just started. The reasoning is the same as in [26]. Advancing the $hp(i)$ job release would not increase its interference on τ_i. Releasing the $hp(i)$ job before the task τ_i's critical instant could increase the interference on τ_i only if the $hp(i)$ task could be blocked or suspended. However, the task cannot suspend and all the tasks that might increase τ_i's response time are taken into account. These tasks are $lp(i)$ tasks—and the blocking that they can introduce—is captured by B_i,

the other $hp(i)$ tasks, and the task τ_i itself (the analysis, if necessary, covers more than one job of τ_i as further explained below).

Busy Window: A level-i busy window is a contiguous time interval within which jobs of priority τ_i or higher are processed [25]. Bril et al. [6] and Davis et al. [11] showed that under non-preemptive fixed-priority scheduling, all task instances within the task's busy window should be verified. The self-pushing might cause a second or later task instance to have a longer response time than the first task instance. Task τ_i during its non-preemptive execution might block the higher priority tasks more than the lower priority tasks at the critical instant. Hence, at the next τ_i release, more high-priority task interference can be accumulated (*i.e., knock-on effect*).

As the scheduling decisions are made earlier than the current task completion, the priority inversion can occur more than once within the task busy window. Consider, for instance, that L time units before task τ_i completion there are only $lp(i)$ jobs pending. A $hp(i)$ job can arrive later while τ_i is still running on the CPU, but the DMA has already been programmed for an $lp(i)$ job, leading to a priority inversion. However, if there are no $hp(i)$ jobs pending L time units before the τ_i completion, then the jobs released later cannot be blocked more than at the τ_i's critical instant (see Formula (3)). Therefore, we can consider the i-level busy window until no more than L computation units are pending. The length of the i-level busy window W_i can be upper bounded by the minimum positive integer satisfying the following recurrent relation:

$$W_i = L + B_i + \sum_{j \in hep(i)} n_j(W_i - L) \cdot \widehat{C}_j, \tag{4}$$

where

$$n_j(t) = \left\lceil \frac{t}{T_j} \right\rceil. \tag{5}$$

is the maximal number that task τ_j jobs that can be released in any interval of length $t > 0$ and the convergence condition for the iteration for Equation (4) is:

$$\sum_{j \in hep(i)} \frac{\widehat{C}_j}{T_j} < 1 \tag{6}$$

If the above condition is satisfied (*i.e.,* the processor is not infinitely busy with the $hep(i)$ jobs), we can solve Equation (4) using iteration starting with $W_i = \widehat{C}_i$. To find the task τ_i worst-case response time, we must check its $\lceil W_i / T_i \rceil$ first instances within the longest i-level busy window.

Worst-Case Response Time: We compute the task τ_i k-th instance worst-case response time upper bound $R_{i,k}$. Figure 2 illustrates the notation used in the further analysis (task τ_i k-th instance load phase start $l_{i,k}$, computation start $s_{i,k}$, unload start $u_{i,k}$, and finish time $f_{i,k}$).

Let $l_{i,k}$ and $s_{i,k}$ be respectively the task τ_i k-th instance loading and computation phase start. The computation phase starts L time units after the load phase starts:

$$s_{i,k} = l_{i,k} + L \tag{7}$$

All $hp(i)$ tasks released before $l_{i,k}$ must be loaded and executed before $s_{i,k}$:

$$\begin{aligned} s_{i,k} &= L + B_i + \sum_{j \in hp(i)} n_j(l_{i,k}) \cdot \widehat{C}_j + (k-1) \cdot \widehat{C}_i \\ &= L + B_i + \sum_{j \in hp(i)} n_j(s_{i,k} - L) \cdot \widehat{C}_j + (k-1) \cdot \widehat{C}_i \end{aligned} \tag{8}$$

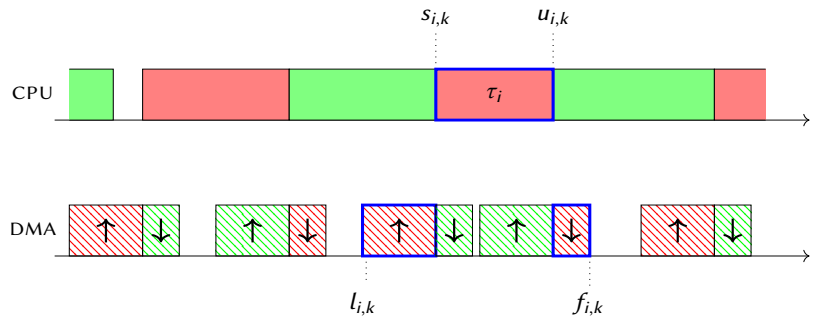


Fig. 2. Task τ_i response-time analysis for three-phase tasks scheduling policy under *Lazy Load*.

The solution of the above equation can be found through iterations with the initial value of $s_{i,k} = L + B_i + (k-1) \cdot \widehat{C}_i$. The k-th instance of task τ_i starts its unload phase at or before:

$$u_{i,k} = s_{i,k} + \widehat{C}_i \tag{9}$$

which completes at or before:

$$f_{i,k} = u_{i,k} + U \tag{10}$$

The worst-case response time of the k-th instance of task τ_i is upper-bounded by:

$$R_{i,k} = f_{i,k} - (k-1) \cdot T_i \tag{11}$$

Finally, the task τ_i worst-case response time upper bound is given by:

$$R_i = \max_{k \in \lceil W_i / T_i \rceil} R_{i,k} \tag{12}$$

5 DESIGN AND IMPLEMENTATION OVERVIEW

5.1 Design Overview

Figure 3 represents the ideal software stack and assignment of resources to domains. The main idea is to provide spatial and temporal isolation to higher-criticality domains. Thus, a lower-criticality domain cannot interfere with a higher-criticality one. The opposite, however, although undesirable, may happen.

A thin static partitioning hypervisor provides isolation to each domain in self-contained address spaces. The partitioning hypervisor has a number of roles, including (1) providing spatial isolation for RTOSes that do not support virtual memory; (2) partitioning cores to criticality domains; (3) enforcing LLC partitioning via page coloring[1] [15]; (4) performing tasks' relocation to/from DRAM into local memories; and (5) providing message-passing channels for inter-domain communication.

To prevent the memory waste caused by cache coloring, we leverage the Programmable Logic (PL) and propose a *bus translator* to prevent coloring-induced memory waste and, to avoid the contention for the shared main memory, we define new hardware components in PL. Programmable Logic (PL). We use dual-ported memories that are only accessible by a single criticality domain and dedicated a PL-PS interface to criticality domains. On each PL-PS interface, we instantiate two memory controllers inside the PL (one handling the accesses from application cores and another one handling the accesses from the DMA).

Finally, to support task relocation when data and code are loaded/unloaded to/from DRAM/SPM, we propose to compile tasks against absolute intermediate physical addresses (IPA). Then, after

[1]In this work we use the terms cache coloring and page coloring interchangeably.

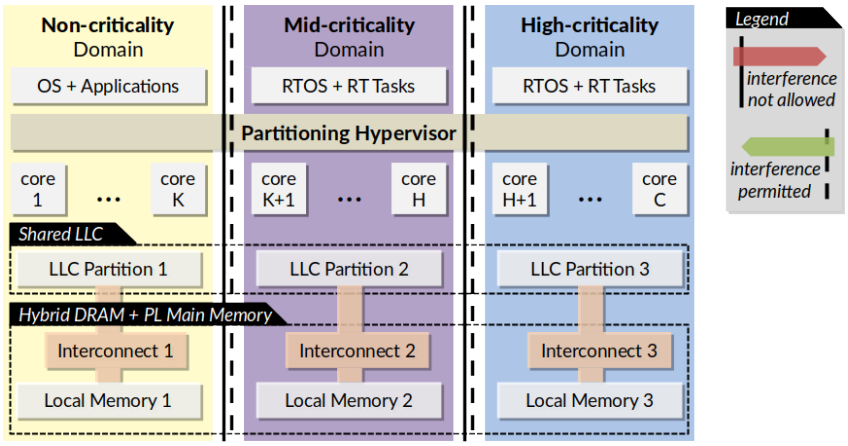


Fig. 3. Ideal software and hardware stack organization.

the communication engine has located a new task at a potentially new physical location in local memory, a hypervisor routine is invoked to map the new physical addresses (PAs) to the set of IPAs against which tasks have been compiled. In the next subsections, we present the implementation details of our design decisions.

