Tech Report: A Virtualized Scratchpad-Based Architecture for Real-time Event-Triggered Applications

Giovani Gracioli, Rohan Tabish**, Reza Mirosanlou*, Renato Mancuso***, Rodolfo Pellizzoni*, and Marco Caccamo
Technical University of Munich, Germany, {g.gracioli, mcaccamo}@tum.de
*University of Waterloo, Canada, {rmirosanlou, rpellizz}@uwaterloo.ca
**University of Illinois at Urbana-Champaign, USA, {rtabish, ayooshb2, mcaccamo}@illinois.edu
***Boston University, USA, {rmancuso}@bu.edu

***Abstract*—Systems-on-a-Chip (SoC) devices integrating hard processing cores with programmable logic (PL) are becoming increasingly available. 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 COTS-based heterogeneous SoCs that incorporates PL and a multicore processor. We show how one can tailor these processors to support a mixed/full 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 LLC, 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 SoC platform, and show results based on both a set of data intensive tasks, as well as a complete case study based on an anomaly detection application for an autonomous vehicle.

I. 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, drone racing, edge cloud, 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". Traditional CPU-intensive workload comprises 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. On the other hand, hardware manufacturers have anticipated the demand for high-performance embedded systems by introducing increasingly more feature-rich systems-on-chip (SoC).

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 there are few indications about their long-term adoption in the embedded market. 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. This is indeed often the case for real-time systems. In fact, this new class of platforms offer 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.

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 sub-shells 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) to 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-existence under strict temporal isolation.
2) We derive schedulability results for real-time applications that execute according to a multi-phase model while exchanging communication data across multiple criticality domains;
3) We show how the flexibility offered by the PL in the

definition of memory storage, controllers, and interconnects, allows preventing unwanted temporal interference by design;
4) We provide a working implementation on one of the latest-generations of partially reconfigurable embedded SoCs. 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.

II. Related Work

Several works have been proposing techniques to deal with shared resources in multicore real-time systems at both OS and hypervisor levels recently. Mancuso *et al.* profile the source code to extract memory accesses patterns for each task [1]. Then, pages that are the most frequently used can be locked in cache (to avoid cache evictions). Also, cache partitioning based on page coloring helps to improve the predictability. Following the same line, some works use coloring to partition the cache in multicore real-time systems [2, 3, 4]. Other works focused on making the DRAM accesses more predictable [5, 6, 7].

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 our architecture [8]. Cache partitioning is used to provide spatial isolation, while a DRAM bandwidth reservation mechanism provides temporal isolation. Both cache partitioning and memory reservation mechanisms were implemented in the XVISOR open-source hypervisor [9] and tested in a quad-core ARM A7 processor. Our proposed hypervisor-based approach, instead, uses a MPSoC platform, which gives the possibility to explore other features, such as specific FPGA DMA blocks (to handle data transfer between PS and PL sides for instance) and data prefetching.

MARACAS addresses shared cache and memory bus contention through multicore scheduling and load-balancing on top of the Quest OS [10]. MARACAS uses hardware performance counters information to throttle the execution of threads when memory contention exceeds a certain threshold. The counters are also used to derive an average memory request latency to reduce bus contention. vCAT uses the Intel's Cache Allocation Technology (CAT) to achieve core-level cache partitioning for the hypervisor and virtual machines running on top of it [11]. 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 [12]. The proposed framework introduces two hypervisor techniques (vLLC and vColoring) that enables cache-aware memory allocation for individual tasks running running in a virtual machine. CHIPS-AHOy is a predictable holistic hypervisor [13]. It integrates shared hardware isolation mechanism, such as memory partitioning, with an observe-decide-adapt loop to achieve predictability and energy, thermal, and wearout management.

Crespo *et al.* use hardware performance counters within the hypervisor to regulate the memory bandwidth and also to act on non real-time cores [14]. The work uses control theory to do the regulation and presents a set of experiments to tune the controller parameters.

SPM-centric OS combines scratchpad, resource specialization, scheduling of shared resources as well as a three-phase model to achieve predictability in multicore real-time systems [15]. 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 work also proposes to split SPM in two halves, but it uses fixed TDMA slot sizes, while our DMA scheduling employs variable memory phases sizes, similarly to [16]. However, in [16] the authors target single-core real-time systems, and thus it does not schedule DMA requests among cores. Furthermore, the proposed work ignores output (response time is based on execution finish rather than unload).

III. System Model and Assumptions

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

A. Criticality Domains

Our goal is to implement multiple *criticality domains* on the same 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 with respect to each other, both in time and space. In other words, we minimize the impact that the activity of applications in a 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. In light of this, we envision three types of criticality domains. First, a *low-critical domain* is used to perform I/O with complex devices, processing of large amounts of data, using general-purpose libraries and applications. A low-critical 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-critical 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 and tight bounds on their worst-case execution times (WCET). Finally, a third *mid-critical domain* is used to process tasks with intermediate criticality. Within this domain, and unlike the low-critical domain, temporal guarantees on performances are still provided. Their tightness, however, is lower compared to the high-critical domain. The number of cores allocated to high- and mid-criticality domains is $M \leq C$.

B. Application Cores

We assume that there are high-performance application cores in the system. These cores are connected to the main memory using high performance interfaces. Each application core has its own local cache and all the application cores have a shared last-level cache (LLC). We also assume that there is a mechanism, either in hardware or software, to partition the LLC among the cores.

C. Tightly-coupled Programmable Logic

We assume that there exists a sizable block of programmable logic (PL) that is tightly-coupled with the embedded processing cores. To be considered "tightly-coupled", the PL area should provide high-bandwidth, low-latency communication interfaces to (as a master) and from (as a slave) the processing subsystems (PS) which includes the main processing cores. The number and capacity, in terms of memory throughput, of the PL-PS interfaces directly impacts the performance and degree of temporal isolation that can be enforced among criticality domains.

The presence of a tightly-coupled PL shall also allow directly accessing I/O peripherals via dedicated high-bandwidth interfaces. This capability directly impacts the possibility to define in-PL high-speed DMA engines, in case they are not included in the PS sub-shell.

D. Application Model

Real-time tasks are scheduled based on a partitioned, fixed-priority approach. We let $\Gamma = \{\tau_1, \ldots, \tau_N\}$ denote the set of real-time tasks assigned to a given core under analysis, ordered by decreasing and distinct priorities. As discussed in Section II, tasks follow a three-phase model. Hence, for a task τ_i, we use $\tau_i.ld$ to denote the time required to load into the scratchpad its code, private and input data using DMA, while $\tau_i.ul$ is the time required to unload from the scratchpad modified and output data. Finally, we use $\tau_i.c^{\max}$ to denote the worst-case execution time of τ_i, $\tau_i.c^{\min}$ to denote its best-case execution time, and $\tau_i.D$ for its relative deadline.

We support both time-triggered and sporadic, event-triggered task activation; consistent with the OSEK standard [17], time-triggered tasks are simply associated with a periodic timer event. Hence, we use $\tau_i.\alpha(t)$ to denote the maximum number of activation events for τ_i that can arrive in any interval of time of length t; this is also the maximum number of jobs of τ_i that can be released in the interval. Note that for a periodic or sporadic task with period $\tau_i.T$, we simply have $\tau_i.\alpha(t) = \lceil t/\tau_i.T \rceil$; furthermore, in this case we allow $\tau_i.D$ to be greater than $\tau_i.T$ (arbitrary deadlines). Since in our model a task outputs its results during the unload phase, the response time of a job is defined as the time between its release and the completion of its unload phase. The analysis in Section V computes upper and lower bounds $R_i^{\max}$, $R_i^{\min}$ on the response time of any job of a task τ_i.

