Real-Time Systems
https://doi.org/10.1007/s11241-019-09340-0

A real-time scratchpad-centric OS with predictable inter/intra-core communication for multi-core embedded systems

Rohan Tabish[1] · Renato Mancuso[2] · Saud Wasly[3] · Rodolfo Pellizzoni[4] · Marco Caccamo[5]

Abstract
Multi-core processors have replaced single-core systems in almost every segment of the industry. Unfortunately, their increased complexity often causes a loss of temporal predictability which represents a key requirement for hard real-time systems. Major sources of unpredictability are shared low level resources, such as the memory hierarchy and the I/O subsystem. In this paper, we approach the problem of shared resource arbitration at an OS-level and propose a novel scratchpad-centric OS design for multi-core platforms. In the proposed OS, the predictable usage of shared resources across multiple cores represents a central design-time goal. Hence, we show (i) how contention-free execution of real-time tasks can be achieved on scratchpad-based architectures, and (ii) how a separation of application logic and I/O operations in time domain can be enforced, and (iii) how predictable asynchronous inter/intra-core communication between tasks can be performed. To validate the proposed design, we implemented the proposed OS using commercial-off-the-shelf (MPC5777M) platform. Experimental results show that novel design delivers predictable temporal behavior to hard real-time tasks, and it provides performance gain of upto 2.1 × compared to traditional approaches.

Keywords IPC · real-time · Predictability · Multi-core · embedded systems · Scratchpad · Operating system · Inter-core and intra-core communication

1 Introduction

Multi-core platforms are mainstream products. Multi-core chips allow different processing tasks to execute in parallel while accessing a set of shared hardware resources,

✉ Rohan Tabish
rtabish@illinois.edu

Extended author information available on the last page of the article

Published online: 05 September 2019

Table 1 Suitable commercial multicore COTS platforms

Features	MPC5777M	MPC5746M	TMS320C6678	ARM Cortex-R8
Scratchpad	✓	✓	✓	✓
DMA engines	✓	✓	✓	✓
Dedicated I/O bus	✓	✓	✗	Impl. dependent
Number of cores	3 + 1 (lockstep)	3 + 1 (lockstep)	8	4

including: main memory, buses, caches, and I/O peripherals. However, the extensive sharing of hardware resources by different cores in a multi-core platform can heavily affect the system's temporal behavior and its predictability. From a real-time point of view, unregulated contention on shared resources induces significant execution time variance. Hence, specific mechanisms to manage and schedule shared resources need to be designed and validated. This problem has also been acknowledged by the Federal Aviation Administration (FAA), which currently imposes the use of a single core for safety-critical avionic applications unless proper analysis and mitigation of inter-core interference channels are demonstrated (FAA 2015).

This problem has been approached from different perspectives: (a) novel multi-core hardware platforms have been designed (Bui et al. 2011; Ungerer et al. 2010), (b) new OS-level techniques have been developed to perform shared resource partitioning and management on commercial-off-the-shelf (COTS) platforms (Mancuso et al. 2015). While a hardware solution might be desirable to meet the needs of modern real-time systems, it does not represent a viable solution in the short term for the embedded industry. Conversely, enforcing determinism at software level on a general-purpose COTS architecture may trade some performance with execution time predictability. In this work, we propose an approach that lies in between with respect to the aforementioned methodologies. In fact, (i) we consider a segment of COTS platforms that are designed to support desirable features for hard real-time computation as shown in Table 1 and (ii) redesign parts of the operating system (OS), leveraging such features to guarantee predictability and preserve performance. With these objectives in mind, we focus on emerging embedded scratchpad-based multi-core platforms. Scratchpad memories, in fact, have been proven to provide better temporal isolation when compared to traditional caches (Puau and Pais 2007; Metzlaff et al. 2011). Alongside, we exploit additional hardware features that vendors are now including in some modern families of multi-core platforms designed for the embedded market, such as: separate I/O and memory buses, the presence of dual-port memories with direct memory access (DMA) support, and core specialization. Thereby, this work provides following contributions:

1. A novel operating system design is built ground-up to achieve temporal predictability. Our OS design targets multi-core embedded COTS platforms and exploits core specialization and low level resource management policies.
2. To the best of our knowledge, this is the first OS that integrates a scratchpad-based CPU scheduling mechanism with a task schedule-aware I/O subsystem.

3. A description of how predictable communication between the tasks running on same as well as different cores can be achieved when scratchpad-based scheduling is performed.
4. A novel analysis is derived to calculate the response time of real-time tasks under the proposed scheduling strategy.
5. Finally, a full implementation of the proposed OS has been performed using a commercially available MPC5777M multi-core micro-controller. Its design has been validated using a combination of synthetic tasks and EMBC benchmarks.

Before dwelling into the details of this paper, we would like to highlight that the proposed execution model and resulting OS design are beneficial for tasks that perform a large number of memory accesses over a relatively small memory footprint—i.e. data-intensive applications (Metzlaff et al. 2011). This is because the main advantage of our proposed model is that overhead of accessing the global (slower) memory is encountered only once using the DMA, and this can be further reduced by pipelining workload execution and memory accesses. After the overhead of loading a task's code and data into a local/faster memory is completed, the task executes without suffering any contention on data/instruction accesses. For tasks with large memory footprint that cannot entirely fit into a scratchpad partition, we suggest to split the task into several segments that fit as discussed in (Software techniques for scratchpad memory management 2015) and Li et al. (2005). Each segment will then leverage the same performance benefits as mentioned above. The main goal of this paper is to describe a full system that incorporates scratchpad memories (SPM) management and corresponding schedulability analysis.

The rest of the paper is organized as follows. Section 2 briefly reviews the related work. Next, Sect. 3 introduces the considered system model and architectural assumptions. The design of the proposed OS is described in Sect. 4, while Sect. 5 discusses how to conduct task schedulability analysis. We describe the performed implementation in Sect. 6, and discuss the experimental results in Sect. 7. Finally, the paper concludes in Sect. 8.

2 Related work

Temporal predictability is a crucial design-time constraint for real-time operating systems (RTOS). Several RTOS designs have been proposed, and a number of implementations are available, such as: QNX Neutrino,[1] FreeRTOS,[2] Wind River VxWorks,[3] These RTOS were designed for single-core platforms, where the use of real-time scheduling policies, efficient inter-process communication and prioritized interrupt handling were enough to ensure temporal predictability. Support for multi-core platforms was later introduced without a substantial change in design. Unfortunately, however, a new set of challenges (mainly related to shared hardware

[1] http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html.

[2] http://www.freertos.org/.

[3] http://www.windriver.com/products/vxworks/.

resource management Bui et al. 2011; Ungerer et al. 2010; Mancuso et al. 2015) is faced when trying to achieve predictability on multi-core systems.

In avionic standards such as ARINC 653 and ARINC 651, the concept of resource partitioning is central for the design of safety-critical systems. Even if different partitions execute on the same physical processor, the behavior/misbehavior of a software component should not affect the execution of another component running on a separate partition (Wilding et al. 1999). In single-core systems, requirement for inter-partition isolation can be achieved by employing time division and fault containment strategies. On multi-core systems, however, how to enforce and certify strong partitioning across different cores is still an active research topic.

In Jean et al. (2012), Jean et al. provide a high-level discussion of the main issues for the extension of existing avionic standards to multi-core systems. The work considers multi-core integrated modular avionic (IMA) systems were partitions may run in parallel on different cores. The authors raise the concern that in the presence of faults, the use of shared hardware resources may lead to a violation of strict inter-partition isolation requirements. In multi-core systems, interference channels (if not carefully mitigated) are also present under normal operating conditions. This has been acknowledged by certification authorities (FAA 2015) and it represents a source of concern for the use of multi-core processors in avionics systems. In this work, we propose a RTOS design that leverages co-scheduling techniques of shared resources to mitigate inter-core performance interference. Although we envision that some of the proposed design principles could be reused to enhance temporal protection in multi-core avionics systems, the proposed OS design rather targets embedded platforms suitable for automotive systems and its extension to IMA is currently out of the scope of this work.

The proposed layer of OS-level strategies to perform co-scheduling of shared resources is in line with the concept of Deterministic Platform Software (DPS) as defined in Girbal et al. (2015). Specifically, in our system we enforce a deterministic execution model for running applications, constructing a DPS that actively controls and schedules access to shared resources. Following the nomenclature proposed in Girbal et al. (2015), since tasks need to be specifically engineered and compiled to comply with our task model, the proposed solution is *application aware*. In this work, the multi-stage task model is also consistent with the Acquisition Execution Restitution (AER) task model proposed in Durrieu et al. (2014).

The AER model proposed in Durrieu et al. (2014) achieves predictability by executing tasks from local core memories (scratchpads), while shared memory resources are only used for inter-core communication and device I/O during acquisition and/or restitution phases. Inter-core interference arising from unregulated access to shared memory is mitigated by ensuring that: (i) the execution phase of different tasks can progress in parallel on multiple cores; and (ii) at most one acquisition or restitution phase is in execution at any instant of time. In Durrieu et al. (2014), the fundamental assumption is that the total footprint of all the tasks assigned to a core fits inside the core's local memory. In this work, we relax this constraint and only require that a task fits in half of the local memory space. This relaxation leads to important differences in the RTOS design: in fact, dynamic loading and unloading of tasks from/to local memories (together with I/O data) need to be handled. For these reasons, tasks' execution phases are parallelized; additionally, task loading/unloading is pipelined with

execution by using DMA engines. Finally, asynchronous I/O device activity is deconflicted from applications by exploiting hardware specialization at the bus level and by handling system-to-device interaction inside an isolated I/O subsystem.

Techniques to derive WCET bounds on a multi-core system accounting for the major sources of unpredictability have been thoroughly analyzed in Chattopadhyay et al. (2014). The latter work provides an in-depth overview of the state-of-the-art analysis methodologies for shared buses, shared caches as well as scratchpad memories. Its focus, however, is the derivation of safe WCET bounds in presence of typical platform features and given a known task set. However, there is no discussion on how a real-time OS can be designed on multi-core platforms to support multi-tasking subject to temporal constraints.

