Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms

Giovani Gracioli
Technical University of Munich, Germany
Federal University of Santa Catarina, Brazil
g.gracioli@tum.de

Rohan Tabish
University of Illinois at Urbana-Champaign, USA
rtabish@illinois.edu

Renato Mancuso
Boston University, USA
rmancuso@bu.edu

Reza Mirosanlou
University of Waterloo, Canada
rmirosan@uwaterloo.ca

Rodolfo Pellizzoni
University of Waterloo, Canada
rpellizz@uwaterloo.ca

Marco Caccamo
Technical University of Munich, Germany
mcaccamo@tum.de

Abstract

Multiprocessor Systems-on-Chip (MPSoC) integrating hard processing cores with programmable logic (PL) are becoming increasingly common. While these platforms have been originally designed for high performance computing applications, their rich feature set can be exploited to efficiently implement mixed criticality domains serving both critical hard real-time tasks, as well as soft real-time tasks.

In this paper, we take a deep look at commercially available heterogeneous MPSoCs that incorporate PL and a multicore processor. We show how one can tailor these processors to support a mixed criticality system, where cores are strictly isolated to avoid contention on shared resources such as Last-Level Cache (LLC) and main memory. In order to avoid conflicts in last-level cache, we propose the use of cache coloring, implemented in the Jailhouse hypervisor. In addition, we employ ScratchPad Memory (SPM) inside the PL to support a multi-phase execution model for real-time tasks that avoids conflicts in shared memory. We provide a full-stack, working implementation on a latest-generation MPSoC platform, and show results based on both a set of data intensive tasks, as well as a case study based on an image processing benchmark application.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded systems; Computer systems organization → Other architectures

Keywords and phrases Mixed-criticality systems, SoC Heterogeneous platforms, FPGA, real-time computing

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.24

1 Introduction

Recently there has been an uptrend in the demand for high-performance real-time applications. The increasing interest in emerging technologies like self-driving cars, drones, cube satellites, and smart manufacturing, to name a few, has determined a shift in the type of workload that has to be considered "real-time" [5]. Traditional CPU-intensive tasks comprise a small percentage of the real-time workload in modern high-criticality systems,

while increasingly more memory- and I/O-intensive applications have been brought into the picture. Additionally, hardware manufacturers have anticipated the demand for high-performance embedded systems by introducing increasingly more feature-rich multiprocessor systems-on-chip (MPSoC) platforms.

In the race to provide the future de-facto standard for pervasive high-performance embedded systems, hardware manufacturers have experimentally introduced a plethora of architectural features. A number of these features have a proven track record in the general-purpose computing domain and multiple indicators suggest their long-term adoption in the embedded market [10]. Such features include 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.

The presence of on-chip "soft" PL, tightly coupled with a group of "hard" embedded CPUs, represents a game-changer for systems that need to be tailored to a well-known application scenario [21]. This is indeed often the case for real-time systems. In fact, this new class of platforms offers the unprecedented ability to define new hardware components to complement the high-performance profile of the embedded cores. If it is possible to devise PL-defined components that mitigate the non-determinism in high-performance CPUs; the result can be an ideal trade-off between processing power and real-time guarantees [21].

In this paper, we study how it is possible to leverage latest-generation partially reconfigurable embedded platforms for a system design that combines high-performance and strict real-time requirements. In our approach, we define multiple *criticality domains* to be intended as subsystems of the computing system. Each criticality domain may be designed with a different trade-off between high-performance and strict temporal determinism. For instance, a high-performance domain may run a general-purpose OS with a complex I/O infrastructure. Conversely, a high-criticality domain is comprised of a Real-Time Operating System (RTOS) supporting time-sensitive applications.

We demonstrate that it is possible to instantiate a critical core 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. In summary, this paper makes the following contributions:

1. We demonstrate that it is possible to leverage partially reconfigurable embedded platforms to instantiate a system where high-performance and time-sensitive applications co-exist under strict temporal isolation. Compared to the ideal case (*i.e.,* task running alone in the system), our set of hardware/software techniques ensures that execution time of a time-sensitive task does not suffer from the potentially large interference caused by memory-intensive tasks running on different cores (only 6% of an increase in the execution time, instead of a large interference).
2. We design and implement a hardware block, named address translator, that prevents the problem of memory waste when cache partitioning based on page coloring is used.
3. We provide a working implementation on one of the latest-generation partially reconfigurable embedded MPSoCs. Our implementation is full-stack, with adaptations at an OS- and application-level, extensions to a partitioning hypervisor, and generation of PL-defined hardware blocks. We demonstrate the feasibility of the implementation by using a case study on image processing and show the hard real-time bounds achieved by our system design.

Organization of the paper. The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents the adopted system model and assumptions. Section 4 discusses the design principles and overviews the proposed approaches. Section 5 presents a design space exploration of a modern MPSoC platform through a set of experiments. It also shows the proposed hardware and software design to support mixed-criticality

applications on top of the platform. Section 6 details the system implementation with multiple criticality domains. Section 7 presents some experimental results carried out to evaluate the proposed real-time computing framework. Finally, Section 8 states the conclusions and outlines some future work.

2 Related Work

Several recent works have proposed techniques to deal with shared resources in multicore real-time systems at both OS and hypervisor levels. Mancuso *et al.* profiled the source code to extract memory access patterns for each task, allowing frequently-used pages to be locked in cache in order to avoid cache evictions [19]. Combined with cache partitioning based on page coloring, their approach significantly improves predictability. Following the same line, some works used coloring to partition the cache in multicore real-time systems [15, 13, 11]. Other works focused on making DRAM accesses more predictable [38, 14, 17]. MemGuard regulates each core's memory request rate by using hardware performance counters to account for the memory access usage [39]. The work defines a threshold and when the number of memory accesses reaches the threshold, an overflow interrupt is generated to keep the specified memory bandwidth. One assumption for MemGuard is that all cores have access to the same memory bus, while in our work we explore the existence of the programmable logic to define dedicated memory interfaces. We also show how page coloring can co-exist with the programmable logic memories and how to prevent wasting cache space due to page coloring.

The use of hypervisors in multicore real-time systems is a recent trend. Modica et al. proposed a hypervisor-based architecture targeting critical systems similar to ours [22]. Cache partitioning provided spatial isolation, while a DRAM bandwidth reservation mechanism provided temporal isolation. Both cache partitioning and memory reservation mechanisms were implemented in the XVISOR open-source hypervisor [24] and tested in a quad-core ARM A7 processor. Our proposed hypervisor-based approach, instead, uses an MPSoC platform, which gives us the ability to explore other features, such as specific FPGA direct memory access (DMA) blocks (for instance, to handle data transfers between the processing system and programmable logic sides) and data prefetching.

MARACAS addressed shared cache and memory bus contention through multicore scheduling and load-balancing on top of the Quest OS [37]. MARACAS used hardware performance counters information to throttle the execution of threads when memory contention exceeded a certain threshold. The counters were also used to derive an average memory request latency to reduce bus contention. vCAT used the Intel's Cache Allocation Technology (CAT) to achieve core-level cache partitioning for the hypervisor and virtual machines running on top of it [36]. vCAT was implemented in Xen and $LITMUS^{RT}$. Although interesting, this approach is architecture dependent and uses non-real-time basic software support (Linux and Xen).