5.2 Architectural Overview of the Chosen Platform

For our implementation, we have used the Xilinx UltraScale+ ZCU102 MPSoC [57]. On this platform, the PS comprises two ARM Cortex-R5 cores, each having its own tightly coupled memory of 128 KB. There are also four application (ARM Cortex-A53) cores, each having its own local instruction and data cache (32 KB each). The Last Level Cache (LLC) of 1 MB is shared by all application cores. There is no dedicated SPM provided for the application cores. The PS includes a DDR4-2666 (main memory) controller with a data bus width of 64-bit connected to a 4GB DDR4 memory module. The PL includes a separate, 16-bit synthesized memory controller wired to a 512 MB DDR4 memory module.

Multiple interfaces between the PL and the PS exist. There are three interfaces going from the PS [2] to the PL. Out of the three, two are high-performance master interfaces (HPM0 and HPM1), whereas the third interface is the low-performance domain (LPD) interface. There are also interfaces from the PL to the PS, specifically the high-performance coherent (HPC) and high-performance (HP – non-coherent). Finally, there are 3 MB of block RAM (BRAM) inside the PL. For the rest of the paper, we will use BRAM and SPM interchangeably.

5.3 Implementation Overview

Based on the design space exploration carried out in [17], our final hardware design is depicted in Figure 4. We assign one of the A53 cores to be a low-criticality core, two of them to be mid-criticality cores, and one of them to be a high-criticality core. The mid- and high-criticality cores run their own Real-Time Operating Systems (RTOS). A few noticeable features of our proposed design are: (i) the low-criticality domain is assigned direct access to PS DRAM, because this domain features applications with sizable footprints; (ii) each mid- and high-criticality domain is assigned a private

[2]Here the direction of the interface indicates which side of the system can initiate transactions towards the other side. On an interface from PS to PL, the PS is the master of the interface, while the PL is the slave.

SPM; (iii) each of these SPMs is dual-ported, and a controller is instantiated on each port to prevent contention between DMA and core at the SPM controller; and (iv) the high-criticality domain also occupies a dedicated PS-PL interface to access its private SPM. In our platform, the maximum size of all SPMs is 3 MB. Thus, we set the size of the SPM used by the high-criticality domain to 2 MB, while the size of the other two SPMs used by mid-criticality domains was set to 512 KB each.

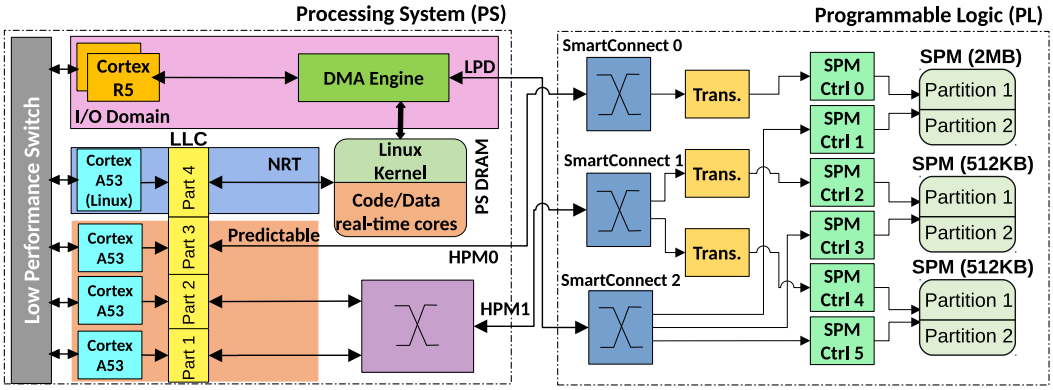


Fig. 4. Proposed system design and usage of PS-PL interfaces. Note the placement of the hardware translator blocks (PL-side, in yellow) between the SmartConnects and SPM controllers.

We propose creating separate SPM in the PL for all the mid- and high-criticality cores. Thus, a dedicated or fast interface such that each core can access its own SPM without seeing a delay from another core is required. Unfortunately, there are only two high-performance (HPM) interfaces between PL and PS available in the platform and three A53 cores. Therefore, in our design, we assign one shared high-performance interface to two A53 cores while the third core has a dedicated interface to its own SPM memory (see Figure 4). A low-performance domain (LPD) interface is assigned to the DMA engine to transfer data to/from SPM/DRAM. The HPM and LPD interfaces are connected to the dual-ported SPMs to allow the execution of a currently running task and the loading/unloading performed by the DMA. The scheduling of the loading and unloading DMA operations is handled by the R5 core in the I/O domain.

In order to avoid the contention between A53 cores in different criticality domains, we partition the LLC via coloring. The use of coloring generally results in portions of physical memory being unusable to applications. This is generally acceptable for main memory because its size is not constrained (few GBs). Conversely, SPMs in the PL are usually limited in size (few KBs or MBs). For instance, if coloring is used to define four equally sized LLC partitions, this would reduce the size of each SPM to 1/4. To avoid this side effect of coloring, we introduce an address translator between the A53 and the SPM. Since the cache is physically indexed, coloring both the PS DRAM and SPM is required to avoid interference (otherwise, there would be a cache interference at every SPM access).

In the following subsections, we provide a brief discussion on each of the main components that form our architecture. For a complete overview, please refer to [17].

5.4 Jailhouse and Page Coloring

We use Jailhouse as a hypervisor because it provides static partitioning of hardware resources and low-overhead, which is ideal for hard real-time systems [35]. Jailhouse runs as a Linux driver, thus

requiring at least one core to be assigned to Linux —the root cell. Once the driver is loaded, it takes control of the entire hardware and reassigns a partitioned view of the hardware resources back to Linux, based on a configuration file. We assign non-critical tasks to the Linux cell, while critical tasks run on isolated partitions (cells) on top of an RTOS. The RTOS used for mid-/high-criticality domains is Erika Enterprise version 3, which is open-source and OSEK/VDX certified [13]. Erika supports fixed-priority scheduling and has a porting for the Xilinx Ultrascale+ platform.

To enforce strong inter-domain (inter-cell) and hence inter-core performance isolation, we leverage page coloring [15]. We use the virtualization extensions of the processor to implement coloring by enforcing appropriate restrictions on the color of pages that Jailhouse maps to intermediate physical addresses (IPAs) of virtualized cells. Specifically, we impose that physical pages with non-overlapping colors are assigned to cells activated on different cores. The advantage of this approach is twofold: (i) it allows us to localize the changes required to implement coloring-based partitioning in a single software component (Jailhouse); and (ii) it allows deploying unmodified and possibly closed-source OS inside our criticality domains. A similar technique was used in [22, 24, 31]. A publicly available version of Jailhouse that implements cache coloring is the Jailhouse-RT project [43, 44].

5.5 Address Translator to Overcome Limitations of Cache Coloring

To overcome the problem of memory waste imposed by coloring, we designed an address translation hardware IP. The component performs physical address translation for memory transactions originating from the PS towards the PL. To better understand how the component operates, let us consider our specific setup.

To access an SPM with a size of 2 MB, 21 bits of the address are provided for requests originated from the PS. With cache coloring enabled (and four colors, one for each core), only one in four memory pages can be used, with addresses aligned at 16 KB boundaries (each page has a size of 4 KB). The adopted solution is the following. Instead of receiving 21 bits of an address, the translator IP receives 23 bits (8 MB) from the PS, removes the specific color bits from that, and passes it to the SPM controller.