The design of multi-core architectures that are able to provide worst-case execution time guarantee have been proposed in Bui et al. (2011) and Ungerer et al. (2010). Specifically, the precision timed (PRET) architecture (Bui et al. 2011) introduces task runtime control and deadline enforcement at the instruction set architecture (ISA) level. Additional hardware modifications allow to achieve better performance without sacrificing predictability. Similarly, in the MERASA project (Ungerer et al. 2010; Wolf et al. 2010), predictability is achieved at hardware level by controlling inter-core interference. These works propose architectural features that have been prototyped on field-programmable gate array (FPGA), but unfortunately such features cannot be found in COTS system-on-chips (SoC).

In Mancuso et al. (2015), Yun et al. (2013), Mancuso et al. (2013), and Yun et al. (2014) the authors presented the Single-Core Equivalence framework: that is a set of OS-level techniques that can be implemented on COTS platforms to enforce spatial and temporal partitioning of shared memory resources. Derived analysis and experimental validation showed that WCET of tasks can be bounded and that inter-core temporal isolation can be achieved. Apart from the SCE there are other works from Lampka et al. (2014), Schranzhofer et al. (2010), Flodin et al. (2014), Lampka and Lackorzynski (2016) and Farshchi et al. (2018) that allow us to bound the worst case execution time of real-time tasks in a multi-core environment. In Lampka et al. (2014) and Schranzhofer et al. (2010), authors bound the non-determinism introduced in the execution of the real-time tasks in a multicore environment because of contention on main memory by proposing the use of time-structured tasks model and timed automata for shared bus (by considering different arbitration schemes such as time division multiple access (TDMA), round robin (RR) or a hybrid of the two). In order to bound the amount of interference from different cores on the shared memory controller, authors in Flodin et al. (2014) adapt a dynamic software-based memory throttling approach.

Similarly, in Farshchi et al. (2018) authors propose Deterministic Memory where the platform OS and hardware architecture guarantees strictly bounded worst-case access timing. Three main differences exist with respect to the proposed approach. First, the work in Mancuso et al. (2015), Lampka et al. (2014), and Flodin et al. (2014) assumes a traditional task execution model, while in this work this assumption is relaxed using a three-phase execution model. Second, in this paper we focus on scratchpad-based architectures. Finally, this work also proposes the design of a novel I/O subsystem.

Proposed work is a contribution to existing literature on the usage of SPM for real-time systems (Puau and Pais 2007; Metzlaff et al. 2011; Wasly and Pellizzoni 2013; Whitham et al. 2012; Whitham and Audsley 2012; Takase et al. 2010). In fact, a number of works have explored the benefits of scratchpad memories over traditional caches for multi-core platforms (Puau and Pais 2007; Metzlaff et al. 2011). Other works on embedded systems exist that propose scratchpad memory allocation strategies targeting real-time applications (Deverge and Puaut 2007; Lu et al. 2013; Bai et al. 2013; Suhendra et al. 2010; Falk and Kleinsorge 2009). In Takase et al. (2010) the authors propose a scratchpad memory management technique for preemptive multi-tasking systems where they introduce three methods for SPM partitioning that are: (i) spatial, (ii) temporal, and (iii) hybrid approaches. By employing these three methodologies on a real-time operating system the authors show that they were able to save 73% of energy when compared to the standard approach. The authors also conclude that hybrid approaches outperform the other two approaches. However, this work has not been applied to multi-core processors. Moreover, the focus of Takase et al. (2010) is not predictability but energy efficiency, making its contribution substantially different from proposed work.

Finally, our design shares some similarities with scratchpad scheduling approaches that have been proposed in Wasly and Pellizzoni (2013), Whitham et al. (2012), Whitham and Audsley (2012), and Wasly and Pellizzoni (2014). Compared to these works, our approach mainly differs in four aspects: (i) it is not focused exclusively on scratchpad management, but we rather show how a scratchpad can be integrated within an overall OS design; (ii) a full OS design is implemented on a commercially available (COTS) micro-controller; (iii) a predictable inter/intra-task communication scheme is provided and analyzed; and (iv) it is also discussed how I/O traffic issued by different cores is deconflicted. Part of this work without inter/intra core communication has been presented at Tabish et al. (2016). Table 2 compares our work with the relevant related work.

3 System model and assumptions

This section summarizes the task model that we use and the hardware assumptions we rely on for the design of proposed predictable operating system, namely SPM-centric OS.

3.1 Scratchpad memories

The first assumption we make is the presence of SPM. We assume that each core in our system features a block of private scratchpad memory. Moreover, in this work we assume that the size of each per-core scratchpad memory is big enough to fully contain the footprint of any two tasks in the system. Hence, the footprint of the sum of the two largest tasks in the system is lower than or equal to the size of the SPM. Although this assumption may appear restrictive, we make the following considerations. First, modern scratchpad-based micro-controllers provide scratchpad memories that have a size

Table 2 Comparison of current work with related work

Work	SPM/cache	I/O core	Single/ multicore	Inter/intra core comm.	Implementation COTS/FPGA/ GEM5	OS (Linux or RTOS)
PRET (Bui et al. 2011)	SPM	N	Single	N	FPGA	X
MERASA (Ungerer et al. 2010)	SPM/Cache	N	Multicore	N	FPGA	X
SPM-custom (Schranzhofer et al. 2010; Flodin et al. 2014; Lampka and Lackorzynski 2016)	SPM	N	Single	N	FPGA	X
SPM-Commodity (Farshchi et al. 2018)	SPM	N	Single	N	COTS	X
SCE (Mancuso et al. (2015); Chattopadhyay et al. (2014); Wolf et al. (2010); Yun et al. (2013))	Cache	Y	Multicore	N	COTS	Linux
Deterministic memory (Lampka et al. 2014)	Cache	N	Multicore	N	GEM5 (simulator)	Linux
This work	SPM	Y	Multicore	Y	COTS	RTOS

in the same order of magnitude as the main memory. For instance, in the MPC5777M that we use for our evaluation, each core includes 80 kB of scratchpad with a total main memory size of about 400 kB. Second, hard real-time control tasks typically are compact in terms of memory size. Third, if a task violates this size constraint, known methodologies exist (Software techniques for scratchpad memory management 2015, Li et al. (2005)) to split a large application into smaller sub-tasks that are individually compliant with the imposed constraint.

In our proposed design before tasks can be executed from SPM, their code and data need to be transferred from main memory. Thus, we adopt a task model that is composed of three phases: a *load phase*, an *execution phase* and an *unload phase*. First, during the load phase, the code and data image for the activated task is copied from main memory to the SPM. Next, during the execution phase, the loaded task executes on the CPU by relying on in-scratchpad data. Finally, the portion of data that has been modified and needs to remain persistent across subsequent activations[4] of

[4] We use the terms activation and arrival interchangeably to mean a task release.

the task is written back to main memory during the unload phase. More details are provided in Sect. 4.

It should be noted that compared to a traditional cache-based platform with locked instruction caches and temporarily locked data caches there are no notable differences in terms of predictability. However, in case of cache-based platforms one is forced to use the CPU to perform cache prefetching and locking, while the work proposed in this paper uses DMA-aided scratchpad re-configuration with task execution. Moreover, in many architectures the last-level cache (LLC) is shared among multiple cores and platform-specific locking and/or cache partitioning features may not be available, which is in line with the current trend in COTS embedded systems. For instance, support for cache locking has been removed in the transition from ARM Cortex-A9 to Cortex-A15. Conversely, the newly introduced Cortex-R family of chips feature core-local scratchpad memories–also known as tightly-coupled memories (TCM) with sizes up to 1 mB per core.

3.2 DMA engines

To avoid to stall the CPU when load-unload operations are performed, we assume that copy operations toward/from the scratchpad memories can proceed in parallel with task executions. This can be achieved as long as execution and load-unload phases belong to two distinct tasks. In order to parallelize load-unload operations with task execution, we rely on DMA engines. We assume that the hardware provides DMA engines that are able to transfer data from the main memory into the scratchpad and vice versa. By exploiting (i) the capability of parallelizing load-unload operations together with task execution, and (ii) the assumption that any task image can fit in half of the SPM, it is possible to hide task loading/unloading overhead during task execution, as we discuss in Sect. 4.

3.3 Dedicated I/O bus

The next made assumption is about the organization of the I/O subsystem. Since the activity of I/O devices is typically triggered by external events, it is inherently asynchronous. Unfortunately, unregulated I/O activity on the system bus can lead to unpredictable contention with CPU activity (Betti et al. 2013). Hence, unarbitrated I/O traffic represents one of the major sources of unpredictability in real-time systems. To deconflict the inherently asynchronous activity of I/O devices from application cores' activity, we assume that a dedicated bus exists to route I/O traffic without directly interfering with CPU-originated memory requests. The idea of co-scheduling CPU activity and I/O traffic is not new and specific solutions have been proposed in Betti et al. (2013) and Pellizzoni et al. (2011). However, the increased awareness of chip manufacturers about this problem has resulted in the design of COTS platforms that use dedicated buses to handle I/O transactions. Table 1 shows a non-exhaustive list of COTS platforms with this feature. In this work, we assume that suitable hardware exists to enforce a separation between I/O and CPU-originated memory transactions. Furthermore, traffic transmitted over the dedicated I/O bus needs to be handled, pre-

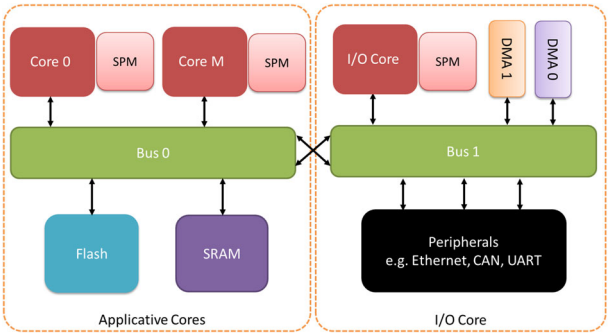

Fig. 1 Multicore architecture satisfying our hardware assumptions

processed and scheduled before reaching the application cores. Thus, we assume that an I/O processor exists, which we hereafter refer to as *I/O core*. Just like the application cores, the I/O core features a SPM that is used to buffer I/O data before they are delivered to applications.

Typically, devices that support high-bandwidth operations are DMA-capable. Instead, slower devices expose memory-mapped input/output buffers that can be read/written using generic platform DMA engines. Without loss of generality, we assume I/O data transfers from/to the I/O core are performed by DMA engines and that data from I/O devices can directly be transferred into the I/O core's SPM. In other words, I/O devices are not allowed to initiate asynchronous transfers directly towards main memory. As previously discussed, this design choice allows us to perform co-scheduling of CPU and I/O activities to achieve higher system predictability. A summary of the architectural assumptions discussed so far is provided in Fig. 1.