Kim and Rajkumar proposed a predictable shared cache framework for multicore real-time virtualization systems [16]. The proposed framework introduced two hypervisor techniques (vLLC and vColoring) that enabled cache-aware memory allocation for individual tasks running in a virtual machine. CHIPS-AHOy is a predictable holistic hypervisor [23]. It integrates shared hardware isolation mechanisms, such as memory partitioning, with an observe-decide-adapt loop to achieve predictability and energy and thermal management.

Crespo *et al.* used hardware performance counters within the hypervisor to regulate the memory bandwidth of critical and non-critical cores [6]. The work used control theory to do the regulation and presented a set of experiments to tune the controller parameters. Awan *et al.* proposed a memory regulation mechanism for mixed-criticality applications [2]. Mendez *et al.* also proposed to use FPGA together with a processing system to reduce interference of mixed-criticality applications. However, in their system model, the authors did not consider multicore processors or shared caches [21].

SPM-centric OS combined scratchpad, resource specialization, scheduling of shared resources as well as a three-phase model to achieve predictability in multicore real-time systems [28]. The three-phase model is also used in this work. It consists of a load phase,

in which code/data is loaded from main memory to the scratchpad (SPM), an execution phase, and an unload phase in which code/data is unloaded from the SPM to main memory. The model relies on a DMA engine to support the load/unload phases. The idea is to load data/code for a task using a DMA before it starts and to unload it after completion, as depicted in Figure 1. Because the SPM is divided into two halves, while one task is executing in one half, DMA is active on the another one. Up arrows in the figure represent the release times of three tasks. While for simplicity we draw the figure assuming that all load and unload phases take an equal amount of time to complete, in reality, their length can vary on a per-task basis. Note that each job starts executing on the core after it is loaded in the scratchpad and the previous job finishes executing, whichever happens last. Also note that while load phases have higher priority over unloads, at time $t = 3$ (and $t = 5$) an unload must be performed first in order to free Partition A for the successive load phase of task τ_3. If there are multiple ready tasks, the decision of which task to schedule is made when starting a load phase; hence, while τ_1 has a higher priority than τ_3, the latter is executed at time 5 because τ_1 is released right after the start of another load phase at $t = 4$. This behavior causes blocking time on the higher priority task and must be accounted for in a schedulability analysis.

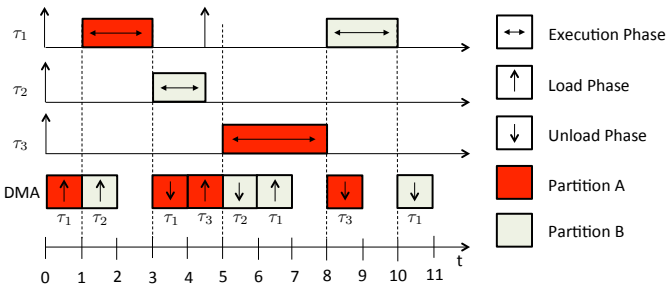

Figure 1 Example of a schedule using the three-phase model.

The work in [28] used a time division multiple access (TDMA) arbitration among cores based on a fixed slot size for DMA transfers. Only a single DMA operation (either a task load or unload) was carried out during a slot. The TDMA slot size length was designed so that it was possible to load or unload an entire scratchpad partition within one slot. Hence, it always allowed to transfer an amount of data equal to the largest scratchpad size, which is undesirable if the SPM size is different per-core. On the contrary, in this work the DMA scheduling employs variable memory phase sizes, similarly to what has been described in [33]. Notice, though, that the work in [33] mainly targets single-core processors, and it does not provide a working implementation for multi-core systems.

3 System Model and Assumptions

In this section we summarize the system model and assumptions of the proposed architecture.

3.1 Criticality Domains

Our goal is to implement multiple *criticality domains* on a single multicore SoC. Given C, the total number of cores in the SoC, we target a system with up to C criticality domains, so that each criticality domain is statically assigned to at least one core. One of the key design principles is that criticality domains are isolated from each other, both in time and space [5]. In other words, we minimize the impact that the activity of applications in one criticality domain can have on tasks in a different criticality domain.

Domain Types: Albeit strong isolation exists between criticality domains, each domain may have different requirements in terms of performance, amount of memory resources, and runtime environment. We envision three types of criticality domains. First, a *low-criticality*

domain is used to perform I/O with complex devices, processing of large amounts of data, using general-purpose libraries and applications. A low-criticality domain may run a generic operating system (OS) – e.g. Linux – and require a large amount of memory with fast-on-average performance. While applications in this domain are shielded from interference from the rest of the system, no strong temporal guarantees can be expressed due to the best-effort nature of the software stack. Second, a *high-criticality domain* consolidates all the hard real-time tasks of the system and interfaces with simple I/O devices. In this domain, applications have strong timing guarantees. Finally, a third *mid-criticality domain* is used to process 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 $M \leq C$.

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 exemplified by Figure 2.

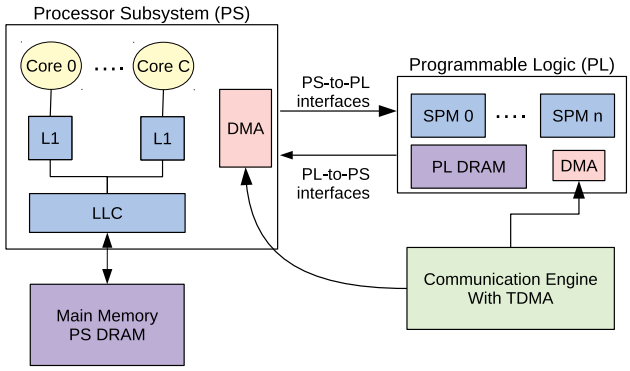

Figure 2 Overview of the platform with the main components.

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). While other organizations for the memory hierarchy are possible, we adopt a widespread model in modern multicore embedded systems. A key difference in the considered class of partially reconfigurable systems is the following. A miss in LLC causes a memory transaction to be performed towards either the main memory (PS DRAM) or the Programmable Logic (see Figure 2). This behavior depends on the exact physical memory address being accessed. 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, as demonstrated in Figure 2. Such a feature for the emerging class of partially reconfigurable embedded systems is of crucial importance, and manufacturers [35] are well aware of it. 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 communication engine capable of accessing and transferring memory from/to PL and PS memories is available. Usually, a Direct Memory Access (DMA) component is available in either the PS or the PL and it can act as the communication engine. Its main role is to provide means for the load and unload phases of the three-phase task model. Differently from the previously implemented three-phase solution in [28], 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 $\Sigma = \sum_{j=1}^{M} \sigma_j$ 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 unload/load phases (refer to Figure 1) require k TDMA slots. Then it is easy to see that the total transfer time Δ is upper bounded by:

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

the core receives one slot every Σ 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

Because multiple criticality domains exist in the system, we make different assumptions on applications in different domains. 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.

Conversely, we assume that mid- and high-criticality applications adhere to more conservative assumptions. 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 non-preemptive rate-monotonic scheduling (RM). 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 assume that the memory footprint of mid-/high-criticality tasks is limited. On 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 into high overheads. Tasks follow the three-phase model, as discussed in Section 2. Since we employ similar scheduling rules with variable time memory phases, we argue that the analysis in [33] can be adapted to provided scheduling guarantees for our proposed system, after using Equation 1 to bound the length of memory phases. Due to space limitations, we defer a complete schedulability evaluation to future work, while in this paper we focus on the hardware and software design of the computing platform.