Given the geometry of the LLC (1 MB, 16 ways), the color bits that can be used to perform partitioning are bits 12 to 15 of each physical address. To create four partitions, one could use bits 12 and 13. Pages with bits [12, 13] = 0b0 would be assigned to partition 1; pages with bits [12, 13] = 0b1 to partition 2; and so on. In this way, four sequential physical pages will be assigned to four different partitions. This is not ideal, however, because the L1-Data cache in this platform is *Physically Indexed, Physically Tagged* (*PIPT*), and fits two pages per way. If a CPU is only given access to one every four pages, only half of the L1-D cache will be utilized. To avoid this problem, we use bits 14 and 15 as the LLC color bits. In this configuration, each partition is given four consecutive pages.

Let us assume that the address of the translator in Figure 4 responds under the address range `0xA0000000` to `0xA07FFFFF` (8 MB). Following the discussion above, bits 14 and 15 are used as LLC coloring bits. Figure 5 shows an example where a request address of `0xA0023456` (offset `0x023456`) from a core arrives to the translator IP. Bits 14 and 15 of the offset are dropped by the translator, and the resulting offset is `0x0B456` in a 2 MB non-colored space.

This PL-aided address translation is a special case of the *cache bleaching* technique presented in [39]. Apart from address manipulation, memory transaction scheduling [19] and on-the-fly data reorganization [38] are other possible PL-aided management strategies for scratchpad data. Moreover, additional performance improvements when accessing in-PL scratchpad data can be unlocked by leveraging *coherence backstabbing* and the CAESAR approach described in [37]. The

use of the aforementioned more advanced techniques, however, is currently out of the scope of this paper.

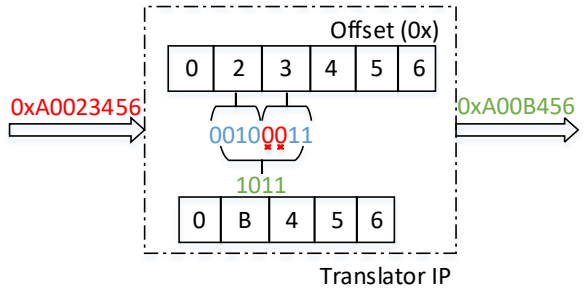


Fig. 5. Translator IP operation. The two most significant bits from the fourth byte (in red) of the input address are dropped.

In our design (Figure 4), there are three translators to handle the requests coming from each core. With this mapping mechanism, the SPM capacity is not affected by the cache coloring (we do not lose space), and since the translator IP is burst-capable, we do not lose bandwidth nor increase latency in accessing the SPMs. Besides that, the area overhead of the module in terms of the numbers of Flip-Flops (FF) and Lookup tables (LUTs) compared with the design without any translation IP are 0.57% and 0.41%, respectively, while the block RAM cell count remains the same.

5.6 Code/Data Relocation

We use code/data relocation to support the loading and unloading of Erika tasks' code and data. Relocation is initiated by the Erika RTOS when its scheduler decides to load or unload a task as required. Recall, however, that applications in Erika are statically compiled against a set of virtual addresses (or intermediate physical addresses, since Erika does not support virtual memory). As such, relocation is performed by modifying the mapping from intermediate physical addresses to physical addresses (IPA→PA) managed by Jailhouse [24].

Erika first informs Jailhouse that a relocation must be performed. This is done via a hypercall (*i.e.,* hvc assembly instruction), which was added to Erika. In Jailhouse, two new hypercalls were added to handle either load or unload operations. The source/destination address, the offset in pages from the beginning of the SPM where the task needs to be loaded to/unloaded from, and the size of the task that needs to be loaded/unloaded are passed as parameters to the hypercalls.

Once Jailhouse receives a request to relocate a task's code/data, it performs the following steps. First, it determines the absolute source (resp., destination) in DRAM and destination (resp., source) in SPM for a load (resp., unload) operation. Next, it modifies the IPA→PA mapping so that the range of intermediate physical addresses starting at the provided source address (resp., destination) and spanning for the number of pages specified by the size parameter, map to the destination address. After the remapping is completed, Jailhouse returns control to Erika. The effective copy of the task into/from SPM is performed by the DMA engine.

5.7 Lazy Load Scheduler Support

The most straightforward implementation of the proposed *Lazy Load* policy is to rely on a time-triggered approach: when a task starts its computation phase, the next load phase is programmed L time units before the task's worst-case finishing time but not earlier than after U time units (*i.e.,* in the case that the task worst-case execution time is shorter than $L + U$). The unloading phase of

the completed job is programmed at the current job's computation phase start. If the system is idle and there are no pending jobs, the new task load phase starts immediately.

The time-triggered approach can result in the processor under-utilization when the tasks execute faster than their worst-case execution times (*i.e.,* the actual execution time can be less than the worst-case execution time). To avoid unnecessary processor stall, the next load operation can be triggered immediately if the current computation phase finishes earlier. Note that if there is a way to estimate an early completion of the task at run-time, then loading no earlier than L time before the end of the task is safe. In what follows, we detail the scheduler implementation using this approach. Figure 6 depicts the scheduler and the various states that a task can lie in during execution.

The scheduler maintains three queues: *load queue*, *ready queue*, and *unload queue*. The tasks in the *load* and *unload queues* are waiting for the DMA, respectively, to load and unload their code and data into/from a scratchpad partition, while the tasks in the *ready queue* are waiting for the CPU to start the computation. The *load queue* capacity should be sufficient to hold all tasks while the *ready* and *unload queues* should only hold a single task. A task can be in the waiting state in each queue, as well as in the *load* (*i.e.,* DMA is loading task code and data), *run* (*i.e.,* CPU is executing task computation phase), and *unload* state (*i.e.,* DMA is unloading the task data). Since we assume a single DMA engine and a partitioned system where tasks are assigned to a single processor, there can only be one task in the *run* state and one task in either the *unload* or *load* state at any given time. Efficient implementation requires an alarm timer that triggers the load of the next task *L* time units before the latest finish time of the running task.

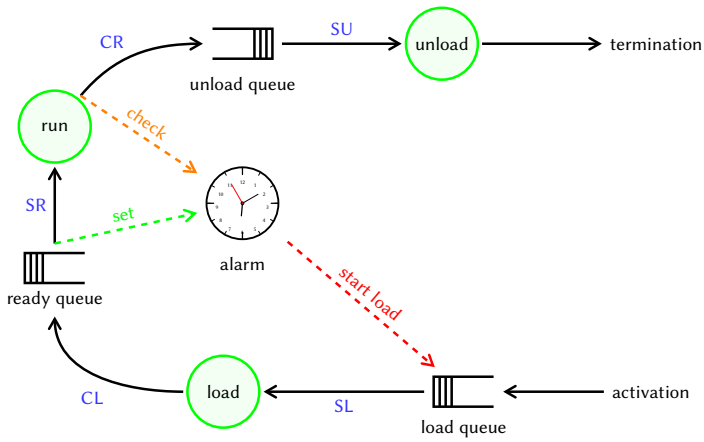


Fig. 6. Task states and transitions in *Lazy Load* CPU-DMA scheduler for three-phase task model.

Whenever the processor becomes idle, and there is a ready task in the *ready* queue, the ready task computation phase is dispatched for the execution (SR). The computation start triggers an unload of the previously completed task (SU). We denote the task in the *run* state by τ_{run}. When τ_{run} starts execution, and the scratchpad is full, the timer alarm is set to $t_{load} = \max\{f_{run} - L, s_{run} + U\}$ where s_{run} is the start of the τ_{run}'s computation phase and $f_{run} = s_{run} + C_{run}$ is the τ_{run}'s worst-case finish time. If only one scratchpad partition is occupied (*i.e.,* there is no need to unload another scratchpad partition), the timer is set to $t_{load} = \max\{f_{run} - L, s_{run}\}$. A timer expiration signal triggers a *load* of a task with the highest priority among all tasks in the *load queue*, if any (SL). If τ_{run}

completes after t_{load}, then the DMA starts the τ_{run}'s unload immediately if the DMA is idle or after the end of the ongoing load operation. If τ_{run} completes before t_{load}, the timer is disarmed, and a *load* of the highest priority task from the *load queue* is triggered (SL). Task τ_{run} is placed into the *unload queue* (CR) and, as soon as the DMA becomes available, its unload starts (SU).