3.4 Memory organization

As micro-controllers evolve into complex multi-core systems, more advanced support of memory protection schemes is provided. However, for the purpose of this work, no specific assumption needs to be made about platform memory protection features. Hence, the presence of a memory management unit (MMU) is not a necessary requirement. We discuss in Sect. 6 how task relocation from main memory to scratchpads can be achieved without MMU support. Intuitively, MMU support allows for a straightforward implementation of task relocation by relying on page table manipulation. Usually, systems without MMU include a memory protection unit (MPU). MPUs support the definition of per-core access permissions based on linear ranges of physical memory addresses. Although they are not necessary to implement our system, MPUs can be easily supported within our design.

The hardware assumptions described so far represent desirable features that are becoming increasingly common in modern COTS micro-controllers used for safety-critical applications. Table 1 provides a list of some of the available COTS platforms that satisfy the described assumptions.

Table 3 Task's parameters

Term	Definition
τ_i	A task in the system
T_i	Task's MIT or period (if task is periodic)
C_i	Task's execution time including all overheads
σ	TDMA slot size for the DMA operation
R_i	Task's response time

3.5 Task model

For the proposed design, we consider a partitioned and fixed priority scheduling policy; additionally, each core has a set Γ of N sporadic tasks, $\{\tau_1, \ldots, \tau_N\}$, each with different priority whereby τ_1 has the highest priority and τ_N has the lowest priority. The deadline of each task is assumed to be less than or equal to its minimum inter arrival time. Table 3 summarizes the notation used for task parameters. As described in Sect. 3.1, the tasks follow a three-phases model. Hence, to satisfy temporal constraints, the last phase (unload) of a task needs to complete before the deadline. For ease of implementation, this work assumes non-preemptive tasks, although we plan to relax this assumption as part of our future work.

3.6 Communication model

We assume an asynchronous communication model that regulates data exchange between tasks, whether they run on the same core or on different cores. In this model, if the previously produced data by a producer task is not consumed by a consumer task, it will be overwritten once the producer generates new data. In details, a producer task's communication data is written to main memory during its unload phase. Similarly, any communication data required by a consumer task is loaded into SPM during its load phase. Therefore, this model does not require to directly move communication data between SPMs: instead, the communication is performed via main memory during the load and unload phases of the tasks. Furthermore, communicating tasks are not required to execute at the same rate. In order to compute the end-to-end latency of the communicating tasks, we assume that communicating tasks are periodically activated according to their Minimum Inter-arrival Time (MIT). For communicating tasks that have sporadic activations, the worst-case end-to-end latency is infinite. As such, in our end-to-end analysis we assume that communicating tasks have periodic activations instead. It should be noted that the proposed model is appropriate for communication schemes where data exchange/sharing occurs only at the boundaries of processing jobs. This is common for processing schemes that progress in stages. A typical example is edge detection, where an image undergoes the following processing stages: (1) smoothing, (2) enhancement, (3) thresholding, and (4) localization. To implement the semantics of edge detection in our system, consider the case where the four stages are implemented as four communicating tasks. For simplicity, let us assume that the four tasks have the same period. In our proposed example, smoothing and enhancement

tasks are assigned to one core. Whereas, the thresholding and localization tasks are assigned to a different core. This assignment allows us to double the throughput at steady-state compared to a single-core implementation. As described earlier, when a task is activated it is loaded into the SPM using a DMA. Each task has an input buffer and an output buffer. Upon activation, the input data is loaded from main memory into the SPM along with its local code/data. Upon completion, the completed tasks output data is written into the input buffer of the next task during the download phase of the task using DMA. In the case of the considered image processing pipeline, assume that at steady-state frame (sample) k of the video stream has been acquired. While smoothing is being performed on frame k, enhancing is performed on frame $k-1$, while in parallel thresholding is carried out on frame $k-2$, and localization on frame $k-3$. In other words, a given task stage is working on new data, while the next stage processes data from the input received in the previous sampling period. Our model is also applicable for processing workflows described according to a DAG (directed acyclic graph) task model, and where data exchange is allowed only at the boundaries of processing nodes. By contrast the communication model considered hereby is not suitable for processing schemes that exhibit tight data sharing between parallel and potentially co-running processes.

In the context of communicating tasks, we define as "flow" a sequence of data exchanges that occur between a terminating task instance (producer) after its unload, and the subsequent load of the instance of a consumer job. Since data exchange occurs in a "chain" of producer-consumer tasks, we often use the term "task chain" or simply "chain" when referring to a communication flow. This allows us to avoid the confusion with I/O flows intended as a sequence of I/O data belonging to the same I/O device, or set of devices.

We assume any number of pairs of sender/receiver tasks, which can be assigned either to the same or to different cores. We assume that all such pairs are statically defined at task configuration time. Note that a task can belong to more than one pair; in particular, it can both send and receive data to/from multiple other tasks. It is often the case that related data is exchanged along a sequence of tasks. For example, a sensing task might receive raw input data from an input device, process it, and pass it to a control task. In turn, the control task might compute the required control action and pass it to an actuating task, which prepares output data for an output device. Hence, we are interested in computing bounds on the end-to-end latency for communication across sequences of tasks. Formally, we consider a set of task chains, where each chain λ_k is a set of N_k communicating tasks $\{\tau_{k,1}, \ldots, \tau_{k,i}, \ldots, \tau_{k,N_k}\}$, with $\tau_{k,i} \in \Gamma$. Each task $\tau_{k,i}$ in λ_k with $1 < i < N_k$ receives data from the previous task $\tau_{k,i-1}$ and sends data to next task $\tau_{k,i+1}$ in the chain. Finally, for implementation reasons, we assume that task-local data, code, I/O buffers, and communication buffers can be accommodated within one partition of the scratchpad memory.

4 Proposed operating system design

In this section, we describe the design of the proposed SPM-centric OS by relying on the previously discussed assumptions.

4.1 Overview

The central idea of the proposed SPM-centric OS is resource specialization. As previously mentioned, a specialized I/O core and I/O bus are used to handle peripheral traffic. Similarly, a specific role is assigned to different memory resources in the system. Specifically, three types of memory resources exist in our system, as depicted in Fig. 1. First, flash memories are used to persistently store application/OS code, read-only data, as well as initialization values of read-write portions of main memory. Next, the SRAM (main) memory contains writable application and system data that represent the time-variant state of the system. Finally, scratchpad memories temporarily store a copy of code and data images for those tasks that are currently being scheduled for execution.

In our solution, applications are never executed directly from main memory, thus we adopt the following strategy: (1) task images are permanently stored in flash and loaded into main memory at system boot; (2) a dedicated DMA engine is used to move task images to/from SPM upon task activation; (3) a secondary DMA engine is used to perform I/O data transfers between devices and I/O core; (4) tasks always execute from SPM; (5) only task-relevant I/O data are transferred upon task load from the I/O subsystem. The benefit of this design is twofold. First, it allows high-level scheduling of accesses to main memory, ultimately achieving conflict-free execution of tasks from local memories. Second, performance benefits derived from the usage of fast scratchpad memories are exploited, ultimately combining better performance with higher temporal determinism.

We refer to the capability of our SPM-centric OS to dynamically move applicative tasks in and out of the SPM memories as *support for relocatable tasks*. As mentioned in Sect. 3, if hardware MMU support exists, task relocation can be achieved using page table manipulation. Otherwise, advanced compiler level techniques can be exploited to generate position independent code, as described in Sect. 6.

In proposed SPM-centric OS, a DMA engine is used to position the image of a relocatable task inside a SPM for execution. We refer to this DMA engine as *application DMA*. Similarly, we refer to the platform DMA used for I/O transfers as *peripheral DMA*. Typically, a single DMA engine is capable of utilizing the full main memory bandwidth in micro-controller platforms. Nonetheless, the design constraint that imposes the use of a single applicative DMA can be relaxed if the main memory subsystem allows two or more DMA engines to transfer data concurrently without saturating the main memory bandwidth.

4.2 Scratchpad and CPU co-scheduling

Load-unload operations for tasks running on the M applicative cores need to be serialized to prevent unregulated contention over the memory bus. Hence, only a single DMA is required as application DMA for all the M applicative cores. Several schemes are known to fairly share a single resource across different tasks. For the scope of our design, we employ a TDMA scheme to serialize task load-unload operations among

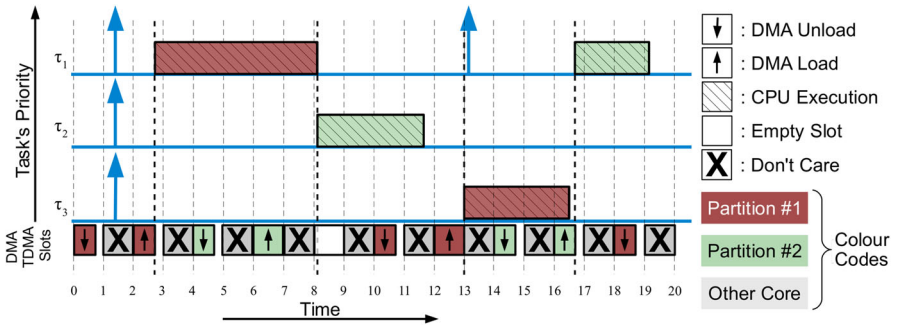

Fig. 2 Scheduling CPU, DMA and local memory

M applicative cores. The main advantage of the TDMA scheme lies in its simplicity of implementation.

In order to perform TDMA-based scheduling of the application DMA, time is partitioned into slots of fixed size. In each slot, only a single DMA operation can be performed, either a task load or unload. The slot size is chosen to ensure that the task with the largest footprint in the system can be loaded within the slot time window. Figure 2 depicts the sequence of operations in our TDMA scheme for a system with $M = 2$ application CPUs. Note that the TDMA enforcement needs to be centralized. Hence, in our design, the I/O core is responsible for interfacing with application cores' schedulers through active/ready queues, programming the application DMA as well as enforcing the time-triggered TDMA slots. In particular, Fig. 2 depicts three tasks scheduled on one core. Up arrows in blue color represent the arrival times of the considered tasks; we use colors for two different partitions. A task can only run after its load operation has been completed and the previous task on the other partition has completed, (see τ_2 to τ_3 and τ_1 to τ_2 for example of the two cases). There might be slots where no load-unload is performed. This happens at time 8: τ_1 finishes right after the beginning of the slot, so both partitions are full at the beginning of the slot and the I/O core can neither load nor unload any applicative core scratchpad. Effectively, the slot is wasted.