4 Design Principles and Approach Overview

Our design revolves around the idea of *freedom from interference* among criticality domains [5]. The ideal software stack and assignment of resources to domains is depicted in Figure 3. We hereby provide a short description of the main challenges and techniques used to achieve a close approximation of what is depicted in Figure 3 by using a commercially available MPSoC embedded platform. We describe additional important implementation details in Section 6.

Inter-domain Interference: Temporal interference between criticality domains should be limited. More specifically, it is fundamental that any interference from a lower-criticality

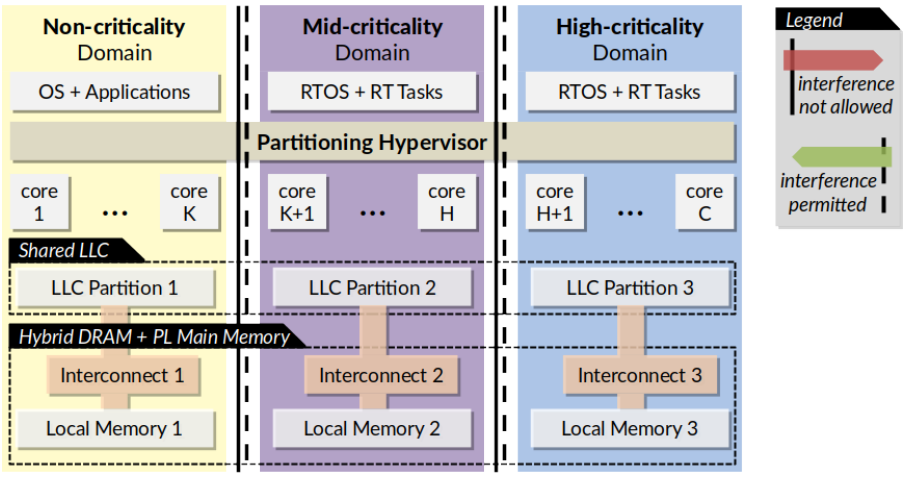

Figure 3 Ideal software and hardware stack organization.

domain towards a higher-criticality domain is prevented —solid vertical lines in Figure 3. It is desirable that higher-criticality applications do not interfere with lower-criticality domains. But some degree of interference is acceptable in this case —dashed vertical lines in Figure 3. The paradigm follows traditional safety-critical systems certification guidelines [21]. High-criticality applications need to be certified regardless of the behavior of lower-criticality workload. Conversely, some degree of knowledge of higher-criticality applications can be assumed when certifying lower-criticality applications.

Partitioning Hypervisor: Applications in different domains operate in self-contained address spaces, with inter-domain communication channels handled at the hypervisor level. Hardware resources (*e.g.*, cores, cache partitions, main memory storage, I/O devices) are statically assigned to criticality domains. As such, we employ a thin partitioning hypervisor which does not perform any online scheduling. 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 memory coloring; (4) performing tasks' relocation to/from DRAM into local memories; and (5) providing message-passing channels for inter-domain communication.

LLC Partitioning: We rely on LLC partitioning based on page coloring[1] [10]. Hypervisor-level coloring has been proposed in [4, 16, 18]. An extensive discussion of the subtle practical challenges to enforce coloring at the hypervisor level is provided in [18]. In this work, we use the same hypervisor as in [18].

Preventing Memory Waste: A well known drawback of cache partitioning via coloring is memory waste. Coloring enforces a restriction on the physical addresses, and hence actual memory, that can be assigned to applications. For instance, if one wants to assign one-fourth of a shared LLC to a guest OS, then one-fourth of available main memory cannot be assigned to any other OS. This represents a significant drawback. The problem is even more severe when local memories like scratchpads are used. In fact, the size of scratchpads is typically very limited —a few hundreds of kilobytes to a few megabytes. Enforcing coloring essentially cripples the ability of applications to access the majority of an already limited memory resource. In this work, we leverage the Programmable Logic (PL) and propose a technique to prevent coloring-induced memory waste. Specifically, we introduce a *bus translator* that acts on transactions forwarded to local memories. In short, the component redirects colored —and hence scattered— memory accesses to contiguous memory locations.

Main Memory Partitioning: Partitioning main memory among guest OSes is necessary due to the problem of shared memory contention. Allowing the access to the same memory bank from cores allocated to different criticality domains would violate the requirement

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

of enforcing strict isolation. Additionally, multiple domains can saturate the shared bus and/or memory controller, experiencing significant contention and delays. Both problems are well known. Solutions based on coloring have been proposed for the former [38, 18, 14]. Software [20, 1] and hardware [40] solutions based on bandwidth regulation have been explored to address the latter. In this work, we propose and explore an alternative approach to both issues. Our approach is made possible by the capability of defining new hardware components in the Programmable Logic (PL). First, we instantiate dual-ported memories that are only accessible by a single criticality domain. Next, we dedicate a PL-PS interface to criticality domains, and on each PL-PS interface we instantiate two memory controllers inside the PL. The first controller is used for memory accesses generated by applications running on the processor. Whereas, the second controller is used for memory transactions originated by the communication engine.

Handling Tasks' Relocation: As described in Section 3, tasks' code and data are moved to/from local memories defined in the PL by the communication engine. To implement task relocation (for loading/unloading), a possible approach consists in compiling applications using position-independent code (PIC) [28]. However, compilation as PIC results in less optimized binaries [28]. Additionally, migrating a running task to a different memory region is challenging [2]. In this work, we propose to compile tasks against absolute intermediate physical addresses (IPA). Then, after 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.

5 Design Space Exploration for Mixed-Criticality in a Modern MPSoC

In this section, we first describe the architectural overview of the considered platform. We then describe the experimental setup and different scenarios that were evaluated to justify our final design.

5.1 Architectural Overview of the Chosen Platform

For our implementation, we have used the Xilinx UltraScale+ ZCU102 MPSoC [34]. 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. This is in line with many high-performance embedded multicore processors. The PS includes a DDR4-2666 (main memory) controller with a data bus width of 64-bit, which on our reference board is connected to a 4GB DDR4 memory module. The PL also includes a separate, 16-bit synthesized memory controller, which on our board is wired to a 512 MB DDR4 memory module.

Multiple interfaces between the PL and the processor subsystem (PS) exist. There are three interfaces going from the PS [3] 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.

[2] This is because registers and stack in a saved context may contain absolute addresses.

[3] 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.

5.2 Experiments

When exploring the characteristics of modern MPSoC platforms, it is easy to realize that there are many possible designs one can create to achieve predictability for mixed criticality domains. For the Xilinx ZCU102, for instance, the communication between the PL and the PS can have one or two high performance master ports (HPM0 and HPM1). Tasks running on the application cores (A53) can use the PS or PL DRAMs or even access the block RAM (BRAM) in the FPGA. We have designed a set of experiments to evaluate the behavior of different configurations under stress. Based on the related work [31, 28, 20], we chose two memory-intensive applications (disparity and mser) from the San Diego Visual Benchmark Suite (SD-VBS) [32] to be used in the evaluations. We chose the SD-VBS benchmark suite, because it provides vision applications similar to those used in autonomous cars. Thus, they represent real-time applications that demand both predictability and performance. We then ported `disparity` and `mser` to Erika RTOS/Jailhouse (we describe Erika and Jailhouse later in Section 6) and executed them with SQCIF (128×96) input data size. To stress the memory subsystem, we used a bandwidth benchmark (BW) from [12]. This benchmark is tailored to issue writes to the main memory (DRAM) or SPM (i.e., block RAM in the PL) by ensuring that every write is a miss in the LLC.