Implementation Overhead. As demonstrated in Figure 6, the implementation overhead is composed of the activities to manage the queues (load, unload, and ready queues), plus the overhead of programming a timer and its interrupt service routine (ISR). We have measured such overheads (obtained worst-case time from 1000 repetitions): the measured worst-case timer programming overhead is 3.89 μs, the worst-case ISR overhead is 989 ns, and the time to dispatch the queue requests is 717 ns. Thus, the total worst-case implementation overhead is 5.59 μs.

6 EVALUATION

In this section, we present the evaluation of our system design and the proposed schedulability test. We start showing an evaluation of the DMA performance, including the time to transfer different data sizes from PS DRAM to the SPM and its programming overhead. We then present the schedulability analysis evaluation through randomly generate synthetic task sets.

6.1 DMA Evaluation

The DMA engine in our architecture implements a fine granularity TDMA-based scheduling to move data between the PS DRAM and SPM memory located in the PL. The DMA scheduling runs on an ARM Cortex-R5 core as a bare-metal firmware (generated using the `armr5-none-eabi-gcc` compiler with `-DARMR5 -W -Wall -O0 -g3` flags). To avoid contention between DMA transfers and application cores, the DMA uses the dedicated low-power domain (LPD) interface.

We measured the DMA transfer time for different data sizes, extracting the average transfer time, standard deviation (STD), and the worst-case transfer time among 1000 samples. Table 1 shows the obtained results. Recall that 1 MB represents half the size of the largest SPM in our design. The obtained standard deviation varies from 0.057 to 0.1. The bandwidth increases proportionally to the amount of contiguous memory transferred.

Table 1. DMA transfer time (in μs) and bandwidth for different data sizes.

Transfer Size	Transfer Time			Bandwidth (MB/s)
	Average (μs)	STD	Worst-case (μs)	
2 KB	4.92	0.057	5.11	397.0
4 KB	7.15	0.04	7.27	546.3
8 KB	11.63	0.01	12.01	671.8
9.1 KB	12.91	0.05	13.11	688.4
16 KB	20.62	0.08	20.96	757.8
22 KB	27.42	0.10	27.72	783.5
32 KB	38.52	0.05	38.81	811.3
1 MB	1149.44	0.05	1149.78	870.0

We denote the time to program and start a DMA transfer as the DMA programming overhead. Considering all the experiments, the worst-case DMA programming overhead we obtained was 3.89 μs. For small data sizes (2 and 4 KB, for instance), the relation between the programming overhead and the transfer time is significant. In this case, it may be beneficial to avoid small data transfer whenever possible or use the own task's core instead of the DMA. We would like to point

out that the model behaves well as long as task execution times are longer than the time required to reload an SPM partition. As an example, if we consider a partition of 256 KB (half the size of a 512 KB scratchpad) and a TDMA slot with a transfer size of 32 KB for each core, then based on Equation (1), we obtain $\sigma_j = 38.81 + 3.89 = 42.7\ \mu$s, $\mathcal{T} = 3 \cdot 42.7 = 128.1\ \mu$s, and $k = 2 \cdot 256/32 = 16$ as the number of slots required to unload/load the partition. This results in a memory reload time $\Delta = 2092.3\ \mu$s, meaning that tasks should execute for at least 2.1 ms to hide the memory time.

6.2 Case Study: Image Processing

To evaluate our system design, we consider a system where video frames captured from a camera are processed in a high-criticality domain. Video frames are processed using the `disparity` benchmark from the *SD-VBS* suite [51]. `Disparity` computes depth information for objects represented in two input images, obtaining relative positions of objects in the scene. This kind of algorithm is useful in applications such as cruise control, pedestrian tracking, and collision control [51]. The objective of this evaluation is to demonstrate how the proposed system behaves in a realistic setup and to show its limits in terms of achievable hard real-time guarantees.

To this end, the `disparity` benchmark is executed as a periodic task. During each activation, it computes the `disparity` of two input images. At every new period, `disparity` reuses one image from the previous iteration and uses a new image transferred by the communication engine. We performed two experiments with two different image resolutions, *i.e.,* 64x48 and 128x96 (SQCIF). We only used these image resolutions due to limitations in the size of the SPM. Also, `disparity` requires input images in the bitmap image file (BMP) format, which is uncompressed. Thus, for a resolution of 64x48, an image has a size of around 9.1 KB, while for 128x96 an image has a size of 22 KB. We use a set of 20 images of a scene from the KITTI vision benchmark suite dataset [47] (the 2015 stereo multiview dataset). The original images had a resolution of 1241x376. We converted the frames to the lower resolutions described above. We move the I/O data of the tasks from/to DRAM to/from the SPM at the load/unload phase of the task using the same approach as described in [49]. Table 1 shows the DMA transfer time for both image resolutions (9.1 KB and 22 KB). Erika RTOS consumes 294 KB of memory (including data and code) and it is fixed on the SPM (we do not load nor unload code/data of the RTOS). `Disparity` using image resolution of 64x48 consumes 349 KB, while for 128x96 it consumes 745 KB, also including data and code. Although not required in this case study, note that when input data is too large to fit into the SPM, it is possible to use compiler-level techniques to break the load/unload phases into small chunks [46].

We considered four scenarios as described in [17]: Lcy-Solo, Lcy-Stress, Our-Solo, and Our-Stress. We run `disparity` alone in the system from the PS DRAM on top of Linux (Lcy-Solo), next `disparity` runs from the PS DRAM with three bandwidth (BW) benchmark instances [18] also executing and accessing the PS DRAM (Lcy-Stress). The `disparity` benchmark is then executed from SPM on top of Erika/Jailhouse with coloring and using our hardware design without (Our-Solo) or with (Our-Stress) interference from the rest of the system. Ideally, when `disparity` runs with contention from the SPM (Our-Stress), it should exhibit comparable performance with respect to the case when `disparity` runs without interference from the SPM (Our-Solo). The case when `disparity` runs solo from PS DRAM (Lcy-Solo) serves as a baseline, while the case when it runs from PS DRAM under contention (Lcy-Stress) provides an idea of what we gain in terms of isolation and performance thanks to the proposed set of software/hardware techniques. Periodic execution of the `disparity` task was achieved under Linux by using a `CLOCK_REALTIME` timer to invoke a handler at the desired frequency. The handler then releases the `disparity` thread using a semaphore. The `disparity` benchmark, Erika OS, and the BW benchmark instances were compiled using gcc version 5.4 for the ARM64 architecture with the `-O2` flag.

First, we present the execution time of `disparity` in each of the four cases using an image resolution of 64x48 in Table 2 and a resolution of 128x96 in Table 3. We measured the execution time of 1000 individual processing jobs and extracted the average execution time, standard deviation (STD), BCET, WCET, and variability window. The variability window is calculated as $(WCET_{stress} - BCET_{solo})/WCET_{stress}$. Time measurements were taken using the processor cycle counter and converted to *ms*. Note that when working at 64x48 resolution, the two input images (9 KB each) fit into the L1 cache (32 KB). Thus, the observed worst-case when `disparity` is running alone is similar for both memories (PS DRAM and SPM). However, when contention is introduced, the benchmark suffers visible interference in the LCY-STRESS setup. Note that there is still some contention when `disparity` uses the dedicated HPM interface and cache coloring in the OUR-STRESS setup. This may be due to contention over Miss Status Holding Registers (MSHRs) in the last level cache [50].

Table 2. Average, standard deviation, BCET, and WCET obtained from 1000 executions for the considered four cases with input image size of 64x48. All values in *ms*. Highlighted values in bold are used to calculate the variability window.

	LCY-SOLO	LCY-STRESS	OUR-SOLO	OUR-STRESS
Average	15.89	17.86	15.94	16.49
STD	0.01	0.07	0.01	0.06
BCET	**15.88**	17.69	**15.92**	16.34
WCET	16.00	**18.18**	15.96	**16.73**
Var. Window	12.6%		4.8%	

Table 3. Average, standard deviation, BCET, and WCET obtained from 1000 executions for the considered four cases with input image size of 128x96. All values in *ms*. Highlighted values in bold are used to calculate the variability window.