Since tasks need to be loaded/unloaded in parallel with respect to CPU activity, two partitions are created on the scratchpad. There is logically no difference between the two scratchpad partitions. Thereby, tasks may execute from either one of the two, depending on their arrival time. Interchangeably, one of them contains the image of the task which is currently being executed, while the second half is used to load (unload) the image of the next (previous) task to be executed (that was completed). Note that when a task is executing on the CPU while a second task is loaded-unloaded in background, CPU and DMA contend for scratchpad access. However, the impact of this contention on the timing of the tasks is typically negligible for two main reasons. First, scratchpads are often implemented as dual-ported memories; thus, they are able to support stall-free CPU and DMA operations. In fact, on the considered MPC5777M platform we have verified this by experimentation and found that both the core and the DMA module do not suffer any delay when they access the SPM simultaneously.

Second, in a system with M CPUs, DMA-CPU contention over scratchpad involves only two masters, as opposed to the traditional approach where up to M masters could contend for main memory.

As depicted in Fig. 2, the application DMA is alternatively assigned to transfer data for a specific core. Within a single slot, either an unload operation for a previously running task or a load operation for the next scheduled task is performed. The specific operation to be performed is decided as follows:

> Rule 1: If a load operation can be performed, a load operation is programmed on the application DMA;
> Rule 2: If a load cannot be performed and there is a previously running task to be unloaded, an unload operation is programmed on the application DMA.

Note that Rule 1 can be activated by the following conditions: (i) at least one of the two SPM partitions is available (i.e. has been previously unloaded), and (ii) a task has been released and is ready to be loaded. Similarly, Rule 2 can be activated if no load can be performed, at least one partition is not empty and the task loaded on that partition has completed.

In the proposed design, the next task to be executed is loaded in background while the foreground running task is not interrupted until its completion. The described mechanism allows to hide the DMA loading overhead, avoiding contention in main memory and exploiting performance benefits deriving from SPM usage.

The workflow followed by an applicative core and the I/O core at the boundary of each TDMA slot is depicted in Fig. 3. Specifically, at each time slot, the I/O core checks the status of the queue of active tasks belonging to the considered core. If a task that is active for execution but not ready (i.e. not relocated in scratchpad) is found, the I/O core checks which SPM partition (P1 or P2) is empty on the application core. If any partition is found to be empty (Slot #1), the I/O core programs the application DMA to load the topmost active task to the empty partition. Once the load is complete, the I/O core updates the active and ready queues of the considered application core. The latter operation allows the application core to begin the execution of the task (Slot #2). Note that since only one task can be in running state on the CPU, there is always a SPM partition that is available for load-unload operations.

4.3 I/O subsystem design

Together with memory resources, applications typically need to communicate with peripherals and thus require I/O data to operate. We propose an I/O subsystem design that enforces a complete separation between task execution and the asynchronous activity of I/O peripherals: this goal is achieved by offering to application tasks a synchronous view of I/O data. It is achieved by distinguishing between data production and their dispatch to/from tasks. In fact, we allow I/O data to flow from/to I/O subsystem to tasks only at the boundary of load-unload operations.

As mentioned in Sect. 3, we assume that a dedicated bus connects the SPM of I/O core with peripherals. Hence, asynchronous peripheral traffic can reach the I/O subsystem without interfering with task execution. For each device used in the proposed system, the OS defines a statically positioned *device buffer* on the I/O core

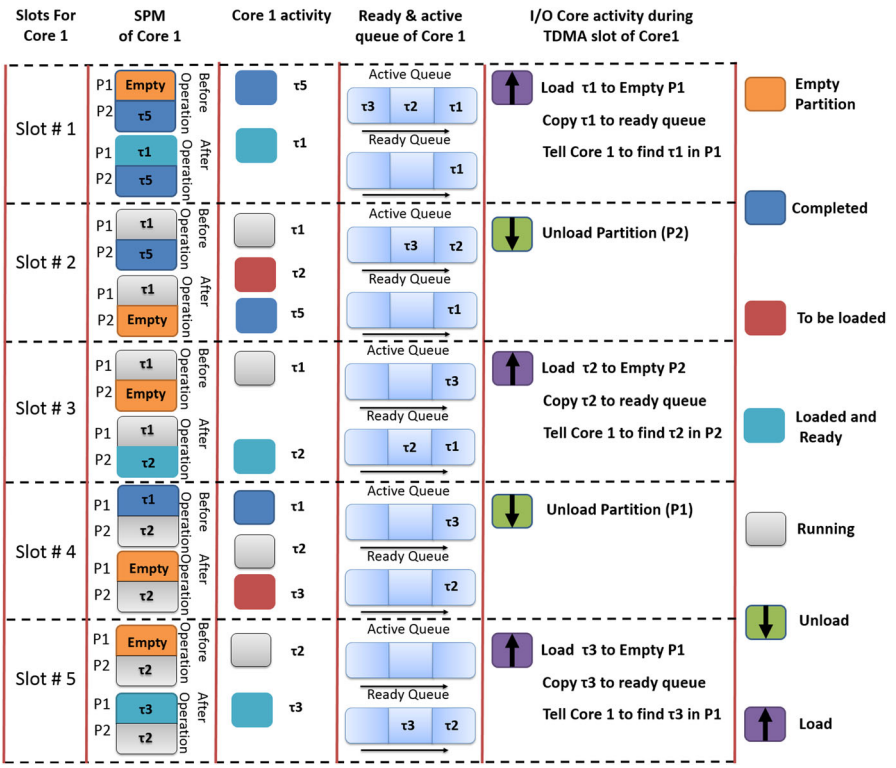

Fig. 3 Interaction between I/O core and core 1 for task scheduling

scratchpad. A device buffer is further divided into an *input device buffer* and an *output device buffer*. The input and output device buffers represent the positions in memory where data produced by devices and tasks (respectively) is accumulated before being dispatched to tasks or devices.

In our design, peripheral drivers can operate with an interrupt-driven or polling mechanism. For DMA-capable peripherals supporting interrupt-driven interaction, the driver only needs to specify the address in SPM of the device buffer from/to where data are transferred. The driver is also responsible for updating device-specific buffer pointers to prevent a subsequent data event from overwriting unprocessed data. For interrupt-driven interaction with non-DMA-capable devices, the driver uses the platform peripheral DMA to perform data movement. Similarly, the device driver is periodically activated and the peripheral DMA is used to perform data transfer for polling-based interaction with devices.

In general, device-originated interrupts as well as timer interrupts for device driver activations are prioritized according to how critical is the interaction with the considered device. Nonetheless, all the device-related events are served with priority levels that are lower than task-scheduling events, such as: (i) TDMA slot timer events and (ii) completion of application DMA loads-unloads.

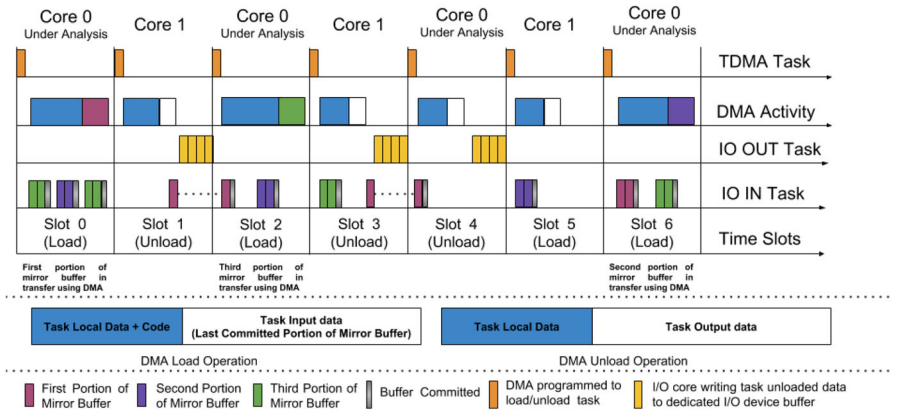

Fig. 4 Timeline of tasks performed on I/O core

In order to interface with a peripheral, application tasks define subscriptions to I/O flows. A subscription represents an association between a task and a stream of data at the I/O device. For instance, a given task could subscribe for all the packets arriving at a network interface with a specific source address prefix. Task subscriptions are metadata that are stored within the task descriptor.

For each task in the system, a pair of buffers (for input and output respectively) is defined on the SPM of the I/O core to temporarily store data belonging to subscribed streams. Since the content of these buffers will be copied to/from the application cores upon task load-unload, we refer to them as *task mirror buffers*. The concept of mirror buffers is similar to what are known as *bounce buffers* in the context of multiple buffering. Consider the arrival of I/O data from a device. As soon as the interaction with the driver is completed, the arrived data is present in the corresponding device buffer. According to task subscriptions, the OS is responsible for copying the input data to all the mirror buffers of those tasks subscribed to the flow.

The advantage of defining mirror buffers lies in the fact that when a task needs to be loaded, all the peripheral data that need to be provided are clustered in a single memory range. Consequently, during the loading phase of a task, the application DMA is programmed to copy the content of the mirror input buffer together with task code and data images to the application core. The reverse path is followed by task-produced output data during the task unload phase.

Since I/O data are delivered to applicative tasks at the boundary of load-unload operations, the approach presented in Sect. 5 for the calculation of tasks' response time can be reused to reason about end-to-end delay of I/O-related events.

Figure 4 provides an example of timeline of the tasks performed on the I/O core. These include TDMA slots management, I/O output dispatching tasks (IO OUT) and I/O input dispatching tasks (IO IN). The TDMA management task has the highest priority among the three types of tasks so that a application core must not stay idle and its utilization can be maximized. IO OUT tasks have intermediate priority level because as soon as the DMA unload operation finishes moving data to the task output buffer in the SPM of I/O core, the data needs to be copied to the output device buffer

of the device this is because the system must be able to send actuation at a certain rate. Where IO IN tasks have lowest priority because we assume asynchronous model where we use old data if new input has not arrive and new data every time if input rate matches or exceeds the task activation rate. In case the input rate matches the task activation the system will not miss any packets. Whereas in the case where the input rate exceeds the task activation rate the system might miss some packets but this is fine since we have asynchronous communication model.