Real-time tasks are allowed to communicate among each other and with applications in the non real-time partition through buffers and IPIs, as detailed in Section VI. Consider two communicating real-time tasks τ_i and τ_j allocated to either the same or different cores, such that τ_j is executed every time τ_i produces data for it. Then $\tau_j.\alpha(t)$ depends on the number of jobs of τ_i that complete in any interval of length t. Note that our activation model is consistent with the concept of upper arrival curves / event functions in real-time calculus [18] and compositional performance analysis [19]; hence, the corresponding theories can be used to derive arrival patterns for event-triggered tasks and derive end-to-end delay and buffer size bounds, as long as we can analyze the response time for each individual task. In particular, a bound on the number of job completions for τ_i can simply be obtained as $\tau_i.\alpha(t + R_i^{\max} - R_i^{\min})$.

IV. Proposed Architecture and Scheduling Model

We schedule three-phase tasks on each real-time core by pipelining execution phases and load/unload phases, using a similar approach to [16, 15]. Each core is associated a local memory, or SPM for short. Each SPM is divided into two equal-size partitions, so that at run-time the scheduler can load a task in either partition. We say that a partition is free if there is no task currently loaded in the partition, and occupied otherwise. Jobs execute non preemptively, alternating between the two partitions: while we execute a job from one partition, we simultaneously unload the previously job from the other partition and load the next one using DMA. In details, the schedule of load/unload phases uses the following two rules [15]: (1) if there is a free partition and a task ready to be loaded, then the DMA is instructed to perform the load phase; (2) otherwise, if there is data to be unloaded for an already finished job, the DMA performs an unload phase. In either case, once started, a load/unload phase cannot be interrupted by another DMA phase on the same core.

Note that because two masters (a core and a DMA) may be concurrently accessing the same SPM, contention at the level of the core-local SPM may arise. In this case, care must be taken in appropriately evaluating the magnitude of contention-induced performance degradation. Alternatively, when using PL-defined SPMs, a design that minimizes the effect of core-DMA contention must be performed. In this paper, we use the latter approach.

Figure 1 depicts an example schedule (modified from [15]) with three tasks; up arrows represent release times. 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 per-task. 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. For this reasons, there are times when the DMA is not used. Also note that while load phases have higher priority over unloads, at time $t = 3$ (and similarly $t = 5$) 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 upon starting a load phase; hence, while τ_1 has higher priority than τ_3, the latter is executed at time 5 because τ_1 is released right after the start of the load phase at $t = 4$. This behavior causes blocking time on the higher priority task, which we account for in the analysis in Section V.

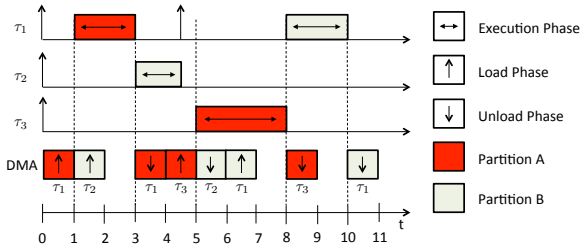


Fig. 1. Scheduling on one core.

Finally, note that so far we have focused on the DMA behavior from the point of view of a single core. However, to avoid contention in main memory and since our platform implements a single DMA component, in practice we have to serialize DMA transfers between the M real-time cores. The work in [15] used a TDMA arbitration among cores based on a fixed slot size; the slot size was designed so that it was possible to load or unload an entire scratchpad partition within

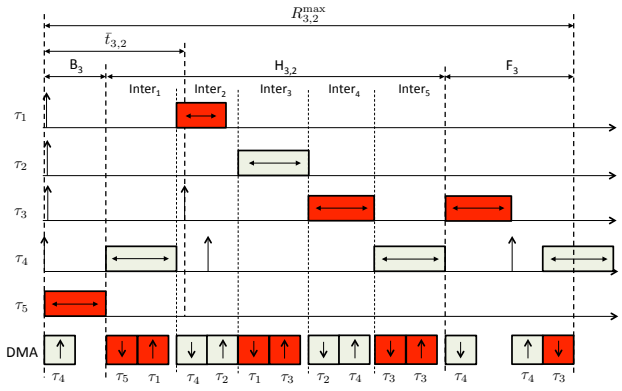


Fig. 2. Example critical instant. The task under analysis, τ_i, is τ_3. $R^{\max}_{3,2}$ is the response time for the second job of τ_3 from the critical instant.

a slot. While simple, we find two issues with such approach. First of all, it forces an even allocation of DMA bandwidth among cores, which might be undesirable in cases where the cores are heterogeneous, or applications are partitioned to cores based on their different memory demands. Second, it forces the TDMA slot length to depend on the largest scratchpad size, which is again undesirable if the SPM size is different per-core. Therefore, we propose an alternative mechanism, namely TDMA at a smaller granularity, with per-core slots. 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 program the DMA controller, during each slot we can only perform DMA transfers for a maximum of $\bar{\sigma}_j$ time, where $\sigma_j - \bar{\sigma}_j > 0$ represents the DMA overhead. We show that the DMA overhead in our target platform (Xilinx Ultrascale+) is small in Section VII.

V. Schedulability Analysis

We now show how to derive an upper bound $R^{\max}_i$ to the response time of a task under analysis τ_i. Since we employ partitioned scheduling for real-time tasks, we only focus on the core under analysis executing τ_i and use $\sigma, \bar{\sigma}$ for the DMA slot size, without and with overhead, assigned to the core. Since our scheduling model follows the same rules as in [16, 15], we can use the same response time analysis in [16] (Algorithm 2), extending it to account for arbitrary deadlines.

A. Critical Instant and Worst-Case Response Time

An example critical instant for a task set where $\tau_i = \tau_3$ is shown in Figure 2. The Figure shows a busy interval where two jobs of τ_i are executed. The critical instant is produced when the first job of τ_i in the interval, as well as the first job of each higher priority task, are simultaneously released immediately after the start of a load phase for a lower priority task (task τ_4 in the figure). This causes τ_i and higher priority tasks to be blocked by the execution of two lower priority tasks (τ_4 and τ_5). All other jobs are released as soon as possible, based on the their arrival curve $\tau_j.\alpha(t)$: formally, for the k-th job of task τ_j, we derive its release time as:

$$\bar{t}_{j,k} = \inf_{t \geq 0} \tau_j.\alpha(t) \geq k, \tag{1}$$

where inf denotes the infimum operator [1]. Also note that in the figure, the second job of τ_i is blocked by another job of τ_4: this is because two jobs of the same task cannot run back-to-back, due to the need to load and unload input and output data of the task.

Since in general a busy interval can contain multiple jobs of τ_i, we need to compute a response time bounds for all such jobs. Hence, let $R^{\max}_{i,k}$ be the response time for the k-th job under analysis of τ_i measured from the beginning of the busy interval. The response time of τ_i can then be bounded as:

$$R^{\max}_i = \max_k R^{\max}_{i,k} - \bar{t}_{i,k}, \tag{2}$$

and the task is schedulable if $R^{\max}_i \leq \tau_i.D$. Note that the number of jobs of τ_i contained in the busy interval can be determined based on $\tau_i.\alpha(t)$.