	LCY-SOLO	LCY-STRESS	OUR-SOLO	OUR-STRESS
Average	61.50	75.09	66.04	69.80
STD	0.02	0.34	0.07	0.26
BCET	**61.45**	74.32	**65.79**	69.04
WCET	61.80	**77.09**	66.30	**70.59**
Var. Window	20.2%		6.8%	

Based on the observed WCET in the various experiments, we vary the image processing task period and study when `disparity` starts missing deadlines in each case. Table 4 presents the obtained results for image size of 64x48. We vary the frequency from 55 Hz (18.18 ms period) to 63 Hz (15.87 ms period). A tick mark in the table indicates that the desired image processing rate was sustainable. In other words, that no instance of `disparity` missed its relative deadline (equal to the period). In contrast, a cross mark indicates that the desired rate was not sustainable. From the results in Table 4, we can see that by running `disparity` without any interference, the maximum sustainable rate is 62 Hz. However, when running under contention and with no isolation enforcement (LCY-STRESS case), the sustainable image processing rate drops to 55 Hz. Conversely, a rate of 59 Hz is sustainable if `disparity` executes from within a high-criticality domain defined using the proposed software/hardware techniques. Note that in this setup, each image processing job has to wait for an image to be transferred in input by the DMA before it can start execution. Because DMA accesses to DRAM can experience contention, a decrease in sustainable rate is

visible between the Lcy-Solo and the Lcy-Stress cases. Nonetheless, this experiment shows that our design provides better predictability and enables higher processing rates when the system is under heavy load.

Table 4. Supported frequencies for image size of 64x48.

Freq. (Hz)	Period (ms)	Lcy-Solo	Lcy-Stress	Our-Solo	Our-Stress
55	**18.18**	✓	✓	✓	✓
56	**17.86**	✓	✗	✓	✓
57	**17.54**	✓	✗	✓	✓
58	**17.24**	✓	✗	✓	✓
59	**16.95**	✓	✗	✓	✓
60	**16.67**	✓	✗	✓	✗
62	**16.13**	✓	✗	✓	✗
63	**15.87**	✗	✗	✗	✗

Table 5 shows results for input images with resolution 128x96 when running the `disparity` benchmark. The average execution time for `disparity` with image resolution of 128x96 when running solo from PS DRAM is 61.5 *ms* — see Table 3, Lcy-Solo case. Thus, we vary the frequency from 10 Hz until 17 Hz and observe that the image processing task starts missing deadlines when activated at 17 Hz. With 128x96 input images, the `disparity` benchmark under contention can sustain a rate of 14 Hz in spite of heavy system load when isolated in a high-criticality container (Our-Stress case). Conversely, the sustainable rate decreases to 12 Hz when no isolation is enforced. In the Our-Solo case, `disparity` can run at a maximum frequency of 15 Hz, which is slightly lower than what can be achieved in the Lcy-Solo case (16 Hz). The drop arises from the fact that the SPM memory in PL is a bit slower than the PS DRAM [57]. We did not see the same behavior for an image resolution of 64x48 due to the cache. Importantly, however, the sustainable rate in the Our-Solo case is very close to the Our-Stress case. Thus, it can be concluded that our software/hardware co-design is able to deliver performance to highly critical applications that are close to the best-case. It is also important to highlight the low performance achieved by disparity for higher resolution images. We plan to investigate how to achieve better processing rates for image applications on top of the platform in future work.

Table 5. Supported frequencies for image size of 128x96.

Freq. (Hz)	Period (ms)	Lcy-Solo	Lcy-Stress	Our-Solo	Our-Stress
10	**100.00**	✓	✓	✓	✓
11	**90.91**	✓	✓	✓	✓
12	**83.33**	✓	✓	✓	✓
13	**76.92**	✓	✗	✓	✓
14	**71.43**	✓	✗	✓	✓
15	**66.67**	✓	✗	✓	✗
16	**62.50**	✓	✗	✗	✗
17	**58.82**	✗	✗	✗	✗

6.3 Schedulability Analysis Evaluation

In this subsection, we present an empirical evaluation using synthetic task sets of the *Lazy Load* and standard *Eager Load* three-phase task scheduling policies as well as tasks executing on the system without SPM that suffer main memory congestion or run with no memory interference.

The task set utilization U is varied from 0.05 to 1.00 in steps of 0.05. For each utilization value examined, 100000 task sets were generated. The default cardinality of the task set is $n = 8$. We used the *UUniFast* algorithm [4] to generate a set of n task utilization values $U_1, U_2, \ldots, U_n$, with total utilization of $\sum_{i=1}^{n} U_i = U$. For each task τ_i, its period T_i was drawn from a log-uniform distribution in the range of [100, 1000] ms and its worst-case execution time C_i was calculated as $U_i \cdot T_i$. The task load phase and unload phase transfer times are assumed to be equal and are drawn from a uniform distribution in the range of [40, 200] µs (according to Table 1, this is a sufficient time to transfer 32-160 KB). Tasks have implicit deadlines and priorities assigned by the *Rate-Monotonic* policy [26]. The experiments investigate the performance of the following scheduling policies:

(LL) Our proposed scheduling policy *Lazy Load* described in Section 3.4. We recall that the *Lazy Load* policy schedules the next load operation as late as possible.

(EL) The three-phase tasks SPM-oriented scheduling policy from [45, 46] where the DMA is reprogrammed at the task computation phase start, hereinafter called *Eager Load*. The analysis in [45, 46] supports multi-segment tasks, but it can be applied to single-segment tasks, like those considered in this work, without any loss of precision.

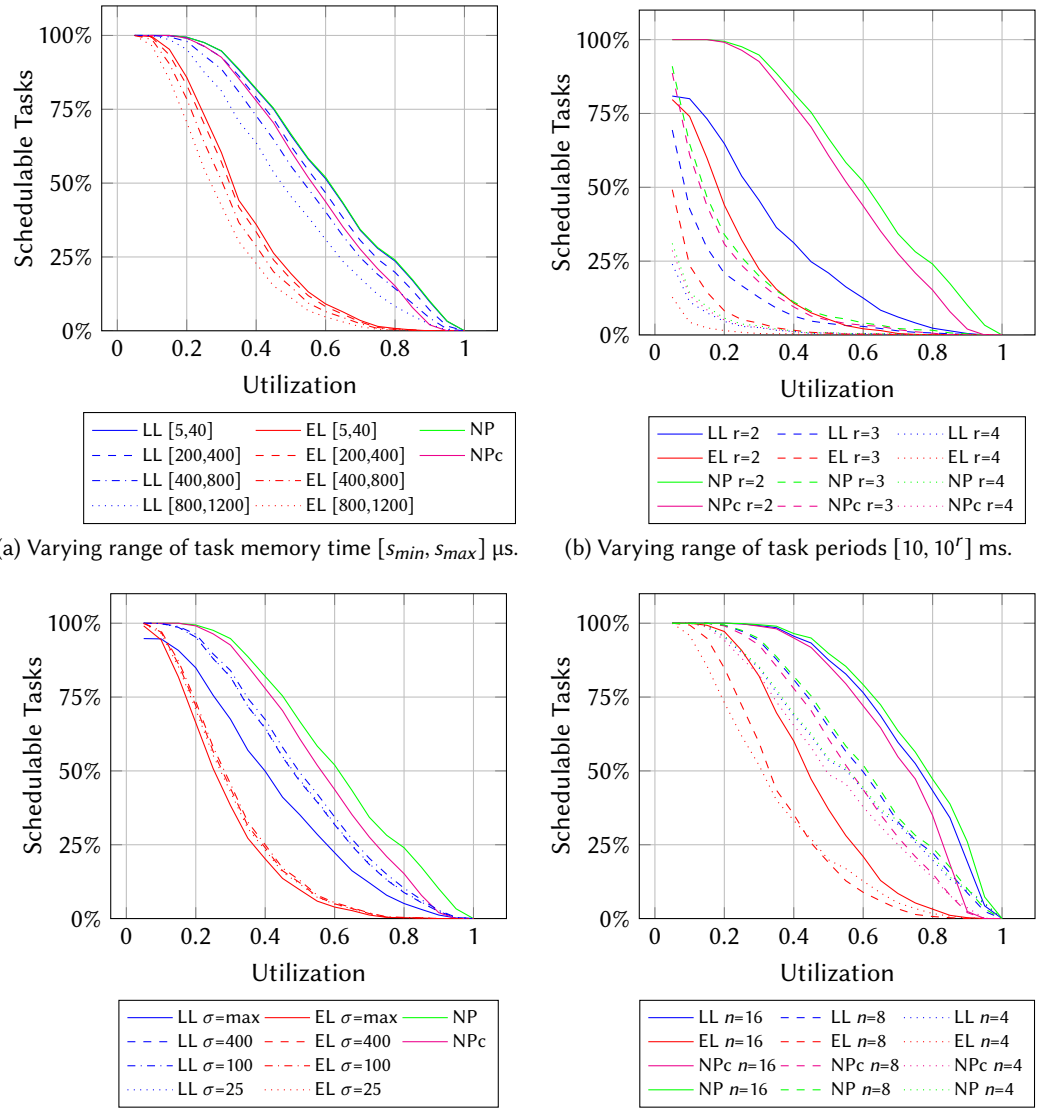
(NP) A standard fixed-priority non-preemptive scheduling policy assuming an ideal system, where tasks execute from the main memory without suffering any contention. A non-preemptive policy is used to avoid cache-related preemption delays. The response time analysis from [11] was applied to verify task set schedulability.