TDMA tasks are activated in a strictly periodic manner, where the periodicity corresponds to the TDMA slot length σ. IO OUT tasks are triggered only following a task unload operation. These tasks are responsible for moving I/O data produced by a completed job from the corresponding task mirror output buffer and into the output buffers of the affected device(s). Since they are activated only after a completed task unload operation, they are inherently synchronous with respect to unload operations. Conversely, IO IN tasks are responsible for moving data arrived in input at some device into the mirror input buffers of application tasks according to I/O subscriptions. As such, their activation is directly related to data input events occurring at the various devices. The system can be comprised of a number of different devices. As such, our system uses device-specific drivers to handle the first part (top-half) of input event handling. Next, the same IO IN data dispatching task is activated as the bottom-half handling routine. It can never happen that an IO IN task instance preempts an already running IO IN job. In other words, I/O dispatching operations are always carried out in a serialized manner.

Top-half buffer handling while the top-half I/O input handling routines are device-specific, the following strategy is used for DMA-capable devices. A set of buffers—at least two, say A and B—are maintained for each device. At any point in time, a data pointer register is exposed by the device to indicate the location where the next input should be transferred, e.g. buffer A. Upon receiving an interrupt that indicates the availability of new data, the top-half routine reads such a register to determine the location of the received input. Next it updates the pointer to an unused buffer, e.g. buffer B, and passes the location of buffer A to the dispatch (IO IN) bottom-half task. In case of a two-buffers setup, it is fundamental that no new input arrives before any pending dispatch operation for a given device is completed. Additional device buffers can be introduced to prevent data loss when this property may not hold.

Bottom-half (IO IN) buffer handling an IO IN dispatching task is activated in response to new data arrivals at any of the supported devices. The first operation performed by the IO IN task consists in the interpretation of data headers to determine which application task(s), say Task 1 and Task 2, should be receiving the input data, depending on I/O subscriptions. The IO IN task strips any unnecessary encapsulation data from the received input and proceeds with dispatching to tasks. In a nutshell, the dispatch operation consists of a number of copy-commit operations—one per each I/O subscribed task. First, the unencapsulated device input data is copied into the mirror input buffer of Task 1. This copy operation is performed by the CPU (busy-copy). Once the copy operation is completed, the IO IN task commits the written task mirror buffer. This operation is atomic. The task input mirror buffer is divided in three slots of equal size, which is larger than or equal to the size of the largest input that can be received on the corresponding device. At a generic point in time, one of the slots

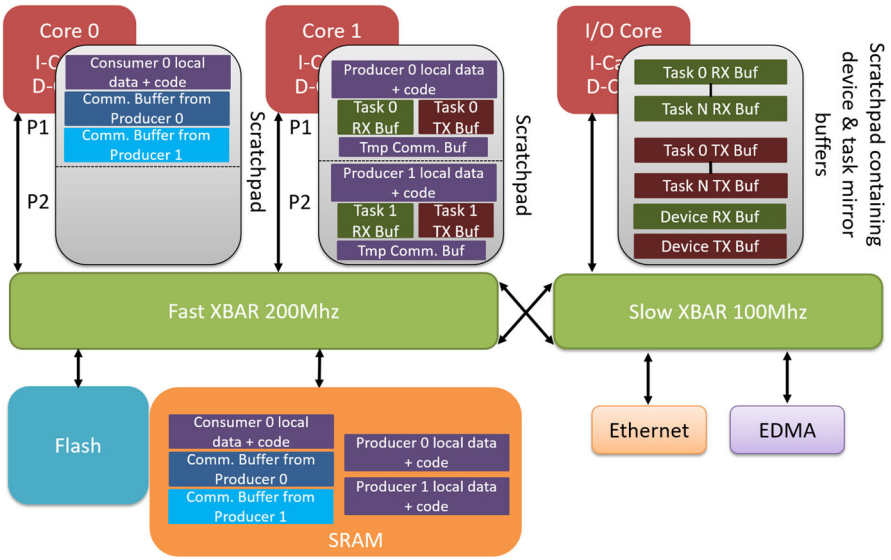

Fig. 5 Example showing I/O and intercore communication

holds the latest-committed input; a second slots holds the input being currently loaded by a TDMA load operation; while the third slot is used to direct the next dispatch operation. After dispatch for Task 1 is completed, the IO IN task proceeds to perform dispatching for Task 2 and so on. The roles of the three slots in a task mirror input buffer are better illustrated in Fig. 4.

4.4 Inter-core and intra-core communication

As mentioned earlier (see Sect. 3.6), for both inter-core (communication between tasks on different core) and intra-core (communication between tasks on same core) communication, the communication flow of how a task sends data to another task is statically defined at configuration time. Consider, for example, two producer tasks that need to send data to a consumer task. We name the two producer tasks as Producer 0 and Producer 1. Whereas, the task that will be receiving data from these tasks is named as Consumer 0. The scenario is depicted in Fig. 5.

At task definition, Producer 1 and Producer 2 need to know the size of each message sent to Consumer 0. This size information is used to create a temporary send buffer inside SPM of the application core where Producer 0 and Producer 1 tasks will be loaded and executed. These temporary buffers are used to store communication data that Producer 0/1's intends to send to the consumer task.

Similarly, Consumer 0 at configuration time needs to define separate receive buffers in main memory for each producer task (two in our example) from which it intends to receive data. In our example, two buffers are defined: for Producer 0's and for Producer 1's data.

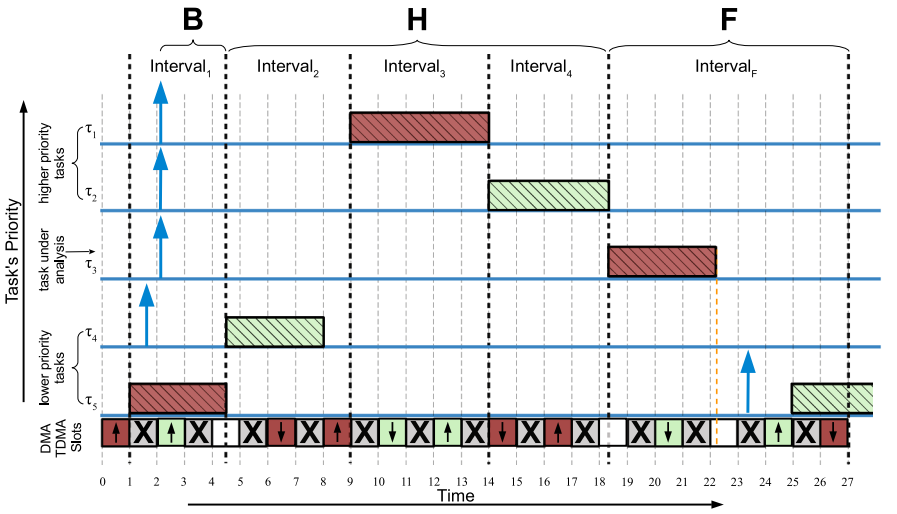

Fig. 6 SPM-centric OS task scheduling. Scheduling intervals are highlighted

During execution of producer tasks in SPM, data to be sent to consumer tasks is written inside temporary buffers. These buffers are unloaded using a DMA from scratchpad directly to main memory, and specifically into the corresponding buffers of consumer tasks. Finally, when Consumer 0 task is activated, all of its local code, data, I/O data, and communication data, received by the time activation occurs, is loaded from main memory (and I/O scratchpad) into the corresponding application core's SPM.

5 Schedulability analysis

Given the scheduling strategy described in Sect. 4, we can calculate a safe bound on the worst case execution time based on all tasks' parameters in an approach similar to Wasly and Pellizzoni (2014). Note that we assume that the task's execution time, C_i, is actually the adjusted execution time in which all the overheads are included, such as the context-switch and the DMA setup routines. Also note that for simplicity we discuss the case with $M = 2$ cores, since it is used in our prototype, but the analysis could be trivially extended to account for any number of cores by changing the length of the DMA operations.

Figure 6 depicts an illustrative example of the worst case scheduling scenario (critical instant) for an example task set where τ_3 is the task under analysis. The schedule depicts a busy period where τ_3 suffers interference from two higher-priority tasks, τ_1 and τ_2. As in Wasly and Pellizzoni (2014), we consider the busy period as composed by a sequence of *scheduling intervals* $Interval_1$, $Interval_2$, $Interval_3$, and $Interval_4$ (each bounded by bold vertical lines in the figure), followed by a *final interval* $Interval_F$ (F). During each scheduling interval, only one blocking or interfering task runs. During the final interval, the task under analysis runs. Each scheduling

interval always starts with a CPU execution and ends either when the CPU finishes executing the task or when the next task finishes being loaded by the DMA, whichever happen last; at this point, the next interval starts with the execution of the loaded task. The final interval starts with the execution of the task under analysis and finishes when the task under analysis is unloaded.

We say that a scheduling interval is *CPU-bound* when it ends with CPU execution (ex: $Interval_1$, $Interval_3$ and $Interval_4$ in the figure), and *DMA-bound* when it ends with DMA load operation (ex: $Interval_2$). The length of a scheduling interval is the maximum between the execution time of the task running in the interval and the DMA operations required to load the next task. We denote the size of the TDMA slot as σ; since in the worst case a load-unload operation can occupy the entire slot, we upper bound the length of DMA operations as a multiple of σ.

5.1 Response time calculation

Similar to Wasly and Pellizzoni (2014), we compute the response time R_i of the task under analysis using the standard response time iterations as follows:

$$R_i \leftarrow B + H(R_i) + F \tag{1}$$

In detail, we compute the response time by adding three components: (1) the blocking time B caused by a lower priority task that starts executing before the beginning of the busy period; this is $Interval_1$ executing task τ_5 in the figure; (2) the interference H comprising the remaining scheduling intervals in the busy period, which are $Interval_2$, $Interval_3$ and $Interval_4$ in the figure. The number of such intervals is equal to the number of interfering higher priority jobs plus one, since an extra lower priority job that starts loading before the beginning of the busy period (τ_4 in the figure) can execute within the busy period itself; (3) the length F of last interval $Interval_F$ in which the task under analysis executes and finishes.

Compared to Wasly and Pellizzoni (2014), our solution differs in three aspects. First, in this work we use fixed-size DMA operations, while Wasly and Pellizzoni (2014) employs dynamic-size DMA operations. Therefore, we need to discuss how to compute the length of the DMA operations as multiple of TDMA slots (σ). Second, we need to recompute the length of the blocking time B since the next task to be loaded is determined at a different time. Finally, unlike Wasly and Pellizzoni (2014), we consider the task under analysis finished when the task is unloaded, at the end of $Interval_F$ instead of finished execution on the CPU. We address each point in sequence.