As an example, Figure 2 shows how to derive the response time $R^{\max}_{3,2}$ for the second job of τ_i. In general, the response time of the k-th job of τ_i is obtained as the sum of three parts, $R^{\max}_{i,k} = B_i + H_{i,k} + F_i$: (1) the initial blocking interval of length B_i; as proven in [15], this is the maximum between the execution time and the load phase of any lower priority task. (2) the interference interval of length $H_{i,k}$. This interval is composed of a sequence of smaller scheduling intervals (Inter_1 to Inter_5 in the figure); during each scheduling interval, the execution of a task is overlapped with a load and an unload operation. (3) The final interval of length F_i, where the job under analysis executes, and is then unloaded.

Since the job under analysis cannot be preempted once it starts executing, the window of time during which released higher priority jobs can interfere with it is bounded by $B_i + H_{i,k}$. Hence, the total number of interfering jobs of higher priority task τ_j can be computed as $\tau_j.\alpha(B_i + H_{i,k})$. In [16] it is then proven that the number of scheduling intervals in $H_{i,k}$ is bounded by:

$$I_{i,k} = \sum_{j=1}^{i-1} \tau_j.\alpha(B_i + H_{i,k}) + 2 \cdot k - 1. \tag{3}$$

In details, there are $\sum_{j=1}^{i-1} \tau_j.\alpha(B_i + H_{i,k})$ intervals of higher priority task, one blocking interval at the beginning of $H_{i,k}$, and $k-1$ jobs of τ_i, each of which can be further blocked by a lower priority task (for the example in the figure, this results in $I_{3,2} = 2 + 1 + 1 + 1 = 5$).

It remains to compute the maximum length of the $I_{i,k}$ scheduling intervals. Since a job cannot start until it has been loaded and the previous job has finished executing, the length of each scheduling interval is the maximum between the length of the executed phase, and the combined length of the load and unload phases in the interval. Since in general we cannot determine the precise sequence in which tasks are executed in the busy interval, Algorithm 2 in [16] determines an upper bound on $H_{i,k}$ [2] by creating a list of execution phases, and combines load plus unload phases, and then takes the $I_{i,k}$ largest such phases. Finally, note that $I_{i,k}$ in Equation 3 depends on $H_{i,k}$ itself; the circular dependency is solved through a standard response time iteration.

B. Scheduling Interval Length

The work in [16] considered a single-core system; hence, the DMA was dedicated to the core, and the length of the

[1] Note that the infimum, rather than the minimum, is required because the arrival curve might be left-discontinuous.

[2] $H_{i,k}$ is referred to as F in [16]; our notation mirrors the one in [15].

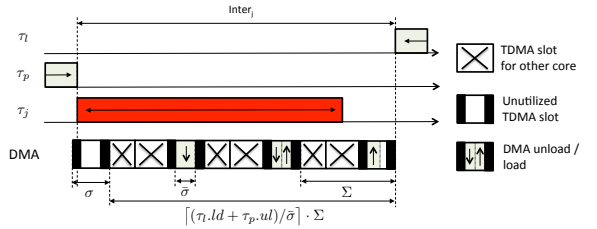


Fig. 3. Example DMA TDMA schedule for a scheduling interval.

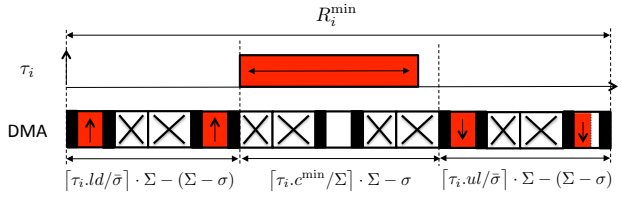


Fig. 4. Example best-case response time $R_i^{\min}$.

load and unload phase for a task τ_j was simply equal to $\tau_i.ld$, $\tau_i.ul$. In our system, the phase length must be adjusted to account for the fine-grained TDMA arbitration. Figure 3 shows the example of a generic scheduling interval Inter$_j$, where we execute task τ_j, and in parallel load task τ_l and unload task τ_p. In the worst case, the interval can start during a TDMA slot assigned to the core under analysis, forcing that slot to be wasted. Since DMA operations for the core under analysis can only be performed for $\bar{\sigma}$ time every Σ, the total number of TDMA rounds required to complete the unload phase of τ_p and the load phase of τ_l is equal to $\lceil(\tau_l.ld + \tau_p.ul)/\bar{\sigma}\rceil$, resulting in a total memory length of:

$$\sigma + \lceil(\tau_l.ld + \tau_p.ul)/\bar{\sigma}\rceil \cdot \Sigma. \tag{4}$$

The length of Inter$_j$ is then the maximum of $\tau_j.c^{\max}$ and Equation 4 [3].

A similar logic can then be employed to derive the length of the initial and final intervals B_i, F_i; combining the lengths B_i, F_i and $H_{i,k}$ then yields $R_i^{\max}$ based on Equation 2. Due to space limitations, we provide the corresponding derivations in appendix.

C. Best-case Response Time

Finally, Figure 4 details how to compute the best-case response time $R_i^{\min}$ for τ_i. In the best-case scenario, the task can be released when the core is idle and all partitions are free, at the beginning of a DMA TDMA slot assigned to the core. $R_i^{\min}$ can then be obtained as the sum of the load, execution and unload phases of τ_i. Referring to Figure 4, this yields:

$$\begin{aligned} R_i^{\min} = &\lceil\tau_i.ld/\bar{\sigma}\rceil \cdot \Sigma - (\Sigma - \sigma) \\ &+ \lceil\tau_i.c^{\min}/\Sigma\rceil \cdot \Sigma - \sigma \\ &+ \lceil\tau_i.out/\bar{\sigma}\rceil \cdot \Sigma - (\Sigma - \sigma). \end{aligned} \tag{5}$$

VI. Implementation

In this section we first provide the details of the architecture that we have considered. We then present a general overview of the implementation and explain the details for each component.

[3]Note that Algorithm 2 in [16] requires the memory length to be a linear combination of $\tau_l.ld+\tau_p.ul$. A linear bound on Equation 4 can be immediately obtained as $\sigma + \Sigma + \tau_l.ld \cdot \Sigma/\bar{\sigma} + \tau_p.ul \cdot \Sigma/\bar{\sigma}$.

A. Architectural Overview of the Platform

For our implementation, we have used Xilinx UltraScale + ZCU102 SoC. The specifications of the chip are shown in the Table I. There are two Cortex R5 cores with each having its own tightly coupled memory. These cores can either run in lock-step mode or can run independently. Also, there are four application (Cortex-A53) cores. All the application cores in the system have their own local instruction/data caches. The Last-Level Cache (LLC) is shared by all the application cores. There is no dedicated SPM provided for the application cores as is the case with the traditional high performance multicore processors that follow standard memory hierarchy.

The SoC also includes the programmable logic (PL) with multiple interfaces between the PL and the processing (PS) domain. There are three interfaces going from the PS side to the PL side. Out of the three, two are high performance interfaces (HPM0 and HPM1) where as the third interface is the low performance interface (LPD). There are also interfaces from the PL side to the PS side and they are named as HPC and HP. There is 3 MB of the block ram (BRAM) inside the PL. For rest of the section we will use BRAM and SPM interchangeably.