(NPc) As above, a standard fixed-priority non-preemptive scheduling policy but assuming a realistic multiprocessor system, where tasks suffer contention when accessing main memory. The contention-related overhead, with respect to the execution from SPM, is assumed to be 8% of the task worst-case execution time, as demonstrated in our previous case study in Section 6.2 (see WCET for Lcy-Stress and Our-Stress in Tables 2 and 3).

The first two policies (*LL* and *EL*) require task data to be transferred from main memory to SPM. We use a TDMA-based memory bus arbitration: the processor under study is assigned a unique time slot σ within which it is granted exclusive access to the memory. The TDMA round length is then set to TDMA fixed slot size multiplied by $M = 4$ (*i.e.,* the number of mid- and high-criticality processors available in the system). We consider four fixed slot lengths σ of 25 µs, 50 µs, 100 µs, 200 µs, and *max* where the slot length is set to the longest DMA transaction that the tasks can issue. If a DMA transaction cannot fit into a single TDMA slot, we split it into multiple smaller transactions. While doing so, we account for overhead to program the DMA. As shown in Section 6.1, this overhead in the ZCU102 platform is $3.98 < 4.00$ µs per slot (*e.g.,* if a transaction spans over ten slots, we add an overhead of 40.00 µs). Equation (1) is used to compute the total transfer time of load and unload phases. Unless stated otherwise, we run the simulation for all slot lengths σ and show the results giving the best schedulability performance.

The results of our schedulability study are shown in Figure 7, which includes four graphs with different parameters of the above experimental setup. For each scheduling policy, the percentage of generated task sets that were deemed schedulable is shown on y-axis, while the task sets utilization is shown on x-axis of the graphs. In what follows, we detail each set of experiments.

Varying task memory time. In the first experiment, we analyze the impact of the task memory transfer times on schedulability. We assume four ranges from which the task memory times are



(a) Varying range of task memory time $[s_{min}, s_{max}]$ µs.

(b) Varying range of task periods $[10, 10^r]$ ms.

(c) Varying TDMA slot size σ µs.

(d) Varying number of tasks n.

Fig. 7. Schedulability ratios for *Lazy Load* (*LL*), standard PREM *Eager Load* (*EL*), and fixed-priority non-preemptive policy with and without contention-related overhead (respectively *NPc* and *NP*).

drawn using a uniform distribution: [5, 40] µs, [200, 400] µs, [400, 800] µs, and [800, 1200] µs. The other parameters have their default values. The results are shown in Figure 7a. The *LL* performance for the shortest transfer times, [5, 40] µs, is close to the ideal *NP* scheduling. The DMA memory transfers can be easily overlapped with the task CPU computation, and the blocking factor they constitute is relatively small. However, increasing the transfer times results in a gradual schedulability decrease. For the transfer times longer than 400 µs, *LL* cannot bring any benefit compared to *NPc* where the tasks suffer main memory contention. The performance of the standard

three-phase task policy *EL* is always less than the *LL* and *NPc*. The *EL* policy can suffer blocking from up to two low-priority tasks [46] and the execution time reduction on the SPM assumed in this paper is not sufficient to compensate for it.

Varying task periods. In the second experiment shown in Figure 7b, we vary the range of task periods (*i.e.,* the ratio between the maximal and minimal possible task period) and show how it affects the task set schedulability. We consider three task periods ranges: $[10, 100]$ ms ($r = 2$), $[10, 1000]$ ms ($r = 3$), and $[10, 10000]$ ms ($r = 4$). The other parameters have their default values. The results of our evaluation are shown in Figure 7b. We observe that increasing the range of task periods degrades the schedulability test performance. This is explained by the fact that tasks with short deadlines cannot tolerate being blocked by tasks with large worst-case execution times (*e.g.,* due to the task generation technique, tasks with long periods are susceptible to have also long worst-case execution times). The gap between different policies is accordingly narrowing. The three-phase task scheduling policies induce worst-case inflation to account for overlapping of computation and memory phases (see Equation 2). This can degrade the schedulability when the worst-case execution time is relatively short. In that case, a hybrid approach can be applied: tasks with the worst-case execution times shorter than scratchpad reload time use main memory while other tasks with longer worst-case execution times scratchpad.

Varying TDMA slot size. In the third experiment shown in Figure 7c, we assign different TDMA slot durations and assess their impact on task set schedulability. Four TDMA slot durations σ are evaluated: 25 µs, 100 µs, 400 µs, and *max*. The transfer times are drawn from a uniform distribution in the wide range of $[5, 1200]$ µs. As shown in the first experiment, long transfer times can have a negative impact on the performance of *LL* and *EL* scheduling policies. However, such values allow testing TDMA slot assignment in scenarios where long transactions must be split, and the DMA must be reprogrammed multiple times. All the other parameters have their default values. The evaluation results are shown in Figure 7c. The schedulability improves for TDMA slots $\sigma \in \{25, 100, 400\}$ µs compared to the slot length set to the largest DMA transaction *max*. The latter approach results in time within a slot that might not be fully used and hence wasted. Recall that the memory-related delay in Equation (3) for blocking depends on L (the longest time of any task to load its code and data), which in turn depends on the TDMA slot and cycle length (see Equation (1), by assigning longer TDMA slots, we also increase the total length of the TDMA cycle). The performance with TDMA slots of 25 and 400 µs is similar (lines in Figure 7c are overlapping), and the best performance is achieved with the TDMA slot of 100 µs. However, a closer examination of the results revealed that among the TDMA slots $\sigma \in \{25, 100, 400\}$ µs, none is strictly dominant. We conjecture that the DMA reprogramming overhead (4 µs) has no detrimental effect on the TDMA performance, and splitting long transactions into multiple slots can improve task set schedulability.

Varying number of tasks. In our last experiment, we vary the task set cardinality n within a set $\{4, 8, 16\}$. The results are shown in Figure 7d. We observe that schedulability improves with increasing task set cardinality. Larger task sets equate to shorter worst-case execution times and, consequently, smaller blocking factors for non-preemptive scheduling.

In summary, the evaluations demonstrate that the *LL* policy implemented in the proposed system design achieves the schedulability performance close to the ideal *NP* scheduling for the tasks with transfer times below 40 µs and can mitigate the main memory congestion for the tasks with transfer times up to 400 µs. In all of the schedulability experiments, *LL* performs significantly better than the standard *EL* policy. Its effectiveness is due to the reduced low-priority task blocking (two low-priority tasks in *EL* and only one low-priority task in *LL*). Finally, breaking long memory transactions into multiple TDMA slots and thus keeping TDMA cycles short does not incur substantial overheads and improves task set schedulability.