5.2 Critical instant and blocking time (B)

Unlike the traditional non-preemptive scheduling where a task under analysis τ_i can at most suffer bocking from one lower-priority task, in our scheduling scheme τ_i can at most suffer blocking from two lower-priority tasks due to the number of SPM partitions available to load tasks into. For instance, at the beginning of the example schedule

in Fig. 6, the system has two free local SPM partitions at time zero. In $Interval_1$, the task under analysis τ_3 is released along with all higher-priority tasks after an arbitrarily small time (ε) when all free partitions have been loaded or have started loading lower-priority tasks (τ_5 and τ_4); this is ε after time 2 in the figure. Since we consider non-preemptive DMA operations and non-preemptive CPU execution, the task under analysis τ_3 cannot run until the pre-loaded lower-priority tasks (τ_5 and τ_4) plus all higher-priority tasks (τ_1 and τ_2) finish execution. We now prove that the discussed scenario is indeed the critical instant for our system, leading to the worst case response time for the task under analysis.

Lemma 1 *The critical instant is produced when the task under analysis τ_i and all higher priority tasks arrive immediately after a lower priority task has started loading into a partition, and the other partition was loaded with another lower priority task as late as possible* (*i.e., two slots before*).

Proof We first show that in the worst case, both τ_i and all higher priority tasks must arrive ε time after the beginning of a slot where a lower priority task is being loaded. If either τ_i or a higher priority task would arrive at or before the beginning of the slot, then such task would be loaded and executed in place of the lower priority task. Hence, the length of the busy period would decrease by one scheduling interval, which cannot produce the worst case response time for τ_i.

If instead τ_i arrives some δ time later during the busy period, then the finishing time of τ_i would not change, but the response time of τ_i would decrease by δ. Similarly, if a higher priority task arrives later during the busy period, the number of interfering jobs of the task could only be lower or equal compared to releasing it immediately after the beginning of the slot. Hence, the described activation pattern must lead to the critical instant.

For what concerns the lower priority task pre-loaded in the other partition, it suffices to notice that loading the task as late as possible maximizes the amount of execution of the task within the busy period. Assuming that the first partition was initially free, so that no unload was required, and given the TDMA arbitration of DMA operations, the latest such load could happen is two slots before τ_i arrives, i.e., during slot [0 : 1] in Fig. 6. □

Based on Lemma 1, the task under analysis τ_i arrives $\sigma + \epsilon$ time after the beginning of $Interval_1$; hence, we can upper bound the blocking time as the length of $Interval_1$ minus σ. Furthermore, the length of $Interval_1$ itself is bounded by $\max(C_5, 2 \cdot \sigma)$; here, C_5 represents any low priority task executed in $Interval_1$, while the length $2 \cdot \sigma$ accounts for the fact that the next task is loaded after $2 \cdot \sigma$ in the second slot of the interval (slot [2:3] in the figure). As noted in the proof of Lemma 1, this is possible as the partition is free and no unload is required during this interval. Similar to Wasly and Pellizzoni (2014), since we can make no assumption on which lower priority tasks execute in $Interval_1$ and $Interval_2$, we simply consider the two lower-priority tasks τ_{l1} and τ_{l2} with the longest execution time and the second longest execution time respectively. Thus, we define the blocking B as follows:

$$B = \max(C_{l1}, \ 2 \cdot \sigma) - \sigma. \tag{2}$$

5.3 Scheduling intervals in the busy period (H)

When the system is busy with both SPM partitions occupied and at least one active task, within each interval we need to first unload the previous partition and then load it with the next task. Therefore, for any scheduling interval, it will require four TDMA slots ($4 \cdot \sigma$) to load the next task if the interval was preceded by another DMA-bound interval, such as for $Interval_3$ in Fig. 6. On the other hand, if the interval is preceded by a CPU-bound interval, it might require up to five TDMA slots ($5 \cdot \sigma$) to finish loading the next task in the worst case, as for $Interval_2$. This is because the CPU-bound interval might finish at the middle of a slot or ε time after the beginning of a slot, hence, inducing an unused (empty) TDMA slot in the next interval (slot [4:5] in the figure).

As a result, the length of any scheduling interval can be computed as either $\max(C_i, 4 \cdot \sigma)$ or $\max(C_i, 5 \cdot \sigma)$. We now formally prove that for any CPU-bound interval to cause the worst case scenario, with the exception of the first interval $Interval_1$, the CPU execution has to be strictly longer than four TDMA slots ($4 \cdot \sigma$). Note that in $Interval_1$, unlike the busy period intervals, there is no DMA unload operation, as the targeted partition is already free.

Lemma 2 *For any scheduling interval in H, no extra empty slot will be induced in the next interval unless the length of the CPU execution is strictly greater than four TDMA slots* ($4 \cdot \sigma$).

Proof We show that any scheduling interval in H with CPU execution less than $4 \cdot \sigma$ cannot induce an empty slot in the next interval. First, by definition, a DMA-bound interval finishes when the load phase of a task is done. Consequently, in the next interval, the CPU execution of the loaded task can start immediately, and the unload of the other partition can start in next TDMA slot dedicated for the same core. Therefore, the DMA-bound interval cannot induce any empty TDMA slot in the next interval by definition.

Second, since in the busy period H both partitions are loaded with tasks, in each interval an unload and a load operations must be performed to load the next task. In the best case, the DMA unload starts at the beginning of the interval, as shown above in the Fig. 7. Thus, the next task is guaranteed to be loaded in $3 \cdot \sigma$ time after the interval started. As a result, any interval with CPU execution less than or equal to $3 \cdot \sigma$, must be a DMA-bound interval and cannot induce an empty TDMA slot in the next interval. However, as shown in the figure above, since the next TDMA slot is dedicated for the other core, it follows that the CPU execution need to be strictly greater than $4 \cdot \sigma$ to miss its own TDMA slot and induce an empty slot. □

In Algorithm 1, we show how to compute an upper bound on the length H during the busy period. Note that we use a python-like syntax for operations on lists: given

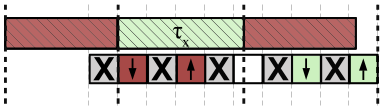

Fig. 7 Example of a CPU-bound interval that affects the next interval and increases the time required to load the next task

lists A and B, A.extend(B) adds the elements in B to A; $A + B$ is the concatenation of A and B; $A * n$ repeats A n times; and len(A) returns the number of elements in A. Based on Lemma 2, we construct a list of DMA times (M) used by the algorithm as follows: we insert in the list a number of $5 \cdot \sigma$ time values equal to the number of tasks executed in H with length greater than $4 \cdot \sigma$, plus one task (to account for the task in $Interval_1$, which can cause an extra empty TDMA slot as shown in Fig. 6). The remaining DMA times in the list are equal to $4 \cdot \sigma$. For this sake, Eq. 3, determines the length of TDMA slots inserted in the DMA list based on the execution time of the task in the previous interval.

$$DMA_i^{next} = \begin{cases} 5 \cdot \sigma & \text{if } C_i > 4 \cdot \sigma \\ 4 \cdot \sigma & \text{otherwise;} \end{cases} \tag{3}$$

Algorithm 1 Calculating a safe upper bound of the length of the busy period H of the task under analysis τ_i

1: $M = [5 \cdot \sigma, DMA_{l2}^{next}]$
2: $E = [C_{l2}]$
3: **for** $\tau_j \in hp(i)$ **do**
4: $\quad M$.extend$\left([DMA_j^{next}] * \left\lceil \frac{R_i - F - \sigma}{T_j} \right\rceil\right)$
5: $\quad E$.extend$\left([C_j] * \left\lceil \frac{R_i - F - \sigma}{T_j} \right\rceil\right)$
6: **end for**
7: $A = M + E$
8: sort A in descending order
9: H = sum(A[0 : len(E) - 1])

Step (1) in the algorithm initializes the list of DMA operations (M) with the lengths of τ_{l1} and τ_{l2}. Note that, since τ_{l1} runs in $Interval_1$ while not all partitions are occupied, it is assumed to induce $5 \cdot \sigma$ slots in the next interval. Similarly, Step (2) initializes the list of CPU executions (E) with C_{l2}, as τ_{l2} runs within the busy period H. Then in Steps (3)–(6), the DMA and CPU lists are extended based on the set of higher-priority jobs that can execute within the response window of task under analysis τ_i. Since we consider a non-preemptive scheduling, we reduce the interfering window to $R_i - F - \sigma$. This is because once the task under analysis starts loading (which happens no later than σ before the beginning of $Interval_F$), it is guaranteed to run non-preemptively in the next interval ($Interval_F$), hence does not suffer further interference from higher-priority jobs. In Step (7), the lists M and E are combined into A, which is then sorted descendingly. Finally, in Step (9), the top contributing DMA operations or CPU executions are selected and summed to upper bound the length of H. Figure 8 shows a visual illustration of the algorithm.