TABLE I
Architectural Features of ZCU102 Ultrascale+.

Chip Name	Xilinx Ultrascale Plus ZCU102
Architecture	ARM Cortex A53 and Cortex R5
CPU Units	4 x ARM Cortex A53 1.2 Ghz 2 x ARM Cortex R5 600 Mhz
Processing Units	DMA, GPU, CPU, Ethernet, Programmable Logic
Memory Hierarchy For A53	32KB Private I/D Cache 1MB Shared LLC
Memory Hierarchy For R5	32KB Private I/D Cache 128 KB of Local Tightly Coupled Memory
Interfaces Between (PL) and (PS)	2 x (HPM) (PS ->PL) 1 x LPD (PS ->PL) 2 x HPC (PL->PS) 4 x HP (PL->PS)
Memories	DDR 4GB 64-bit (PS) OCM 256KB (PS) DDR 512MB 16-bit (PL) Block RAM 3MB (PL)

B. Overview of Implementation

Using the characteristics of the considered platform there are many ways to show how one can achieve predictability or design mixed criticality domains. We experimentally explored various options on the platform, selecting as a final candidate the design depicted in Figure 5. In our proposed design we assign one of the A53 core to run Linux to take care of non-real time tasks and reserve the other three cores to execute critical tasks on top of a Real-Time Operating System (RTOS). A few noticeable features shall be noted: (i) the non-critical domain is assigned direct access to DRAM because this domain features applications with sizable footprint; (ii) each mid- and high-critical 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-critical domain also occupies a dedicated PS-PL interface to access its private SPM.

The non-critical core is also responsible for booting a hypervisor (Jailhouse). Jailhouse allows us to partition shared memory resources, especially the LLC and DRAM by implementing page coloring [12]. We have two partitions in the

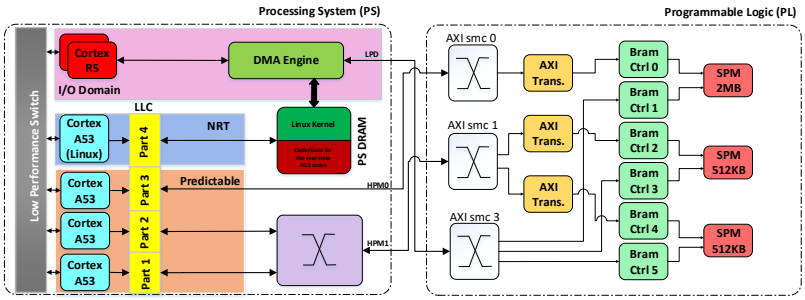


Fig. 5. PS-PL interface and design

DRAM; one to run the kernel from and another one to place the code/data of the tasks running on the real-time A53 cores.

We propose creating Scratchpad Memory (Block RAM) in the Programmable Logic (PL) for each core. In order for each core to access its own scratchpad memory (SPM) there needs to be a dedicated or fast enough interface such that each core can access its own SPM without seeing a delay from another core. Unfortunately, there are only two high performance interfaces between PL and PS available in the platform and three real-time A53 cores. Therefore, in our design we propose the use of one high performance interface that is shared between two A53 cores while the third core has dedicated interface to its own SPM memory.

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. This is because we found through our experiments that a core accessing the data from the BRAM over the LPD interface is similar in performance to the case when two cores are accessing the same HPM interface. Hence, we decided to use a shared interface for two A53 core, while the third has its own independent HPM interface (see Figure 5). Another reason for doing that is to keep the LPD interface so that we could perform the DMA transfers to/from the SPM/DRAM on the behalf of the real-time A53 cores. The real-time cores rely on the TDMA based scheduling of the DMA to load/unload tasks from the SPM memory in the PL. The TDMA based scheduling of the DMA is handled using the real-time R5 core.

We assign 2 MB of SPM memory to the A53 core that has a dedicated HPM interface to the PL and a 512 KB to the other two real time cores that share HPM1. From our experiments we also found out that having a single ported SPM in the FPGA lowers the performance when two entities are accessing it at the same time. Since we want to pipeline the execution of the current task with the load of the next task, we divide the SPM into two halves. Moreover, we chose a dual ported BRAM so that DMA and the A53 core both can directly write to it at the same time.

In order to avoid the conflict between all the A53 cores we partition the LLC by coloring since there is no hardware support provided on the platform to achieve that. However, coloring generally results in size reduction of the memory which in terms of DRAM is fine because it is generally big. Since we have very limited SPMs in the PL we end up reducing the size of it by 1/4 as a result of coloring. To avoid the coloring effect on the BRAM, we introduce address translator between the A53 and the BRAM path (which is detailed in Section VI-E).

In the following subsections we provide a detailed discussion on each of the main components including Jailhouse, page coloring, address translator, how the different A53 cores communicate using the hypervisor, RTOS, and how the tasks are relocated from the PS DRAM to the SPM.

C. Jailhouse to partition the Shared Resources

As the hypervisor we use Jailhouse, which is a statically partitioning, non-scheduling, real-time hypervisor [20]. Jailhouse inserts "virtual barriers" in the shared resources of a multicore processors, such as I/O peripherals and processing cores, transforming a symmetric multiprocessing (SMP) system into an asymmetric multiprocessing (AMP) system. Jailhouse is in fact a Linux driver (a mix of type-1 and type-2 hypervisors) and favors simplicity and low overhead over sophisticated (para-)virtualized techniques, which is ideal for real-time systems [20]. Thus, it requires at least one core to be assigned to Linux. Once the driver is loaded, it takes over control of the entire hardware and reassigns the hardware back to Linux based on a configuration file (named root cell). 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 domain [20]. The idea is to have non critical tasks running on the Linux cell and critical tasks running on isolated partitions on top of a Real-Time Operating System (RTOS).

Virtual memory of root and non-root cells are handled by Jailhouse. In case of this work, the A53 cores (ARM 64 bits architecture) has a two-stage virtual memory translation. A guest OS, such as Linux or an RTOS that supports virtual memory, translates virtual addresses (VA) to intermediate physical addresses (IPAs) in a transparent way, which means that the OS has its own page tables. Jailhouse then translates IPAs to physical addresses (PAs) and also has its own page tables.

As the RTOS for critical tasks, which runs on non-root cells, we use Erika Enterprise version 3, which is open-source and OSEK/VDX certified [21, 22]. Erika supports fixed-priority scheduling with resource access protocol and runs on top of our Xilinx Ultrascale+ platform. We discuss the modifications in Erika to support our system model in Section VI-G.

D. Page Coloring

In order to enforce strong inter-domain (inter-cell) and hence inter-core performance isolation, we leverage page coloring. In a nutshell, a physically-tagged, physically indexed set-associative cache can be pictured as a 2D array. If we move horizontally in the array, we move across *ways*; if we move vertically in the array we move across *sets*. If a read/write

request for a cacheable memory location results in a cache miss, allocation in cache occurs at a certain (set, way) coordinate. The set is uniquely determined by the value of a group of bits in the physical address being accessed. These bits are referred as *index bits*. Next, the way is determined by the replacement policy implemented by the controller (e.g. LRU, FIFO, pseudo-random).