Using the memory intensive and bandwidth benchmarks, we evaluate the scenarios described in Table 1. We consider two legacy (Lcy) scenarios, and three scenarios in which our solution is used (Our). In the first legacy scenario (Lcy-Solo), the benchmark under analysis (`disparity` or `mser`) runs solo from the PS DRAM without cache coloring on top of Linux (kernel 4.14). Note that it does not use any high performance master (HPM) port, because it does not access the PL. In the second legacy scenario (Lcy-Stress), contention is added. Specifically, three bandwidth benchmark instances access the PS DRAM also from different cores in Linux. This scenario represents the simplest possible design in the platform since no special technique is used to avoid contention. Next, we consider our solution. In scenario Our-Solo, the benchmark under analysis runs alone in the system using a dedicated HPM port and accessing an SPM in FPGA. In this and the following cases, the cache has also been partitioned via coloring. In scenario Our-Mid, the benchmark under analysis runs from the mid-criticality domain and three contending bandwidth benchmark instances are added. The first runs in the low-criticality domain (Linux), the second in the high-criticality domain, and the third in the mid-criticality domain. The latter shares an HPM port with the benchmark under analysis. Finally, in scenario Our-High, the benchmark under analysis executes from a high-criticality domain, using a dedicated HPM port. Two contending bandwidth benchmark instances run from a mid-criticality domain and share a single HPM port, while an additional contending bandwidth benchmark instance runs in the low-criticality domain.

Table 1 Summary of the five scenarios considered for the evaluation.

Scenario	Experiment	Accessed Memory	Coloring	HPM Port	Contention Type
Lcy-Solo	Solo	PS DRAM	No	Not used	None
Lcy-Stress	Contention	PS DRAM	No	Not used	3× BW
Our-Solo	Solo	SPM	Yes	Dedicated	None
Our-Mid	Contention	SPM	Yes	Shared	1× BW from low-crit. 1× BW from mid-crit. 1× BW from high-crit.
Our-High	Contention	SPM	Yes	Dedicated	1× BW from low-crit. 2× BW from mid-crit.

Figure 4 and Figure 5 depict the results for the `mser` and `disparity` applications, respectively, under the five aforementioned scenarios. Each experiment reports the result of

1000 executions. In both figures, the x-axis reports the execution time in clock cycles[4]. On the left y-axis we present the experimentally derived execution time distribution, while on the right y-axis we show the cumulative distribution function (CDF). The vertical dashed line shows the average, while the vertical dotted line corresponds to the observed WCET. The annotation in each plot provides the numerical values for average, WCET, best-case execution time (BCET), and variability window — which is a metric of predictability and is computed as $(WCET - BCET)/WCET$.

A few important trends can be highlighted in the results for `mser` (Figure 4). First, for the two legacy scenarios, in Lcy-Stress (Figure 4b) the application exhibits a drastic 1.73× increase in WCET compared to Lcy-Solo (Figure 4a) due to added contention. Moreover, the execution time in the Lcy-Stress case becomes unstable, with a variability window of 27.8%. Next, when executing in a mid-criticality (or high-criticality) domain without contention (Our-Solo case — Figure 4c), the performance of the application under analysis is comparable to the Lcy-Solo case. If the application is deployed in a mid-criticality domain (Our-Mid case — Figure 4d), a sharp improvement in predictability and WCET is observed compared to the Lcy-Stress case. In fact, the variability window is reduced by 42% and the WCET is reduced by 31%. Finally, in Figure 4e, the application is run inside a high-criticality domain, and hence with a dedicated HPM port — Our-High case. By considering the Lcy-Stress case as the baseline, we observe a 58% reduction in variability window, as well as a 37% reduction in WCET. Additionally, note that in the Our-High case the application performance is remarkably close to what is observed in the Our-Solo case.

The results for `disparity` reported in Figure 5 follow similar trends. First, the WCET shows a 1.27× increase between Lcy-Solo and Lcy-Stress, reported in Figure 5a and Figure 5b respectively. When the benchmark is executed alone in the system in a mid-criticality (or high-criticality) domain (case Our-Solo in Figure 5c), its WCET and average execution time increases only slightly by 1.04× and 1.07× respectively. Intuitively, this is because the SPM is a slower memory compared to the PS DRAM. Next, consider the Our-Mid (Figure 5d) case where `disparity` is executed in a mid-criticality domain with contention from the rest of the system. Compared to the Lcy-Stress case, we observe a 8% reduction in WCET and a 32% decrease in variability. When the application runs in a high-criticality domain (case Our-High in Figure 5e), its WCET is minimally affected by contending workload, with a 1.06× increase compared to the Our-Solo case. Notably, the variability window in the Our-High case is lower than in the Lcy-Solo case.

Based on the evaluated scenarios, we can conclude that the hardware isolation provided by a dedicated memory interface, a dedicated SPM memory, together with cache coloring (Our-High case), is able to deliver better predictability and performance close to the ideal case (Our-Solo). We present and discuss the final hardware design in the next section.

6 Support for Mixed-Criticality Applications on MPSoCs

In this section we present a general overview of the implementation carried out in the ZCU102 platform to provide predictability for mixed-criticality applications. We start giving an overview of the implementation in Section 6.1, then we present the details for each implemented component (hypervisor, cache coloring, address translator, RTOS, and code/data relocation).

6.1 Overview of Implementation

Based on the experiments described in the previous section, our final hardware design is depicted in Figure 6. 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 System (RTOS). A few noticeable

[4] 1 clock cycle is equal to 0.01 us.