Lemma 3 *Assume that the task under analysis τ_i suffers interference from a set J_{hp} of higher priority jobs, plus one job of lower priority task τ_{l2}. Then the value of H computed by Algorithm 1 is a safe upper bound to the length of all intervals where jobs of either higher priority tasks or τ_{l2} delay the task under analysis.*

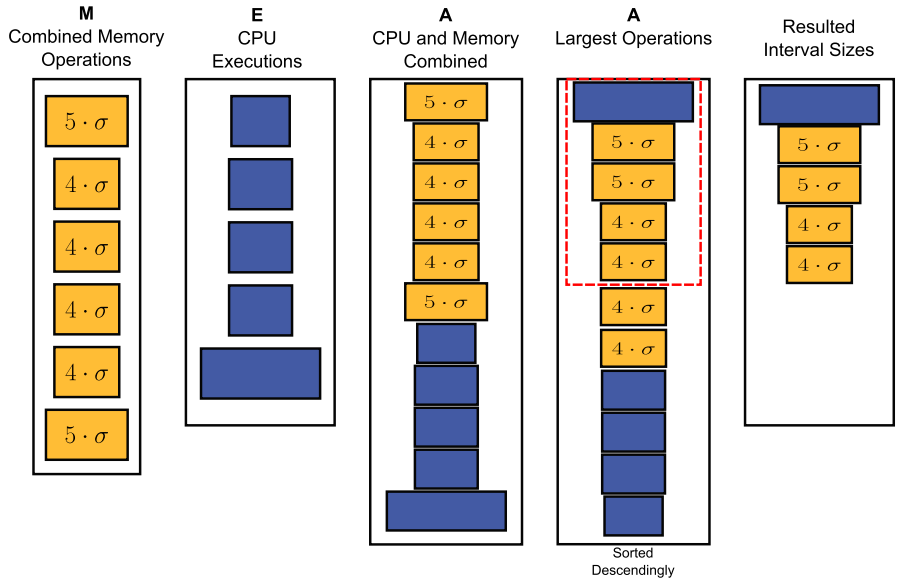

Fig. 8 An arbitrary example to illustrate how Algorithm 1 maximizes the intervals' sizes in the busy period H

Proof As discussed above, the maximum number of intervals where higher priority jobs interfere with the task under analysis τ_i is $I = len(J_{hp}) + 1$: one interval where the first higher priority jobs is loaded, plus $len(J_{hp})$ intervals to execute each higher priority job. In each interval there is one CPU execution that overlaps with DMA operations; therefore, the length of each interval is the maximum of the two. We can then obtain H as the sum of the lengths of I intervals, where each length is either a CPU execution time or a DMA operation of length $5 \cdot \sigma$ or $4 \cdot \sigma$ as proved in Lemma 2. Now, our algorithm simply takes all CPU execution times and all DMA times in those intervals, as shown above, and picks the maximum I elements among all of them; hence, this must result in a upper bound to the actual combined length of the I intervals. □

5.4 Final interval (F)

The length of the final interval $Interval_F\ (F)$ can be computed as $\max(C_i + 5 \cdot \sigma, 7 \cdot \sigma)$, where τ_i is the task under analysis. In the example depicted in Fig. 6, the length of $Interval_F$ is $C_3 + 5 \cdot \sigma$. The other case can happen when C_3 is short enough and slot [22:33] is utilized, in the worst case, to load the next task. In this situation, up to seven TDMA slots are required to finish unloading τ_3, as formally proven below.

Lemma 4 *The length of* $Interval_F$ *is upper bounded by* $\max(C_i + 5 \cdot \sigma, 7 \cdot \sigma)$.

Proof Similar to scheduling intervals in H, we need to consider two cases: (1) the length of the interval is bounded by C_i plus the time required to unload the task; (2) the length of the interval is bounded by the time required to unload/load the other

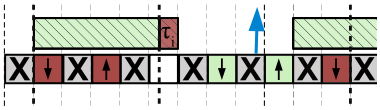

Fig. 9 Worst case when execution of task under analysis is small

partition before unloading C_i. For the first case, note that differently from scheduling intervals in H, there might be no active task at the start of $Interval_F$, since the last task in the busy period (task under analysis) is running. Therefore, a new job of any task could be released later during $Interval_F$ and start a load operation. In particular, the new job could arrive just before the unload of the task under analysis (τ_i), as shown in Fig. 6. In this case, since there is one free partition and load operations have priority over unload operations (Rules 1,2), the new job has to be loaded first; thus, the unload of τ_i is delayed by up to $5 \cdot \sigma$ in the worst case (one empty slot plus four other slots, as shown in Fig. 6 for slots [22:27]; no more than 5 slots are possible since after the load at [24:25], both partitions are full and thus an unload must happen next). In this case the length of $Interval_F$ is upper bounded by $C_i + 5 \cdot \sigma$.

On the other hand, Fig. 9 shown above illustrates the other case where the length of $Interval_F$ is $7 \cdot \sigma$ in the worst case. This case happens when the execution of the task under analysis is very small to the point that $C_i + 5 \cdot \sigma$ is smaller than the required number of TDMA slots to actually unload τ_i. When CPU execution of τ_i is sufficiently small, the load of the next task has to be after $5 \cdot \sigma$ at most regardless of the release time of the next task, otherwise τ_i would be unloaded by the fifth slot. If the next task is indeed loaded before the unload of τ_i as shown in the figure, then in the worst case it takes two more slots to unload τ_i (given that both partitions are full after loading the next task), hence resulting in a bound of $7 \cdot \sigma$. To conclude, by taking the maximum of the two cases we guarantee to capture the worst case. □

Theorem 1 *The worst-case response time of the task under analysis (R_i) is computed by Eq. 1.*

Proof We show that if Eq. 1 converges to a fixed-point R_i, then R_i is a valid upper bound to the response time of the task under analysis τ_i. Let τ_{l1} and τ_{l2} be the first and second lower priority tasks loaded before the critical instant, as required in Lemma 1. Then Eq. 2 computes an upper bound to B. Furthermore, Lemma 4 computes an upper bound to F. Finally, since according to Lemma 1, higher priority tasks arrive at the same time as the task under analysis, which cannot be preempted after it starts loading no later than one slot before the beginning of $Interval_F$, it follows that the interfering window for higher priority tasks has a length of $R_i - F - \sigma$; hence, all interfering higher priority jobs must be contained in the set J_{hp} computed at Line 1 of Algorithm 1, and as a consequence, according to Lemma 3 the value of H computed by the algorithm is a valid upper bound. In summary, since the response time is decomposed in three intervals, and the values computed for the three intervals are valid upper bounds, it follows that R_i is indeed an upper bound to the response time for a given choice of τ_{l1} and τ_{l2}.

Since we are interested in computing the worst case response time for any possible arrival pattern of lower priority tasks, it remains to determine how to select τ_{l1} and

τ_{l2} so as to maximize the response time of τ_i. Note that since τ_{l2} is loaded while τ_{l1} executes, they must be different tasks. Hence, it suffices to consider two cases: (1) τ_{l1} is the lower priority task with the longest execution time, and τ_{l2} is the second longest; (2) or vice-versa. Now note that since Algorithm 1 only selects the I largest times among all DMA operations, and CPU executions of higher-priority tasks plus C_{l2}, it follows that C_{l2} might not contribute to the worst case response time as it might get hidden by larger times. Therefore, if follows that case (1) leads to the maximum total response time of the task under analysis. □

5.5 Bounding communication latency

As mentioned in Sect. 4.4, tasks communicate asynchronously without any precedence constraint between them. In other words, when a job of a higher-priority task τ_1 is ready, it will be scheduled and start loading as soon as there is a free partition regardless of any other running tasks that send data to it. Since the access to main memory is serialized using DMA, integrity of the communication data is assured as there will be no data race (lock-less data sharing). Suppose τ_1 is a receiver task for data sent by τ_2. Then τ_1 will access the previous (old) communication data from τ_2 if τ_1 is loaded while τ_2 is still running or not yet unloaded from the SPM partition to main memory.

From a schedulability point of view, this communication model does not affect task scheduling. However, we still need to bound the worst-case end-to-end communication latency. As mentioned earlier in Sect. 5, the computed response time of a task is the time that elapsed from when a task becomes active (released) to the time it finishes and is fully unloaded. Therefore, communication data sent by a sender task τ_i will be available to a receiver task after R_i, which is the worst-case response time of τ_i, accounting for interference and overheads.

As described in Sect. 3.6, a set of task chains is defined at compile time; where a chain $\lambda_k = \{\tau_{k,1}, \ldots, \tau_{k,2}, \ldots, \tau_{k,N_k}\}$ defines an ordered set of communicating tasks. The communication data in the chain pass through all the tasks in that chain in sequence. Consequently, the end-to-end communication latency (L^{λ_k}) is the time it takes for the data to be consumed (loaded) by the first task in the chain, i.e, $\tau_{k,1}$, until the last task in the chain, i.e., τ_{k,N_k}, writes the data to main memory (unload). The end-to-end latency of a chain λ_k is computed in Eq. 4:

$$L^{\lambda_k} = \sum_{i=1}^{N_k-1} (R_{k,i} + T_{k,i+1} - 2 \cdot \sigma) + R_{k,N_k}, \tag{4}$$

where $R_{k,i}$ represents the worst-case response time of task $\tau_{k,i}$, as determined in Sect. 5.

Theorem 2 *The worst-case total end-to-end latency of a task chain* λ_k *is* $L^{\lambda_k} = \sum_{i=1}^{N_k-1} (R_{k,i} + T_{k,i+1} - 2 \cdot \sigma) + R_{k,N_k}$.

Proof We seek to maximize the time between the load operation for a job $\tau_{k,1}$, and the unload operation for a successive job of τ_{k,N_k}. Hence, we shall assume that the

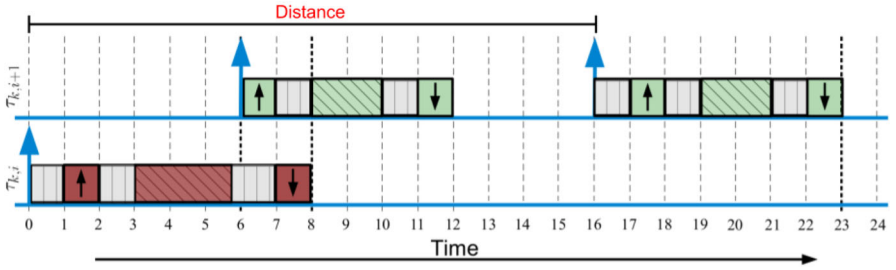

Fig. 10 Worst-case arrival pattern between two communicating tasks

load operation happens as soon as possible, i.e., immediately after $\tau_{k,1}$ arrives, and the unload operation happens as late as possible, i.e., after the worst-case response time R_{k,N_k} of τ_{k,N_k}. We can thus determine the end-to-end latency as follows: we first sum the maximum distance between the arrival times of communicating jobs along the chain, i.e., between a job of $\tau_{k,1}$ and $\tau_{k,2}$, then $\tau_{k,2}$ and $\tau_{k,3}$, and so on until the distance between the arrival times of jobs of τ_{k,N_k-1} and τ_{k,N_k}; and finally we add the response time of τ_{k,N_k}. In the rest of the proof, we show that the maximum distance between the arrival times of communicating jobs of $\tau_{k,i}$ and $\tau_{k,i+1}$ is $R_{k,i} + T_{k,i+1} - 2 \cdot \sigma$; Eq. 4 then directly follows.