Assume now that we are able to assign physical addresses (and hence memory) with non-overlapping indexes to different criticality domains. In this case, applications on domain A cannot cause cache evictions on lines allocated by domain B, which is the premise to achieve temporal isolation in shared caches. Contiguous physical memory regions have increasing addresses and thus increasing cache indexes. It follows that in order to partition cache space, different domains need to be assigned to *interleaved* portions of physical memory.

For instance, in a 2 MB cache with 16-ways, where each line is 64 bytes, each way contains 2048 cache lines. As such, we have 11 index bits. By leveraging on virtual memory, it is possible to assign physical memory to do domains at the maximum granularity of a single memory *page*. A page is usually 4 KB in size. Each page contains $2^6 = 64$ lines (each of 64 bytes). This implies that only 5 out of the 11 total bits can be directly controlled by software when allocating physical memory pages. The *color* of the page is then determined by the exact value of these 5 bits, which go under the name of *color bits*. Clearly, lines in pages with different colors cannot evict each other in cache when accessed.

It is possible to modify the physical memory allocator in the OS to be aware of the structure of a cache, and to implement a color-aware allocation [23, 5, 24]. The required modifications, however, can be heavy and have unforeseen consequences on the stability of the OS. Moreover, the approach is not applicable in case of closed-source OS's.

In our approach, we leverage virtualization extensions. In fact, we 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. One the one hand, it allows us to localize the changes required to implement coloring-based partitioning in a software component (Jailhouse). On the other hand, it allows deploying unmodified and possible closed-sources OS inside our criticality domains. A similar technique was used in [8, 12].

E. Address Translator due to Cache Coloring

Due to the use of cache coloring to partition the SPMs, we use only one-fourth (four colors, one for each core) of the SPM capacity which is not an optimized approach in terms of memory usage. To overcome this limitation, we designed an in-house AXI (Advanced eXtensible Interface) full hardware IP which is responsible for the destination address translation coming from the PS to the programmable logic path. A path here refers to an AXI connection between PS master ports including high- and low-performance ports and targeted hardware IP blocks. All the modules in programmable logic are connected via AXI4 interconnects.

To access an SPM with a size of 2 MB, for instance, we need 21 bits of address provided by one of the master ports from PS. With cache coloring enabled, two bits of the address will be used for the color and consequently, we can only address the SPM controller with 19 bits. As a solution, instead of receiving 21 bits of address, the AXI Translator IP receives 23 bits (8 MB) from the PS through the AXI, removes the specific address bits from that, and passes it to the SPM controller. Assuming that the address of the AXI Translator in Figure 5 has a range from 0xA0000000 to 0xA07FFFFF which is 8 MB space, bits 12 and 13 are responsible for the cache coloring. As an example, a request address of 0xA0023456 from a core to the AXI Translator would remap to 0x008456 in a 2 MB space.

Figure 5 also shows the address translator IP placement in the design. The implemented AXI Translator module has one slave port to be able to communicate with the master port (connected to PS) and one master port responsible for passing the translated address to the slave port of the SPM controller. We need three AXI Translators in which the requests coming from each core should be translated. With this mapping mechanism, the SPM capacity is not affected by the cache coloring (we do not lose space) and since the AXI 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 FF and LUTs counts compared with the design without any translation IP are 0.57% and 0.41% respectively while the SPM count remains the same.

F. Communication among Jailhouse Cells

The communication among Jailhouse cells (either root or non-root cells) is accomplished by Inter-Process Interrupts (IPIs). Originally, Jailhouse supports IPIs only among processors that are assigned to the same cell. Since in our platform each cell has only one processor (one for Linux and three for Erika RTOS instances), we had to modify Jailhouse to send an IPI to a processor that is not in the cell configuration of the sender processor. IPI is required to interrupt Erika when new data has arrived for its tasks. We changed Jailhouse to bypass the IPI number 15. So whenever a processor issues an IPI 15, Jailhouse delivers it to the receiver processor even if it is not in the sender's cell configuration. Thus, we allow Linux (root cell) to send an IPI to an Erika instance (non-root cell) whenever new data for a task running on the non-root cell arrives. To allow the communication among the OSes, we rely on FIFO buffers.

G. Erika RTOS Running on Real-Time Cores

This section provides the details of the Erika OS running on real-time A53 cores. All the real-time A53 cores run a partitioned fixed priority scheduler and always execute from dedicated SPM memory assigned to them. The SPM used by a dedicated real-time A53 core and the DRAM memory used the non-real time A53 running Linux are both colored to avoid cache evictions in the shared cache.

Erika OS does not support virtualization on ARMv8 CPUs, as such it would not use VAs. By default, however, Jailhouse performs the setup of a flat 1:1 stage-one (VA→IPA) addressing space before booting any non-root cell. This is required to support cacheable memory. An application in the Erika OS is always statically compiled against VA/IPA addresses.

As shown in Figure 6, the task running on the Erika core can be in one of the following states:

- **Running:** The tasks is executing from BRAM.
- **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 to be released.

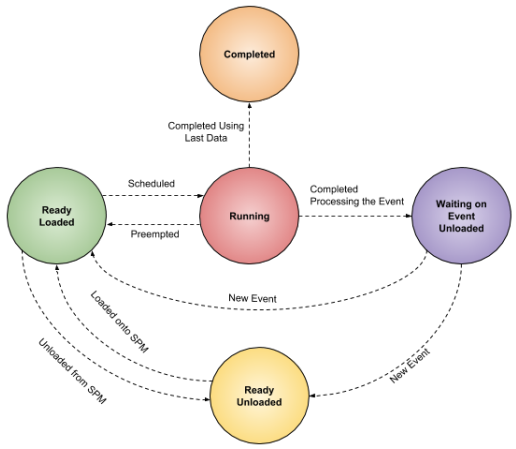


Fig. 6. Overview of the Different States of a Task in SPM-Stream OS.

To allow the load and unload of code and data of Erika's tasks, we use the support for virtual memory implemented in Jailhouse. We detail how code/data relocation in the next subsection.

H. 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, as discussed in the previous section. Relocation is initiated by the Erika OS, 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 VA (= IPA) addresses. As such, relocation is performed by modifying the IPA→PA address mapping 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, as shown in Figure 7. 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 the loaded to/unloaded from (the largest SPM is 2 MB, so the maximum offset is 512-1, 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 [25], the overhead of a hypercall in Jailhouse on the Ultrascale+ platform is around 400 ns.

```
1 /* 0x4a48 is the Jailhouse hypercall number
2  * r0 contains the hypercall ID
3  * r1 and r2 are the two arguments */
4 asm volatile ("hvc #0x4a48" : "=r" (__r0) : "r" (__r0),"r" (__r1),"r" (__r2));
```

Fig. 7. Erika code to perform a Jailhouse hypercall.

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 IPA 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 OS). The effective copy of the task into/from SPM is performed by the DMA, as previously described.

VII. Evaluation

In this section we present the evaluation of our proposed schedulability test and architecture. We start the evaluation showing an experimental analysis of the target platform in Section VII-A with benchmarks. We then evaluate DMA in Section VII-B and present the case study of an anomaly detection application in Section VII-C.

A. EEMBC and SD-VBS Benchmarks

In order to benchmark our proposed system design, we first run some EEMBC benchmarks [26]. Since the EEMBC benchmarks are very small and are not designed for high performance processor, we do not see the slow-down of running them solo versus contention case. We chose to present here only FFT and DCT since all of them show a similar pattern. Here the solo case refers to the scenario where the A53 core is accessing one of the partition of dual-ported SPM memory using its dedicated HPM interface. Whereas in the contention case the benchmark is executing from the DRAM while all the other A53 cores are also accessing the DRAM. Cache coloring is enabled in both the experiments.