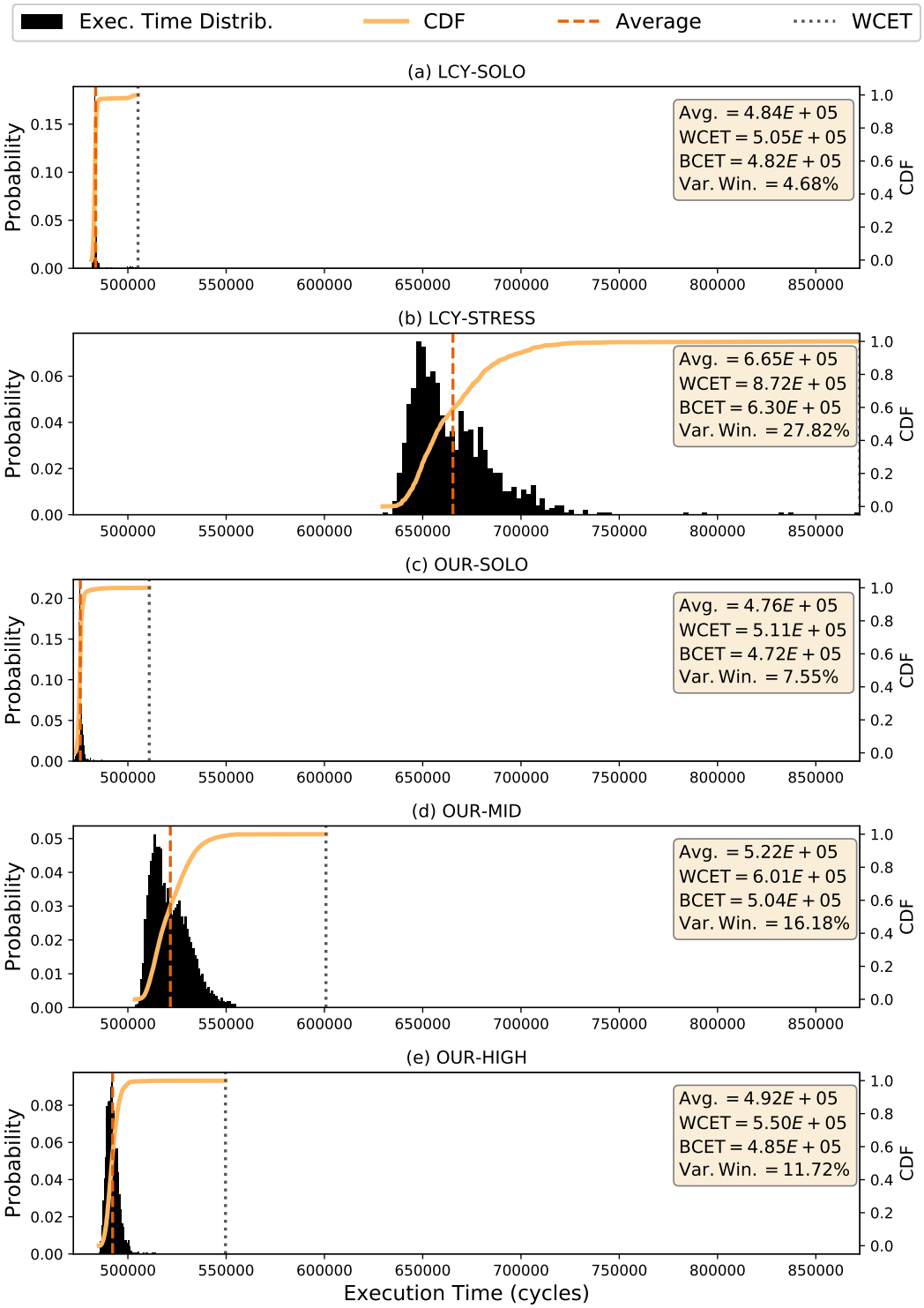

Figure 4 Results for `mser` application. See the summary of the scenarios in Table 1.

features of our proposed design are: (i) the low-criticality domain is assigned direct access to DRAM because this domain features applications with sizable footprints; (ii) each mid- and high-criticality domain is assigned a private 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

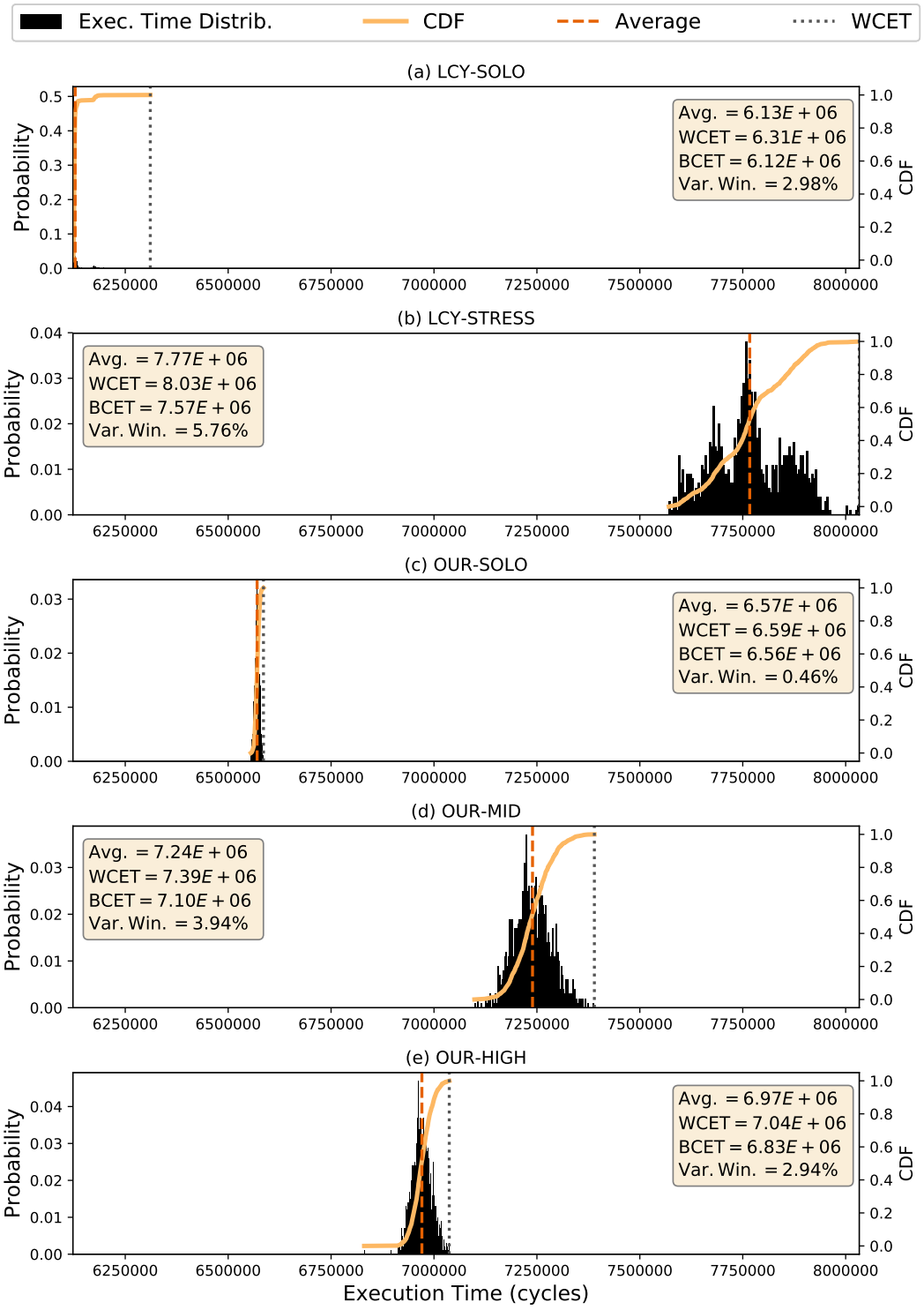

Figure 5 Results for `disparity` application. See the summary of the scenarios in Table 1.

interface to access its private SPM. It should be noted that the SPM (BRAM) memories in the Xilinx FPGA are dual-ported and thus there is no extra overhead (which may not be the case when using dual-ported memories in Application Specific Integrated Circuits). Moreover, the size of each SPM can be defined according to the applications and RTOS requirements for each criticality domain. Since in our platform the maximum size of all

SPMs is 3 MB, the size of the SPM used by the high-criticality domain was set to 2 MB, while the size of the other two SPMs used by mid-criticality domains was set to 512 KB each.