7 CONCLUSION

This paper has explored the rich hardware features found in modern heterogeneous MPSoC architectures to define multiple criticality domains for real-time applications. We have used the PL to define dedicated PS-PL interfaces, scratchpad memories, and an address translator component to avoid the contention for the shared main memory by applications running on different cores and to provide a better utilization of the scratchpad when cache partitioning is applied. From the software side, we have used an RTOS and a hypervisor to provide isolation and cache partitioning for the criticality domains. We described our full-stack implementation of the proposed techniques and evaluated the system using realistic SD-VBS benchmarks.

We used a TDMA-scheduled DMA engine to support external I/O and data transfers to/from the mid-/high-criticality domains. We measured the DMA reprogramming overhead and showed that splitting long memory transactions into a small batch of separate transactions can significantly improve the system schedulability. The proposed *Lazy Load* scheduling policy for multi-phased tasks aims at reducing the low-priority tasks blocking. As demonstrated by our scheduling experiments, the *Lazy Load* significantly outperforms state-of-the-art scheduling policies for multi-phase tasks (even 50% improvement in the terms of system schedulability) and can ensure the temporal isolation of critical tasks.

8 ACKNOWLEDGMENTS

The material presented in this paper is based upon work supported by the National Science Foundation and ONR under grants numbers CNS 18-15891, CNS 19-32529, CCF-2008799 and N00014-17-1-2783. The work was also supported through the Red Hat Research program. Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. Giovani Gracioli was supported by Fundação de Desenvolvimento da Pesquisa - Fundep Rota 2030/Linha V 27192.02.01/2020.09-00. 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 sponsors.

REFERENCES

[1] Ahmed Alhammad and Rodolfo Pellizzoni. 2014. Time-Predictable Execution of Multithreaded Applications on Multicore Systems. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)* (Dresden, Germany). 1–6. https://doi.org/10.7873/DATE.2014.042

[2] Muhammad Ali Awan, Konstantinos Bletsas, Pedro F. Souto, Benny Akesson, and Eduardo Tovar. 2018. Mixed-Criticality Scheduling with Dynamic Memory Bandwidth Regulation. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 111–117. https://doi.org/10.1109/RTCSA.2018.00022

[3] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. 2016. Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. 14–24. https://doi.org/10.1109/ECRTS.2016.14

[4] Enrico Bini and Giorgio C. Buttazzo. 2005. Measuring the Performance of Schedulability Tests. *Real-Time Systems* 30, 1-2 (2005), 129–154. https://doi.org/10.1007/s11241-005-0507-9

[5] Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. 2012. Deterministic Execution Model on COTS Hardware. In *Architecture of Computing Systems – ARCS 2012*. Springer, 98–110. https://doi.org/10.1007/978-3-642-28293-5_9

[6] Reinder J. Bril, Johan J. Lukkien, and Wim F.J. Verhaegh. 2007. Worst-Case Response Time Analysis of Real-Time Tasks under Fixed-Priority Scheduling with Deferred Preemption Revisited. In *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*. 269–279. https://doi.org/10.1109/ECRTS.2007.38

[7] Paolo Burgio, Andrea Marongiu, Paolo Valente, and Marko Bertogna. 2015. A memory-centric approach to enable timing-predictability within embedded many-core accelerators. In *2015 CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*. 1–8. https://doi.org/10.1109/RTEST.2015.7369851

[8] Alan Burns and Robert I. Davis. 2017. A Survey of Research into Mixed Criticality Systems. *ACM Comput. Surv.* 50, 6, Article 82 (Nov. 2017), 37 pages. https://doi.org/10.1145/3131347

[9] Daniel Casini, Paolo Pazzaglia, Alessandro Biondi, Marco Di Natale, and Giorgio Buttazzo. 2020. Predictable Memory-CPU Co-Scheduling with Support for Latency-Sensitive Tasks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218640

[10] Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J. Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. 2020. Multi-Core Devices for Safety-Critical Systems: A Survey. *ACM Comput. Surv.* 53, 4, Article 79 (Aug. 2020), 38 pages. https://doi.org/10.1145/3398665

[11] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. 2007. Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised. *Real-Time Systems* 35, 3 (April 2007), 239–272. https://doi.org/10.1007/s11241-007-9012-7

[12] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. 2014. Predictable Flight Management System Implementation on a Multicore Processor. In *Embedded Real Time Software (ERTS'14)*. Toulouse, France. https://hal.archives-ouvertes.fr/hal-01121700

[13] Evidence. 2020. Erika Enterprise RTOS v3. http://www.erika-enterprise.com/ Online.

[14] Björn Forsberg, Luca Benini, and Andrea Marongiu. 2018. HePREM: Enabling predictable GPU execution on heterogeneous SoC. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 539–544. https://doi.org/10.23919/DATE.2018.8342066

[15] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. 2015. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *ACM Comput. Surv.* 48, 2, Article 32 (Nov. 2015), 36 pages. https://doi.org/10.1145/2830555

[16] Giovani Gracioli and Antônio Augusto Fröhlich. 2017. Two-phase colour-aware multicore real-time scheduler. *IET Computers & Digital Techniques* 11 (July 2017), 133–139(6). Issue 4. https://digital-library.theiet.org/content/journals/10.1049/iet-cdt.2016.0114

[17] Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. 2019. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 133)*, Sophie Quinton (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 27:1–27:25. https://doi.org/10.4230/LIPIcs.ECRTS.2019.27

[18] Heechul Yun. 2019. Latency and Bandwidth Utilities. https://github.com/heechul/misc

[19] Denis Hoornaert, Shahin Roozkhosh, and Renato Mancuso. 2021. A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 196)*, Björn B. Brandenburg (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:22. https://doi.org/10.4230/LIPIcs.ECRTS.2021.2

[20] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. 2014. Bounding memory interference delay in COTS-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 145–154. https://doi.org/10.1109/RTAS.2014.6925998

[21] Hyoseung Kim, Arvind Kandhalu, and Ragunathan Rajkumar. 2013. A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems. In *2013 25th Euromicro Conference on Real-Time Systems*. 80–89. https://doi.org/10.1109/ECRTS.2013.19

[22] Hyoseung Kim and Ragunathan (Raj) Rajkumar. 2017. Predictable Shared Cache Management for Multi-Core Real-Time Virtualization. *ACM Trans. Embed. Comput. Syst.* 17, 1, Article 22 (Dec. 2017), 27 pages. https://doi.org/10.1145/3092946

[23] Namhoon Kim, Bryan C. Ward, Micaiah Chisholm, Cheng-Yang Fu, James H. Anderson, and F. Donelson Smith. 2016. Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 1–12. https://doi.org/10.1109/RTAS.2016.7461323

[24] Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko Bertogna. 2019. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 1–14. https://doi.org/10.1109/RTAS.2019.00009

[25] John P. Lehoczky. 1990. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *[1990] Proceedings 11th Real-Time Systems Symposium*. 201–209. https://doi.org/10.1109/REAL.1990.128748

[26] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (Jan. 1973), 46–61. https://doi.org/10.1145/321738.321743

[27] Méndez Miguel Macías, José L. Gutierrez, David Fernández, and Javier Díaz. 2013. Open platform for mixed-criticality applications. In *Proceedings of the Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems (WICERT 2013)*. 1–7. http://atcproyectos.ugr.es/wicert/downloads/wicert_papers/wicert2013_submission_8.pdf

[28] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. 2013. Real-Time Cache Management Framework for Multi-core Architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 45–54. https://doi.org/10.1109/RTAS.2013.6531078

[29] Joel Matějka, Björn Forsberg, Michal Sojka, Luca Benini, Zdeněk Hanzálek, and Andrea Marongiu. 2019. Combining PREM Compilation and Static Scheduling for High-Performance and Predictable MPSoC Execution. *Parallel Comput.* 85 (12 2019), 27–44. https://doi.org/10.1016/J.PARCO.2018.11.002

[30] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. 2015. Memory-Processor Co-Scheduling in Fixed Priority Systems. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems* (Lille, France) *(RTNS '15)*. Association for Computing Machinery, New York, NY, USA, 87–96. https://doi.org/10.1145/2834848.2834854