We also tried the SD-VBS [27] suite that includes vision benchmarks, since these are computation and data intensive. Based on the related work [28, 15], we chose two applications that present high usage of memory (disparity and mser) and ported them to Jailhouse. We then executed them with SQCIF input data either from SPM (solo) or from the PS DRAM (when all the cores are also stressing the DRAM). The results of the SD-VBS in Figure 8 show that the disparity when executing from the SPM has an improvement of 11 percent. Whereas, the mser shows an improvement of approximately 46 percent compared to the contention case in the DRAM.

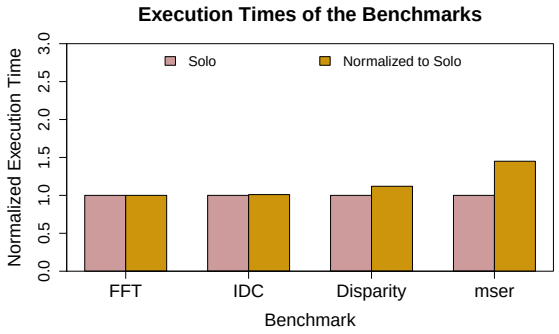


Fig. 8. SD-VBS and EEMBC Benchmarks Solo Vs Contention

B. DMA Evaluation

In order to move data between the PS DRAM and the SPM memory inside the PL, we use the PS side DMA. Since there is only one DMA and three A53 cores, we propose a fine granularity TDMA-based scheduling of the DMA. The

TABLE II
OBTAINED WCET OF THE BENCHMARKS.

Benchmark	BRAM Solo (us)	DMA Contention (us)
Disparity	108130.99	119529.54
mser	6315.21	9272.38
IDC	216180.62	217454.01
FFT	5206.47	5206.42

scheduling of the DMA module in our system is handled using an ARM Cortex R5. Moreover, the DMA transfers the data between DRAM and the SPM using the LPD interface which is a low power domain. Table VII-B below provides the DMA transfer time as well as the overhead of programming the DMA.

Parameters	Value
DMA Copy Time (1 MB)	880 us
DMA Programming Overhead	3.89 us

C. Case Study: Anomaly Detection

Real-time embedded systems are frequent targets for malicious attacks [29]. Recent examples include disabling a car's brake system [30], CAN bus attack [31], and injecting a fatal dose of insulin in an insulin pump [32]. Anomaly-based detection algorithms monitor the generated data by the target embedded system and detect deviations from pre-established specification of normal behavior [29]. In this case study, we execute a set of anomaly detection algorithms on top of our predictable platform using data generated by an autonomous car. We consider the following anomalies in the case study:

- **Spikes:** a spike is a sequence of contiguous samples that lie farther a given number of standard deviations from the current mean of the input data.
- **Clipping and Loss:** a clipped data signal represents a series of identical samples at the maximum or minimum extend of the sample medium. Loss indicates a complete lost of the input data signal.
- **Level Change:** this symptom is observed when the mean of the input data changes significantly in a short amount of time and then remains consistent at the new level.
- **Frequency Change:** it occurs when the primary frequency of a data signal changes over a short period of time.
- **Sampled Value Flip-Flop:** this anomaly occurs when a sampled value flip-flops in a short period of time. For instance, the gear in a vehicle cannot change from rear to drive and return to gear in a couple of microseconds.

For detecting each of these anomalies, we execute a specific anomaly detector. The Spike detector keeps tracks of the input data mean and standard deviation (STD) in a buffer. When the STD increases or decreases by a defined factor (for instance, mean + 10 * STD), it issues an anomaly alert. The level change detector calculates the new mean after a data is received and if the difference between the new and old means is greater than a defined value, a level change is detected. The clipping detector keeps a buffer with N values and for each new data it verifies how many values are equal to the maximum value in the buffer (or minimum). If there are an amount of data points with the same value greater than a defined value, then there is a clipping. If there is an amount of zeros greater than a defined value, then there is a loss. The spectrum detector uses Fast-Fourier Transformation (FFT) and math calculations to detect changes in the frequency domain. Finally, nfer, a recently introduced language and system for inferring event stream abstractions, detects the sampled value flip-flop anomaly [33]. The output from each detector is a boolean informing whether an anomaly was found or not in the input data, the timestamp in which the anomaly has occurred, a note informing the type of the anomaly, and the detector name (256 bytes in total). We also consider that a Voter task receives the output from all detectors and processes them to finally output if the anomaly is a true or false positive. We want to provide a predictable end-to-end delay for detectors and the voter, so that anomaly detection has timing guarantees.

Figure 9 shows an overview of the communication flow of the tasks in the case study and a possible assignment of tasks to real-time cores. The anomaly detection algorithms were implemented in different Erika instances as tasks on top of Jailhouse and using the Xilinx ZCU102 Ultrascale+ platform. Recall that each Erika has its own SPM memory and uses a different cache partitioning, as described in Section VI. The autonomous car data were collected during operation and stored in a database. The data is streamed into our case study through the network using the Redis publish-subscriber interface based on channels [34]. The detectors subscribe to channels and receive data once it arrives on those channels. A channel here means input data from a specific sensor or controller, and hence its input frequency is known. Examples of data for the channels are those from the Gear, Throttle, and Steering subsystems of the car. All channels have an input frequency of 50 Hz (one new data at every 20 ms). Finally, the Voter is a periodic task; every time it runs, it reads all available outputs of the five detectors. It is important to highlight that once the detectors are integrated into the autonomous car, the data does not need to be stored, it would go directly from the sensors/controllers to the detectors, without passing through Linux. In our case study, we use Linux to allow the streaming of data at the same rate they were generated in the car. Although Linux may add unpredictable behavior, we run it in a dedicated core with cache partitioning.

The autonomous car data (from Gear, Steering, Throttle, etc) also contains the timestamp in which data was collected, the input frequency, and the Redis channel name. A data point has 64 bytes in total. A Redis client running on Linux receives the data and notifies the DMA controller running on the R5 core. The DMA controller then gets the received data and transfers it to the detector's SPM memory (in case it is loaded) or DRAM memory (in case it is unloaded). After that, the controller requests Jailhouse to interrupt the Erika instance, informing that new data has arrived in the subscribed channel and to load/unload tasks accordingly (i.e., local scheduling in Erika as shown in Section VI-G).

Table III shows the execution time, memory consumption, and DMA load/unload times for each Erika task (detector). The execution time was measured using input data from the autonomous car and represents the obtained WCET in each detector (*i.e.,* time to process the buffer or input data, depending on the detector). NFER is the one with more data, and thus with the longest load and unload times. Spectrum is the longest in terms of execution time, because it computes FFT and deals with array transformations.