The low-criticality core is also responsible for booting a hypervisor (Jailhouse). Jailhouse allows us to partition shared memory resources, especially the LLC and DRAM by implementing cache coloring. We have two partitions in the DRAM; one dedicated to run Linux and another one to place the code/data of the tasks running on the A53 application cores (to support the three-phase model as will be discussed below).

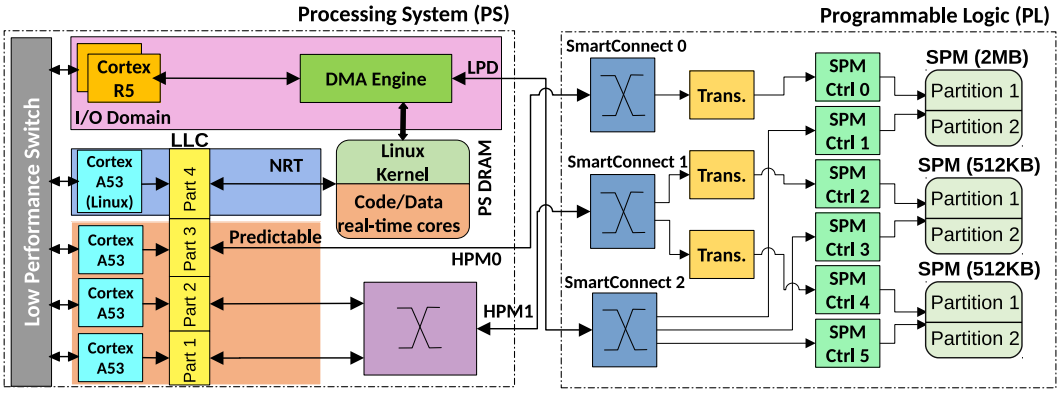

Figure 6 PS-PL interface and design

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 interfaces between PL and PS available in the platform and three A53 application 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 6).

Although there is another interface between PS and PL called low performance domain (LPD) that can be used for the third A53 core, we opt not to use it. We have used a latency benchmark [12] to measure the performance when one single core is accessing the LPD interface and when two cores access the same HPM interface under stress. The obtained latency for the HPM interface under stress was 202 ns, while for the LPD was 220 ns. Thus, the LPD interface is used to carry DMA transfers to/from the SPM/DRAM on the behalf of the A53 application cores, as part of the TDMA-based scheduling. The TDMA-based scheduling of the DMA is handled by the R5 core. To pipeline the execution of a currently running task with the load of the next task, we divide the SPM into two halves. A dual ported SPM was used so that a DMA and an application core can both write/read to/from SPM at the same time.

In order to avoid the contention between A53 cores in different criticality domains, we partition the LLC via coloring. Coloring is used since no hardware support is available to partition the LLC. 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 detailed discussion on each of the main components including Jailhouse, page coloring, address translator, how the A53 cores in different criticality domains communicate using the hypervisor, RTOS support for the system model, and task relocation to support the three-phase model.

6.2 Jailhouse to Partition the Shared Resources

As the hypervisor we use Jailhouse. Jailhouse is a partitioning hypervisor which can be used to transform a symmetric multiprocessing (SMP) system into an asymmetric multiprocessing (AMP) system [25]. Jailhouse is bootstrapped via a Linux driver and favors simplicity and low-overhead over sophisticated (para-)virtualized techniques, which is ideal for real-time systems [25]. It requires 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. Then, to create additional domains (called non-root cells), Jailhouse removes hardware resources assigned to Linux (such as a processor core or a specific I/O device) and reassigns them to the new cell [25]. The idea is to have non-critical tasks running on the Linux cell and critical tasks running on isolated partitions on top of an RTOS.

The A53 cores support a two-stage virtual memory translation. User-space applications in a guest-OS, such as Linux or an RTOS, are assigned virtual addresses (VA). The first stage of translation uses page tables maintained by the guest OS to translate VAs into intermediate physical addresses (IPA). The second stage of translation is in control of the hypervisor, and it is used to translate IPAs to physical addresses (PAs) via a second set of page tables.

The RTOS used for mid-/high-criticality domains is Erika Enterprise version 3, which is open-source and OSEK/VDX certified [9, 7]. Erika supports fixed-priority scheduling and a porting for Xilinx Ultrascale+ platform is available.

6.3 Page Coloring

To enforce strong inter-domain (inter-cell) and hence inter-core performance isolation, we leverage page coloring (see [18, 10] for a complete overview of the technique). We use the virtualization extensions of the processor to implement coloring by enforcing appropriate restrictions on the color of pages that Jailhouse maps to 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. On the one hand, it allows us to localize the changes required to implement coloring-based partitioning in a single software component (Jailhouse). On the other hand, it allows deploying unmodified and possibly closed-source OS inside our criticality domains. A similar technique was used in [22, 16, 18].

6.4 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, 22 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 22 bits of an address, the translator IP receives 24 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 4 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 2 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 4 consecutive pages.

Let us assume that the address of the translator in Figure 6 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 7 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.

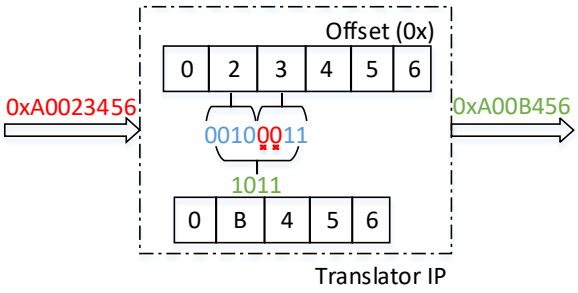

Figure 7 Translator IP operation. The two most significant bits from the fourth byte (in red) of the input address are dropped.

In our design (Figure 6), 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.

6.5 Communication among Jailhouse Cells

Usually, communication among processors is done by sharing memory buffers and Inter-Processor Interrupts (IPIs). Jailhouse allows to issue direct IPIs only among processors that are assigned to the same cell, and hence the same criticality level. To avoid a low-criticality domain (*i.e.,* Linux) to interfere with higher-criticality domains (by sending an unlimited number of IPIs), currently we do not allow issuing IPIs among processors with different criticality levels. In the future, the communication among tasks from different criticality levels will be performed through the Jailhouse shared memory mechanism, based on the creation of virtual PCI devices and their legacy interrupts in the static configuration for each cell [25]. This means that the system designer could specify which criticality level (*i.e.,* Jailhouse cell) can communicate with other criticality levels. As future work, we will investigate how to enhance Jailhouse with a server-based scheduling mechanism for IPIs [8], such that real-time guarantees can be preserved in the event that a low-criticality domain tries to send an unlimited number of IPIs to a higher-criticality one. Also, we propose the use of FIFO buffers implemented in the shared memory (which can also be colored) to support the communication among OSes.