[31] Paolo Modica, Alessandro Biondi, Giorgio Buttazzo, and Anup Patel. 2018. Supporting Temporal and Spatial Isolation in a Hypervisor for ARM Multicore Platforms. In *2018 IEEE International Conference on Industrial Technology (ICIT)*. 1651–1657. https://doi.org/10.1109/ICIT.2018.8352429

[32] Tiago Mück, Antonio A. Fröhlich, Giovani Gracioli, Amir M. Rahmani, João Gabriel Reis, and Nikil Dutt. 2018. CHIPS-AHOy: A Predictable Holistic Cyber-Physical Hypervisor for MPSoCs. In *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation* (Pythagorion, Greece) *(SAMOS '18)*. Association for Computing Machinery, New York, NY, USA, 73–80. https://doi.org/10.1145/3229631.3229642

[33] Anup Patel, Mai Daftedar, Mohamed Shalan, and M. Watheq El-Kharashi. 2015. Embedded Hypervisor Xvisor: A Comparative Analysis. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 682–691. https://doi.org/10.1109/PDP.2015.108

[34] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. 2011. A Predictable Execution Model for COTS-Based Embedded Systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. 269–279. https://doi.org/10.1109/RTAS.2011.33

[35] Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. 2017. Look Mum, no VM Exits! (Almost). In *Proceedings of the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '17)*. http://arxiv.org/abs/1705.06932

[36] Juan M. Rivas, Joël Goossens, Xavier Poczekajlo, and Antonio Paolillo. 2019. Implementation of Memory Centric Scheduling for COTS Multi-Core Real-Time Systems. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 133)*, Sophie Quinton (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 7:1–7:23. https://doi.org/10.4230/LIPIcs.ECRTS.2019.7

[37] Shahin Roozkhosh, Denis Hoornaert, and Renato Mancuso. 2022. CAESAR: Coherence-Aided Elective and Seamless Alternative Routing via on-chip FPGA. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. 356–369. https://doi.org/10.1109/RTSS55097.2022.00038

[38] Shahin Roozkhosh, Denis Hoornaert, Ju Hyoung Mun, Tarikul Islam Papon, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. 2023. Relational Memory: Native In-Memory Accesses on Rows and Columns. In *2023 International Conference on Extending Database Technology (EDBT)*. Ioannina, Greece. https://doi.org/10.48786/edbt.2023.06

[39] Shahin Roozkhosh and Renato Mancuso. 2020. The Potential of Programmable Logic in the Middle: Cache Bleaching. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 296–309. https://doi.org/10.1109/RTAS48715.2020.00006

[40] Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. 2017. Tightening Contention Delays While Scheduling Parallel Applications on Multi-Core Architectures. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 164 (Sept. 2017), 20 pages. https://doi.org/10.1145/3126496

[41] Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, and Isabelle Puaut. 2019. Hiding Communication Delays in Contention-Free Execution for SPM-Based Multi-Core Architectures. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 133)*, Sophie Quinton (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 25:1–25:24. https://doi.org/10.4230/LIPIcs.ECRTS.2019.25

[42] Gero Schwäricke, Tomasz Kloda, Giovani Gracioli, Marko Bertogna, and Marco Caccamo. 2020. Fixed-Priority Memory-Centric Scheduler for COTS-Based Multiprocessors. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 165)*, Marcus Völp (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1:1–1:24. https://doi.org/10.4230/LIPIcs.ECRTS.2020.1

[43] Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. 2020. E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management. In *2020 IEEE Real-Time Systems Symposium (RTSS)*. 345–357. https://doi.org/10.1109/RTSS49844.2020.00039

[44] Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. 2022. Profile-Driven Memory Bandwidth Management for Accelerators and CPUs in QoS-Enabled Platforms. *Real-Time Syst.* 58, 3 (Sep 2022), 235–274. https://doi.org/10.1007/s11241-022-09382-x

[45] Muhammad R. Soliman, Giovani Gracioli, Rohan Tabish, Rodolfo Pellizzoni, and Marco Caccamo. 2019. Segment Streaming for the Three-Phase Execution Model: Design and Implementation. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. 260–273. https://doi.org/10.1109/RTSS46320.2019.00032

[46] Muhammad R. Soliman and Rodolfo Pellizzoni. 2019. PREM-Based Optimal Task Segmentation Under Fixed Priority Scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 133)*, Sophie Quinton (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 4:1–4:23. https://doi.org/10.4230/LIPIcs.ECRTS.2019.4

[47] The KITTI Vision Benchmark Suite. 2020. KITTI. http://www.cvlibs.net/datasets/kitti/ Online.

[48] Rohan Tabish, Renato Mancuso, Saud Wasly, Ahmed Alhammad, Sujit S Phatak, Rodolfo Pellizzoni, and Marco Caccamo. 2016. A real-time scratchpad-centric os for multi-core embedded systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 1–11. https://doi.org/10.1109/RTAS.2016.7461321

[49] Rohan Tabish, Renato Mancuso, Saud Wasly, Rodolfo Pellizzoni, and Marco Caccamo. 2019. A real-time scratchpad-centric OS with predictable inter/intra-core communication for multi-core embedded systems. *Real-Time Systems* 55, 4 (2019), 850–888. https://doi.org/10.1007/s11241-019-09340-0

[50] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. 2016. Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 1–12. https://doi.org/10.1109/RTAS.2016.7461361

[51] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. 2009. SD-VBS: The San Diego Vision Benchmark Suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 55–64. https://doi.org/10.1109/IISWC.2009.5306794

[52] Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. 2013. Outstanding Paper Award: Making Shared Caches More Predictable on Multicore Platforms. In *2013 25th Euromicro Conference on Real-Time Systems*. 157–167. https://doi.org/10.1109/ECRTS.2013.26

[53] Saud Wasly and Rodolfo Pellizzoni. 2014. Hiding Memory Latency Using Fixed Priority Scheduling. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 75–86. https://doi.org/10.1109/RTAS.2014.6925992

[54] Jack Whitham and Neil C. Audsley. 2012. Explicit Reservation of Local Memory in a Predictable, Preemptive Multitasking Real-Time System. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*. 3–12. https://doi.org/10.1109/RTAS.2012.19

[55] Jack Whitham, Neil C. Audsley, and Robert I. Davis. 2014. Explicit Reservation of Cache Memory in a Predictable, Preemptive Multitasking Real-Time System. *ACM Trans. Embed. Comput. Syst.* 13, 4s, Article 120 (apr 2014), 25 pages. https://doi.org/10.1145/2523070

[56] Jack Whitham, Robert I. Davis, Neil C. Audsley, Sebastian Altmeyer, and Claire Maiza. 2012. Investigation of Scratchpad Memory for Preemptive Multitasking. In *2012 IEEE 33rd Real-Time Systems Symposium*. 3–13. https://doi.org/10.1109/RTSS.2012.54

[57] Xilinx. 2019. Zynq UltraScale+ Device - Technical Reference Manual. https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf

[58] Meng Xu, Linh Thi, Xuan Phan, Hyon-Young Choi, and Insup Lee. 2017. vCAT: Dynamic Cache Management Using CAT Virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 211–222. https://doi.org/10.1109/RTAS.2017.15

[59] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. 2012. Memory-Centric Scheduling for Multicore Hard Real-Time Systems. *Real-Time Systems* 48, 6 (Nov. 2012), 681–715. https://doi.org/10.1007/s11241-012-9158-9

[60] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Heechul Yun, and Marco Caccamo. 2016. Global Real-Time Memory-Centric Scheduling for Multicore Systems. *IEEE Trans. Comput.* 65, 9 (2016), 2739–2751. https://doi.org/10.1109/TC.2015.2500572

[61] Ying Ye, Richard West, Jingyi Zhang, and Zhuoqun Cheng. 2016. MARACAS: A Real-Time Multicore VCPU Scheduling Framework. In *2016 IEEE Real-Time Systems Symposium (RTSS)*. 179–190. https://doi.org/10.1109/RTSS.2016.026

[62] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. 2014. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 155–166. https://doi.org/10.1109/RTAS.2014.6925999

[63] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 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)*. 55–64. https://doi.org/10.1109/RTAS.2013.6531079