We then use the described case study to compare our proposed schedulability test in Section V with two alternatives: (1) the analysis in [15], which assumes a fixed DMA slot size, each sufficient to load/unload the largest SPM partition as discussed in Section IV. We extend the analysis to arbitrary deadlines by considering the response time of each job under

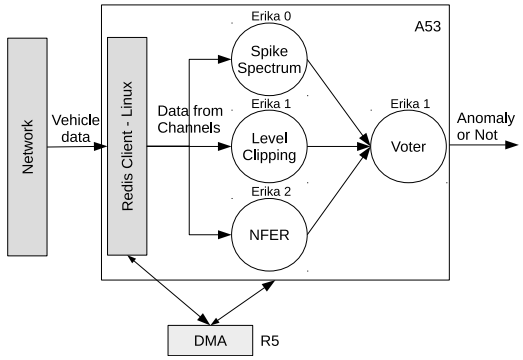


Fig. 9. Overview of one possible configuration of the case study on anomaly detection, showing the communication among tasks (detectors) and the assignment of tasks to cores.

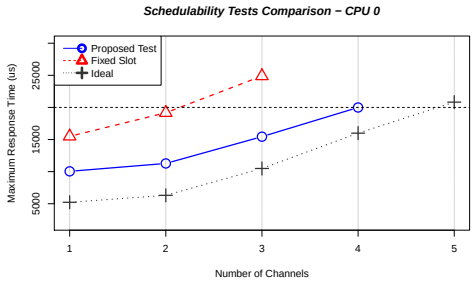
TABLE III
TIMING AND MEMORY DEMANDS OF THE ANOMALY DETECTION ALGORITHMS.

Detector	BRAM WCET (us)	Code Size (bytes)	Data Size (bytes)	DMA Transfer Time Code+Data (us)
Spike	1784	696	6400	6.23
Clip/Loss	1008	336	12800	11.54
Level	1160.4	796	7680	7.44
Spectrum	4169.6	2520	1912	3.89
NFER	3185.6	15400	309664	285.45
Voter	300	8223	1280	8.34

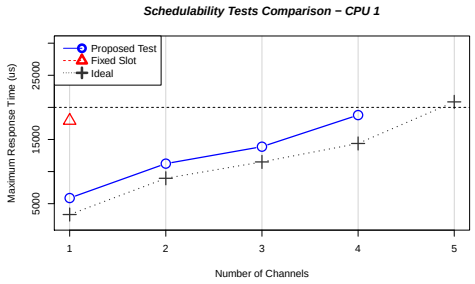
analysis in the busy interval, similarly to Section V-A. Note that to ensure a fair comparison, outside of the DMA schedule, all other parameters are derived from our implemented FPGA platform, rather than the COTS microcontroller used in [15]; and (2) an ideal (but unrealistic) system, where tasks execute from main memory without suffering any contention. To avoid cache-related preemption delays, we consider a standard fixed-priority non preemptive scheduler.

For defining the size of the fixed DMA slot needed by the test in [15], we used the same methodology as described in [15]. We consider the SPM with two partitions of 512 KB each (1 MB in total), which is large enough to accommodate the largest detector (NFER) in one partition and still is a multiple of two. Then, we get the DMA time to transfer 1 byte (0.0009 µs - as shown in Section VII-B), and multiply it by 512 KB, resulting in the DMA fixed slot size of 471.85 µs. The DMA round length is then set to DMA fixed slot size multiplied by three (the number of real-time cores).

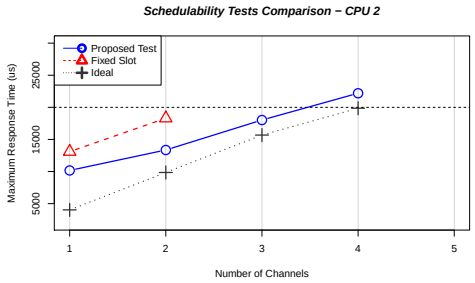
As shown in Section VII-B, the overhead to program the DMA in the ZCU102 platform is 3.98 µs. For our schedulability test, we consider the DMA overhead to be equal to 4% of the DMA slot size (σ), resulting in $\sigma = 100$ µs. To allow a fair comparison with [15], we split the DMA bandwidth evenly among the three cores, resulting in a DMA round length (Σ) of 300 µs. We set the period of all detectors to 20 ms, which is the period of the input data. For the Voter task, we set its period to 15 ms, so we are sure it gets the most up to date output from the detectors whenever it runs. Deadlines are equal to periods. Considering this configuration, the total utilization for one input data channel with the 6 tasks is 0.58 (WCETs from Table III and periods of 20 ms and 15 ms). We then vary the number of input data channels from 1 to 5, which in fact increases the number of tasks and the total utilization of the system. As we increase the number of tasks, we statically assign them to the three real-time cores so that the utilization in each core remains balanced. We compute the response time for all tasks in all scenarios and present the maximum response time of any task executing in the same core, obtained by each schedulability test.



(a)



(b)



(c)

Fig. 10. Maximum response time (per CPU) for each schedulability test as the number of channels in the case study increases.

Figure 10 shows the obtained maximum response time per schedulability test and per core. On the x-axis we vary the number of channels, while on the y-axis we present the response time in µs. The dashed horizontal line at 20000 represents the deadline (20 ms) for the detector tasks (recall that the voter task has a deadline of 15 ms). A point above the dashed line means that the maximum response time is greater than the deadline (task set is not schedulable). We do not plot a data point in the graph when the schedulability test did not converge after a threshold (period * 4); for instance, as in the Fixed slot line in the CPU 1 graph with 2 channels. Note that with 6 channels, the utilization on all cores exceeds 100%.

We can clearly note that our proposed test is superior when compared to [15]; the reason is that [15] forces the DMA slot to be sized based on the transfer time for NFER, which is significantly larger compared to the other tasks. In particular, note that the results for our proposed test and [15] are closer on CPU 2, where NFER is allocated, while they show a

much larger difference on CPU 0 and 1. While our proposed technique supports up to 3 channels, [15] supports only one (over 100% of more utilization). For a channel number of 4, our proposed technique is schedulable in CPUs 0 and 1, but is a little bit above the deadline in CPU 2 (22 ms). As long as we relax the deadline to be slightly larger than period then we achieve the same performance as the ideal case. We implemented the schedulability tests and obtained the results using SchedCAT [35] tool.

VIII. Conclusion and Future Work

In this paper, we have showed how one can design mixed criticality applications by exploiting the HW features provided by a modern heterogeneous SoC architecture that incorporates a multiprocessors and Programmable Logic (PL) . Using the PL provided in the system, we create a separate piece of ScratchPad Memory (SPM) to guarantee predictable execution for the real-time cores that run tasks with hard real-time requirements. A new finer-granularity DMA scheduling scheme and associated schedulability analysis of the tasks running from the SPM is provided. We describe a full-stack implementation of the proposed techniques and evaluate the system based on existing benchmark suites and a detailed case study.

References

[1] 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.

[2] 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.

[3] 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.

[4] G. Gracioli and A. A. Fröhlich. Two-phase colour-aware multicore real-time scheduler. *IET Computers Digital Techniques*, 11(4):133–139, 2017. ISSN 1751-8601.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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. ISSN 1539-9087.

[13] 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.

[14] A. Crespo, P. Balbastre, J. Sim, J. Coronel, D. Gracia Prez, and P. Bonnot. Hypervisor-based multicore feedback control of mixed-criticality systems. *IEEE Access*, 6:50627–50640, 2018. ISSN 2169-3536.

[15] 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.

[16] 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.

[17] Osek/vdx website, Oct 2018. URL http://www.osek-vdx.org/. Online; accessed 18 October 2018.

[18] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System architecture evaluation using modular performance analysis: a case study. *International Journal on Software Tools for Technology Transfer*, 8(6):649–667, November 2006.