6.6 Erika RTOS Running on Real-Time Cores

All the A53 application cores run a partitioned fixed priority scheduler, provided by the Erika RTOS, and always execute from dedicated SPM memory assigned to them. Both the dedicated SPMs and the PS DRAM assigned to Linux are colored to avoid cache evictions in the shared cache. Erika does not support virtualization on ARMv8 CPUs, as such it would not use virtual addresses (VAs). By default, however, Jailhouse performs the setup of a flat 1:1 stage-one (virtual address to intermediate physical address – VA→IPA) addressing space before booting any non-root cell. This is required to support cacheable memory. An

application in the Erika RTOS is always statically compiled against VA/IPA addresses. As shown in Figure 8, the task running on the Erika core can be in one of the following states:

- **Running:** The tasks is executing from SPM.
- **Ready Loaded:** The task is loaded and is ready to execute from SPM.
- **Ready Unloaded:** The task is released but it is not yet loaded to SPM.
- **Completed**: A task has completed.
- **Waiting on Event (Unloaded)**: The task is waiting on a timer or on an event.

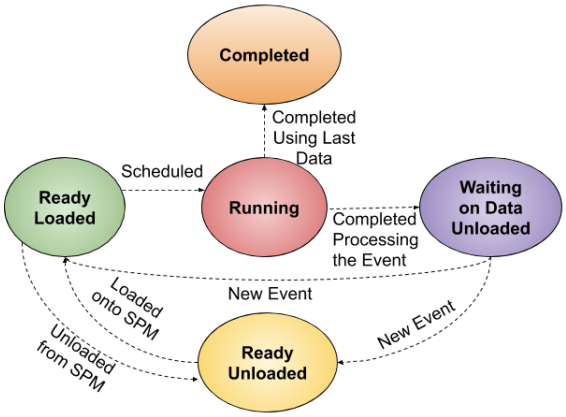

Figure 8 Overview of the different states of a task in Erika RTOS.

Note that the transition from Waiting on Event Unloaded to Ready Loaded is performed when there is no other ready task and the waiting task that was unloaded receives all the events it needs. The state transition also encompasses the loading phase. To allow the load and unload of code and data of Erika's tasks, we use the support for virtual memory implemented in Jailhouse, described in the next subsection.

6.7 Code/Data Relocation

Relocation is the process of assigning addresses to position-independent code and data. 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.

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. Hypercalls in Jailhouse are services provided by the hypervisor to its cells. A Jailhouse hypercall receives three arguments; the hypercall code or ID and two arguments that are specific to the hypercall. We added to Jailhouse two new hypercall IDs, indicating either load or unload operations. The second argument is used to encode (i) the source/destination address in DRAM (page-aligned, least-significant 12 bits are zero); and (ii) the offset in pages from the beginning of the SPM where the task needs to be loaded to/unloaded from (the largest SPM is 2 MB, so the maximum offset is 512 - 1 pages, and it takes the 9 least-significant bits). The third argument encodes the size of the task that needs to be loaded/unloaded. As shown in [3], the overhead of a hypercall in Jailhouse on the ZCU102 platform is around 400 ns.

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 the calling environment (Erika RTOS). The effective copy of the task into/from SPM is performed by the DMA engine.

7 Evaluation

In this section, we present the evaluation of our system design. 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 demonstrate the limits in terms of hard real-time guarantees of our system through a case study on image processing.

7.1 DMA Evaluation

In order to move data between the PS DRAM and the SPM memory inside the PL, we use the PS-side DMA. Because only one DMA is used to move data on behalf of three A53 cores, we propose a fine granularity TDMA-based scheduling of the DMA. When activated, the DMA transfers data between PS DRAM and SPMs using the low-power domain (LPD) interface. In our design, a dedicated LPD port is used instead of a shared HPM port to avoid contending with the application cores when performing DMA transfers. The TDMA schedule is handled by one of the ARM Cortex-R5 cores. For this purpose, a bare-metal firmware was deployed on the R5, created using the Xilinx SDK 2018/02. The SDK uses the `armr5-none-eabi-gcc` compiler. The following compilation flags were used: `-DARMR5 -W -Wall -O0 -g3`.

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 2 shows the obtained results. Recall that 1 MB represents half the size of the largest SPM in our design. The results show that the standard deviation remains within the range [0.057, 0.1]. It can also be noted that the achievable bandwidth increases proportionally to the amount of contiguous memory transferred, peaking at around 870 MB/s with transfers of 1 MB in size.

We also measured the DMA programming overhead (*i.e.,* the time to program and start a DMA transfer). The worst-case DMA programming overhead obtained from all the experiments 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 plan to fully analyze the impact of the DMA programming overhead into the schedulability of real-time tasks in future work. However, based on insights provided by previous work on the three-phase model [28, 33], 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 transfer size of 32 KB for each core, then based on Equation 1 we obtain $\sigma_j = 38.81 + 3.89 = 42.7$ us, $\Sigma = 3 \cdot 42.7 = 128.1$ us, 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 $\Delta = 2092.3$ us, meaning that tasks should execute for at least 2 ms to hide the memory time.

7.2 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 [32]. `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 [32]. The objective of this evaluation is to demonstrate how

Table 2 DMA transfer time (in us) 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

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 to be 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 [27]. In particular, we used 20 frames from 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 load/unload phase of the task using the same approach as described in [29]. Table 2 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 [26].

We considered four out of the five scenarios described in Table 1. 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 (see Section 5) 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 3, and a resolution of 128x96 in Table 4. 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 [30]. The results for 128x96 (SQCIF size) input images were presented in Section 5 and correspond to the cases analyzed in Figure 5(a), 5(b), 5(c), and 5(e).

Table 3 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-HIGH
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 4 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-HIGH
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 5 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). Whereas a cross mark indicates that the desired rate was not sustainable. From the results in Table 5, we can see that by running `disparity` without any interference, the maximum sustainable rate is 62 Hz. However, when running under contention and 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 6 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 4, 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-HIGH case). Conversely, the sustainable rate decreases to 12 Hz when no isolation is enforced. In the OUR-SOLO case, `disparity` can run at a

Table 5 Supported frequencies for image size of 64x48.

Freq. (Hz)	Period (ms)	LCY-SOLO	LCY-STRESS	OUR-SOLO	OUR-HIGH
55	**18.18**	✓	✓	✓	✓
56	**17.86**	✓	✗	✓	✓
57	**17.54**	✓	✗	✓	✓
58	**17.24**	✓	✗	✓	✓
59	**16.95**	✓	✗	✓	✓
60	**16.67**	✓	✗	✓	✗
62	**16.13**	✓	✗	✓	✗
63	**15.87**	✗	✗	✗	✗

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 [34]. 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 6 Supported frequencies for image size of 128x96.

Freq. (Hz)	Period (ms)	LCY-SOLO	LCY-STRESS	OUR-SOLO	OUR-HIGH
10	**100.00**	✓	✓	✓	✓
11	**90.91**	✓	✓	✓	✓
12	**83.33**	✓	✓	✓	✓
13	**76.92**	✓	✗	✓	✓
14	**71.43**	✓	✗	✓	✓
15	**66.67**	✓	✗	✓	✗
16	**62.50**	✓	✗	✗	✗
17	**58.82**	✗	✗	✗	✗

8 Conclusion and Future Work

In this paper, we have shown how one can define multiple criticality domains by exploiting the rich hardware features provided by a modern heterogeneous SoC that incorporates multiple CPUs and PL. Following the proposed design, the PL is used to define dedicated portions of scratchpad memory for mid-/high-criticality applications. Additionally, we ensure that no contention exists for high-criticality applications by routing their memory transactions using a dedicated high-performance memory interface inside the PL. Similarly, mid-criticality applications access their SPM using a memory interface shared only with other mid-criticality applications. External I/O and communication data from the rest of the system is transferred to the mid-/high-criticality domains by a TDMA-scheduled DMA engine using a real-time R5 core. We described our full-stack implementation of the proposed techniques and evaluated the system using realistic SD-VBS benchmarks.