[19] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the symta/s approach. *IEE Proceedings - Computers and Digital Techniques*, 152(2):148–166, March 2005.

[20] 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.

[21] 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.

[22] Evidence. Erika enterprise RTOS v3, Oct 2018. URL http://www.erika-enterprise.com/. Online; accessed 16 October 2018.

[23] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: A dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 381–392, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2809-8.

[24] Noriaki S., Hyoseung K., Dionisio de N., Bjorn A., Lutz W., Mark K., and Ragunathan (Raj) R. Coordinated bank and cache coloring for temporal protection of memory accesses. In *Proceedings of the 2013 IEEE 16th International Conference on Computational Science and Engineering*, CSE '13, pages 685–692, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-5096-1.

[25] 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.

[26] EEMBC AutoBench v1.1 - data book.

[27] San Deigo Vision . http://parallel.ucsd.edu/vision/, Accessed Oct. 2018.

[28] 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.

[29] S. Lu and R. Lysecky. Time and sequence integrated runtime anomaly detection for embedded systems. *ACM Trans. Embed. Comput. Syst.*, 17 (2):38:1–38:27, December 2017. ISSN 1539-9087.

[30] C. McCarthy, K. Harnett, and A. Carter. Characterization of potential security threats in modern automobiles: A composite modeling approach. Technical report, National Highway Traffic Safety Administration, Washington, 2014.

[31] A. Taylor, S. Leblanc, and N. Japkowicz. Anomaly detection in automobile control network data with long short-term memory networks. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 130–139, Oct 2016.

[32] E. Marin, D. Singelée, B. Yang, I. Verbauwhede, and B. Preneel. On the feasibility of cryptography for a wireless insulin pump system. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 113–120, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3935-3.

[33] S. Kauffman, K. Havelund, and R. Joshi. nfer–a notation and system for inferring event stream abstractions. In *International Conference on Runtime Verification*, pages 235–250. Springer, 2016.

[34] M. Dunne, G. Gracioli, and S. Fischmeister. A comparison of data streaming frameworks for anomaly detection in embedded systems. In *Proceedings of the 1st International Workshop on Security and Privacy for the Internet-of-Things (IoTSec)*, Orlando, FL, USA, 2018.

[35] B. Brandenburg. SchedCAT: Schedulability test collection and toolkit, Oct 2018. URL https://www.mpi-sws.org/~bbb/projects/schedcat. Online; accessed 10 October 2018.

Appendix

A. Analysis: Initial and Final Interval Lengths

We now compute the maximum length of the initial blocking interval B_i and final interval F_i. Figure 11 shows an example for B_i, where τ_{l1} and τ_{l2} are the lowest priority tasks with the

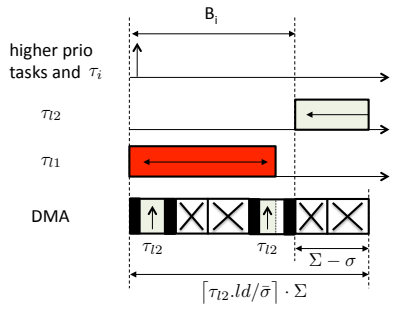


Fig. 11. Example initial blocking interval with length B_i.

longest execution and load phases, respectively. As in Figure 2, τ_i and all higher priority tasks arrive just after the load phase for τ_{l2} has started. This causes the interval length to be bounded by:

$$B_i = \max\left(\tau_{l1}.c^{\max}, \lceil \tau_{l2}.ld/\bar{\sigma} \rceil \cdot \Sigma - (\Sigma - \sigma)\right). \quad (6)$$

For F_i, we employ the following lemma, which follows directly from Lemma 3 in [15].

Lemma 1: The length of the final interval is bounded by the maximum of: (1) the execution time of τ_i, followed by one load phase of any other task and the unload of τ_i; or (2) one unload and one load phase for any two tasks, plus the unload of τ_i,

Note that Figure 2 shows the first case in the lemma, which is usually the worst one. In this case, a new job of τ_4 arrives right at the end of the execution of the job under analysis; since load phases have higher priority over unload phases, this forces the load of τ_4 to be completed before τ_i is unloaded. Based on Lemma 1, and computing the memory phase length according to Equation 4, we obtain:

$$F_i = \max\big(\tau_i.c^{\max} + \sigma + \lceil (\tau_l.ld + \tau_i.ul)/\bar{\sigma} \rceil \cdot \Sigma, \\ \sigma + \lceil (\tau_l.ld + \tau_p.ul + \tau_i.ul)/\bar{\sigma} \rceil \cdot \Sigma\big), \quad (7)$$

where τ_l and τ_p are the two tasks (different than τ_i) with longest load and unload phases.

B. Analysis: Schedulability Tests

Based on Section V-A, V-B, and Appendix A, Algorithm 1 derives a sufficient schedulability analysis for the task set. For each task τ_i, at lines 4-19, the algorithm iterates over the index k of the job under analysis of τ_i. For each job, the iteration at lines 5-13 computes an upper bound $\bar{R}_{i,k}$ on the length $B_i + H_{i,k}$ of the interference window; $\bar{R}_{i,k}$ is used to derive the number of interfering intervals $I_{i,k}$ at line 5. The new value of $H_{i,k}$ is then obtained by summing the length of the longest $I_{i,k}$ execution or memory (load plus unload) phases. If at any point the response time of the job, based on its release time $\bar{t}_{i,k}$, becomes higher than the deadline, the algorithm terminates unsuccessfully at line 10. Otherwise, the response time $R_i^{\max}$ of the task is updated based on Equation 2. If the length of the busy interval is sufficiently long to include the release on the next job $k+1$, then the iteration at lines 4-19 is repeated. If the maximum response time of all tasks is less than or equal to their deadlines, we terminate successfully at line 22.

Algorithm 1 Schedulability test

Input: A task set according to Section III
Output: Task set is schedulable or undecided

1: **for** $i = 1 \ldots N$ **do**
2: Compute B_i, F_i based on Equations 6, 7
3: $k \leftarrow 1, R_i^{\max} \leftarrow 0$
4: $\bar{R}_{i,k} = B_i$
5: compute $I_{i,k}$ based on Equation 3 using $B_i + H_{i,k} = \bar{R}_{i,k}$
6: Let E, DMA be the set of $I_{i,k}$ execution and memory phase lengths of scheduling intervals
7: Sort $M = E \cup DMA$ by decreasing order
8: $H_{i,k} \leftarrow \sum_{j=1}^{I_{i,k}} M_j$
9: **if** $B_i + H_{i,k} + F_i - \bar{t}_{i,k} > \tau_i.D$ **then**
10: **return** UNDECIDED
11: **end if**
12: **if** $B_i + H_{i,k} > \bar{R}_{i,k}$, then $\bar{R}_{i,k} \leftarrow B_i + H_{i,k}$ **then**
13: go back to 5
14: **end if**
15: $R_{i,k}^{\max} \leftarrow \bar{R}_{i,k} + F_i$
16: $R_i^{\max} \leftarrow \max(R_i^{\max}, R_{i,k}^{\max} - \bar{t}_{i,k})$
17: **if** $R_{i,k}^{\max} > \bar{t}_{i,k+1}$ **then**
18: $k \leftarrow k + 1$
19: go back to 4
20: **end if**
21: **end for**
22: **return** SCHEDULABLE