As a part of future work, we plan to investigate how to enhance Jailhouse with a server-based scheduling mechanism for limiting the number of IPIs between processors from different criticality levels, how to integrate compiler code generation to automatically generate load/unload requests for tasks following the three-phase model, to discuss the schedulability analysis of the proposed fine-granularity DMA transfer based on TDMA, how to guarantee that the communication engine executes safely, since if it fails (due to a security attack for instance) the entire system also fails, and how to achieve higher computational power for image processing applications without giving up on the relatively strong isolation achieved by the current design.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedback, and our shepherd for helping to significantly improve this paper. This work has been supported in part by the NSF under grant number CNS-1646383, by the NSERC and by CMC Microsystems. Marco Caccamo was also supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. 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. A. Agrawal, R. Mancuso, R. Pellizzoni, and G. Fohler. Analysis of dynamic memory bandwidth regulation in multi-core real-time systems. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 230–241, Dec 2018. doi:10.1109/RTSS.2018.00040.
2. Ali Awan, Konstantinos Bletsas, Pedro F. Souto, Benny Akesson, and Eduardo Tovar. Mixed-criticality scheduling with dynamic memory bandwidth regulation. In *IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 111–117, 08 2018.
3. Maxim Baryshnikov. FPGA-based support for predictable execution model in multi-core CPU. Master's thesis, Czech Technical University in Prague, Prague, Czech Republic, 5 2018.
4. Alessandro Biondi, Mauro Marinoni, Giorgio Buttazzo, Claudio Scordino, and Paolo Gai. Challenges in virtualizing safety-critical cyber-physical systems. In *Proceedings of Embedded World Conference 2018*, pages 1–5, Feb 2018.
5. Alan Burns and Robert I. Davis. A survey of research into mixed criticality systems. *ACM Comput. Surv.*, 50(6):82:1–82:37, November 2017. doi:10.1145/3131347.
6. A. Crespo, P. Balbastre, J. Simó, J. Coronel, D. Gracia Pérez, and P. Bonnot. Hypervisor-based multicore feedback control of mixed-criticality systems. *IEEE Access*, 6:50627–50640, 2018.
7. Evidence. Erika enterprise RTOS v3, Oct 2018. Online; accessed 16 October 2018. URL: http://www.erika-enterprise.com/.
8. T. Facchinetti, G. Buttazzo, M. Marinoni, and G. Guidi. Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 98–105, July 2005. doi:10.1109/ECRTS.2005.21.
9. Paolo Gai, Enrico Bini, Giuseppe Lipari, Marco Di Natale, and Luca Abeni. Architecture for a portable open source real time kernel environment. In *In Proceedings of the Second Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, 2000.
10. G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 48(2), November 2015.
11. G. Gracioli and A. A. Fröhlich. Two-phase colour-aware multicore real-time scheduler. *IET Computers Digital Techniques*, 11(4):133–139, 2017.
12. Heechul Yun. Latency and Bandwidth Utilities. https://github.com/heechul/misc, Feburary 2019.
13. C. Kenna, J. Herman, B. Ward, and J. H. Anderson. Making shared caches more predictable on multicore platforms. In *ECRTS '13*, pages 157–167, 2013.
14. H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, April 2014.
15. Hyoseung Kim, Arvind Kandhalu, and Ragunathan Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *Proc. of the ECRTS 2013*, pages 80–89, 2013.
16. Hyoseung Kim and Ragunathan (Raj) Rajkumar. Predictable shared cache management for multi-core real-time virtualization. *ACM Trans. Embed. Comput. Syst.*, 17(1):22:1–22:27, December 2017.
17. N. Kim, B. C. Ward, M. Chisholm, C. Fu, J. H. Anderson, and F. D. Smith. 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)*, pages 1–12, April 2016.

18 T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Montreal, Canada, April 2019.

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

20 R. Mancuso, R. Pellizzoni, M. Caccamo, Lui Sha, and Heechul Yun. WCET(m) estimation in multi-core systems using single core equivalence. In *2015 27th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 174–183, July 2015.

21 M. Mendez, J.L.G. Rivas, D.F. Garca-Valdecasas, and J. Diaz. Open platform for mixed-criticality applications. In *Proc. of the Conference on Design, Automation and Test in Europe, WICERT (DATE)*, pages 1–7, 2013.

22 P. Modica, A. Biondi, G. Buttazzo, and A. Patel. Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms. In *Proceedings of the IEEE International Conference on Industrial Technology (ICIT 2018)*, pages 1–7, Feb 2018.

23 T. Mück, A. A. Fröhlich, G. Gracioli, A. Rahmani, and N. Dutt. Chips-ahoy: A predictable holistic cyber-physical hypervisor for mpsocs. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 1–8, Samos Island, Greece, 2018.

24 A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi. Embedded hypervisor xvisor: A comparative analysis. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 682–691, March 2015.

25 R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer. Look mum, no VM exits! (almost). In *Proc. of the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2017)*, pages 13–18, 2017.

26 M. R. Soliman and R. Pellizzoni. PREM-based optimal task segmentation under fixed priority scheduling. In *2019 31st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 1–24, July 2019.

27 The KITTI Vision Benchmark Suite. Kitti, Oct 2019. Online; accessed 20 April 2019. URL: `http://www.cvlibs.net/datasets/kitti/`.

28 R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A real-time scratchpad-centric os for multi-core embedded systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, April 2016.

29 R. Tabish, R. Mancuso, S. Wasly, R. Pellizzoni, and M. Caccamo. A real-time scratchpad-centric OS with predictable inter/intra-core communication for multi-core embedded systems. *Real Time Systems*, 2019.

30 P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016. `doi:10.1109/RTAS.2016.7461361`.

31 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*, pages 1–12. IEEE, 2016.

32 S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, Oct 2009.

33 S. Wasly and R. Pellizzoni. Hiding memory latency using fixed priority scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 75–86. IEEE, 2014.

34 Xilinx. Zynq UltraScale+ Device - technical reference manual. URL: `https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf`.

35 Xilinx. Versal: The First Adaptive Compute Acceleration Platform (ACAP). `https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf`, 2018. [Online; accessed 16-January-2019].

36 M. Xu, L. Thi, X. Phan, H. Y. Choi, and I. Lee. vCAT: Dynamic cache management using CAT virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 211–222, April 2017.

37 Y. Ye, R. West, J. Zhang, and Z. Cheng. MARACAS: A real-time multicore VCPU scheduling framework. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 179–190, Nov 2016.

38 H. Yun, R. Mancuso, Z.P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 155–166. IEEE, 2014.

39 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. 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)*, pages 55–64. IEEE, 2013.

40 Y. Zhou and D. Wentzlaff. Mitts: Memory inter-arrival time traffic shaping. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 532–544, June 2016. doi:10.1109/ISCA.2016.53.