The worst-case arrival pattern (critical instant) that leads to the maximum distance between any two communicating jobs is shown in Fig. 10: a job of the receiver task ($\tau_{k,i+i}$) starts loading right before the unload phase of the sender job ($\tau_{k,i}$). This behavior, which is possible if the tasks are mapped to different cores, prevents $\tau_{k,i+1}$ from loading the fresh communication data until its next job. Furthermore, to maximize the distance between the arrival times of the jobs of $\tau_{k,i}$ and $\tau_{k,i+1}$ under such scenario, we assume that $\tau_{k,i}$ unloads its data as late possible, while $\tau_{k,i+i}$ starts loading as soon as possible right after its arrival time. As a consequence, the distance can be computed as the response time $R_{k,i}$, plus the distance between successive jobs of $\tau_{k,i+1}$ which is the period $T_{k,i+1}$, minus $2 \cdot \sigma$ to account for the overlap between the load and unload operations (slots [6 : 8] in Fig. 10). This is equal to $R_{k,i} + T_{k,i+1} - 2 \cdot \sigma$, concluding the proof. □

6 Implementation

In this section, we provide the details of SPM-centric OS implemented using a COTS platform that supports the hardware assumptions described in Sect. 4.

6.1 Architectural overview of considered platform

For the implementation, we used a Freescale MPC5777M micro-controller unit (MCU). This MCU is the most advanced SoC in the Freescale MPC line as of Q4 2015. A brief summary of the architectural features of the MPC5777M MCU is provided in Table 4. The chip includes four processors: two E200Z710 application cores operating

Table 4 Characteristics of Freescale MPC5777M SoC

Chip name	MPC5777M (Matterhorn)
Manufacturer	Freescale
Architecture	Power-PC, 32-bit
CPU unit	2x E200-Z710 + 1x E200-Z709 + 1x E200-Z425 (I/O)
Processing unit	CPUs, DMA, Interrupt Controller, NIC
Operational modes	Parallel + Lockstep (on one applicative core)
ECC protection	Cache, RAM, Flash Storage
Cache hierarchy	L1 (Private Instructions + Data) + Local Memory
Local memory (SPMs)	Instructions (16 KB) + Data (64 KB)
L1 cache size	Instructions (16 KB) + Data (4 KB)
SRAM size	404 KB
Flash size	8 MB
Main peripherals	Ethernet, Flexray, CAN, I2C, SIUL

at 300 MHz and a single E200Z425 I/O core. An additional non-programmable core is included for delayed lockstep operation.

Each core in the system has its own private instruction and data cache as well as globally accessible instruction and data scratchpads. No shared cache is present in the system and the data in the SPM is not cached because of the following reasons: (1) the performance of the SPM on the considered platform is same as that of the cache; (2) there is no difference in the execution time if we cache the SPM data; and (3) the architecture cannot cache the SPM memory. No MMU is available on this platform. Hence, there is no support for virtual memory. Application cores can directly access the SRAM through a dedicated bus. A separate and slower bus is dedicated for transferring peripheral data to/from the I/O core.

6.2 Implementation of SPM-centric OS using Erika Enterprise

Proposed SPM-centric OS was implemented using Evidence Erika Enterprise.[5] Erika Enterprise is an open-source RTOS that is compliant with the AUTOSAR[6] (Automotive Open System Architecture) standard. AUTOSAR is an open standard for automotive architectures providing a basic infrastructure for vehicular software. Erika Enterprise features a small memory footprint, supports multi-core platforms and implements common scheduling policies for periodic tasks. We performed a porting of Erika Enterprise on the MPC5777M MCU, adding support for UART communication interface, interrupt controller, caches, memory protection unit (MPU), data engines (DMA), and Ethernet controller.

[5] http://erika.tuxfamily.org/drupal/.

[6] http://www.autosar.org/.

In order to implement our SPM-centric OS, we have augmented Erika Enterprise to support position-independent (relocatable) tasks. We rely on the compiler[7] support for `far-data` and `far-code` addressing modes. In this way, tasks are compiled to perform program-counter-relative jumps and indirect data addressing with respect to an OS-managed base register. We have extended the default task loader to exploit DMAs for transferring task images from SRAM to local memories and vice-versa. Similarly, the OS scheduler has been adapted to implement the strategy discussed in Sect. 4.

In Erika Enterprise, tasks are compiled and linked directly inside the image of the OS. For each task in the system, Erika-specific meta-data need to be defined. Additionally, meta-data that extend the task descriptors for SPM-centric operations are required. Manually configuring these parameters is tedious and error-prone; hence, we developed an OS configurator. The tool uses high-level task definitions and generates the final configuration for our SPM-centric OS. Specifically, each core is associated with a set of configuration files that describe: number of tasks, their priority, task entry points, initial status and so on. When a task is added, these files need to be configured accordingly.

First, the body of all the tasks is placed in an ad-hoc file. Similarly, task-specific data that need to be preserved across activations are defined in different files and surrounded with appropriate compiler-specific `PRAGMA`. This is fundamental to ensure that: (A) specific linker section is used to store task code and data images; and (B) position-independent data and instructions are generated. A separate file also defines the relocatable task table, which stores the status of each relocatable task. This structure includes: (A) position in SRAM of the task code and data images; (B) position of the task's I/O data buffers; (C) current status of the task (e.g. loaded, completed, unloaded); (D) SPM partition of last relocation.

7 Evaluation

To validate the proposed design and implementation, we performed a series of experiments, whose results are summarized in this section. First, we investigate the overhead of SPM management. Next, we consider the performance and predictability benefits of our approach with synthetic as well as real benchmarks. The achievable I/O bandwidth supported by our design is also measured. Finally, we investigate the schedulability results of the proposed strategy.

7.1 SPM-centric OS overhead evaluation

A crucial parameter of proposed system is the size of the TDMA slot. This should be long enough to allow the completion of a load (or unload) operation for the task with the largest footprint in the system. However, in order to derive an upper-bound, we assume that a task footprint is constrained by the size of an SPM partition. Thereby,

[7] Applications and OS are compiled using the WindRiver Diab Compiler version 5.9.4—http://www.windriver.com/products/development-tools/.

Table 5 Details of OS parameters

Parameter	Time (μs)
Partition load time	432
Partition unload time	432
DMA setup	3.16
Context switch	0.46

we measured the time to copy from/to half SPM (one partition) of an applicative core and derive the TDMA slot size accordingly. The results are reported in Table 5.

The application DMA needs to be programmed by the I/O Core to perform task relocation. Hence, DMA programming time represents an overhead introduced by our design. The time required to program the DMA has been measured and is reported in Table 5. Similarly, Table 5 reports the measured context-switch overhead of the implemented scheduler.

7.2 Results of achievable I/O bandwidth

The performance of the proposed I/O subsystem (see Sect. 4.3) depends on the frequency of load-unload operations. In order to measure the achievable I/O bandwidth of proposed design, we have implemented support for the onboard Fast Ethernet Controller (FEC). The FEC is capable of transmitting data at the highest bandwidth among all the devices of the considered MCU. Hence, it represents the best I/O component to stress-test our design.

We have connected the FEC to an external node which generates constant-rate traffic. Specifically, the traffic source generates a 1 KB packet every 100 μs (1000 Hz, about 82 Mb/s). The payload of each packet contains a flow-ID chosen from four different values in round-robin. On used MCU, each applicative core runs two tasks that have subscribed to I/O data flows based on packets' flow-IDs. Device buffers and task (mirror) I/O buffers have been dimensioned to accommodate a single packet per task, with an overwrite policy.

With this setup, we have derived the raw achievable bandwidth considering two different values of TDMA slot size. Specifically, we measured the data rate of packets that are processed and looped back on the network interface using the Wireshark packet analyzer.[8] Our experiments revealed an achievable bandwidth for the outgoing traffic of 4 Mb/s with a TDMA slot of 800 μs, and 8 Mb/s with a TDMA slot of 400 μs. Although this represents a fraction of the physically available bandwidth (100 Mb/s), being able to sustain a bandwidth higher than 1 Mb/s constitutes a promising result given that the platform operates at a clock frequency of few hundred Hz.

[8] https://www.wireshark.org/.

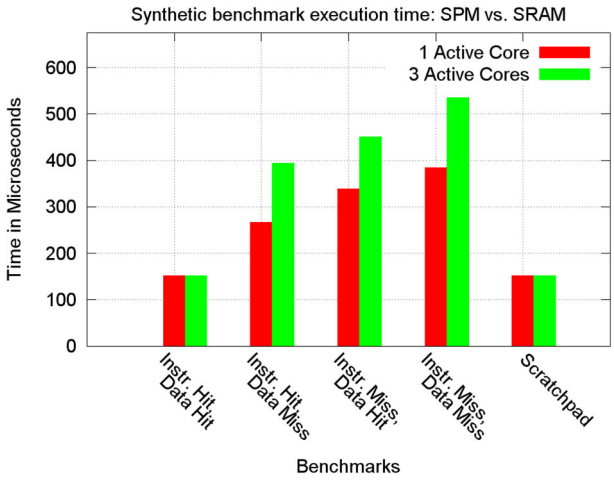

Fig. 11 Experimental execution time for synthetic benchmarks

7.3 Results of synthetic benchmarks

We investigate the performance of SPM-based execution as opposed to a traditional execution model. For this purpose, we have developed a set of synthetic benchmarks that exhibit different memory access benchmarks on one of the two applicative cores. These benchmarks are designed such that they either exhibit good data locality meaning that they are sequential, repetitive and fit well in the cache or they have poor instruction/data locality. Conversely, those benchmarks that exhibit poor instruction/data locality incur in a cache miss for every fetch of a data item and/or instruction. The results of the bechmarks are shown in Fig. 11. The first cluster of bars in Fig. 11 refer to the runtime of the benchmark that exhibits good instruction and data locality. Hence, when it is executed from SRAM, caches are effective at hiding SRAM access latency and significantly reduce task execution time. The next two clusters of bars show that misses suffered for only instruction fetches or only data fetches already induce a significant execution slowdown (around 2x). The need for accessing SRAM data also introduces runtime fluctuation (about 25%) as a result of inter-core interference. Such effect becomes even more severe with applicative code that experiences misses while accessing both instructions and data. If the cost of accessing SRAM memory together with the slowdown due to inter-core interference are considered, an overall 3.5x slowdown is experienced when compared to what has been observed in the ideal case (100% cache hits). Finally, notice that if a task is able to entirely execute from scratchpad, its execution time is comparable to the ideal case and inter-core interference is prevented. These results are a strong motivation to best use available scratchpads in order to improve performance and avoid inter-core interference.

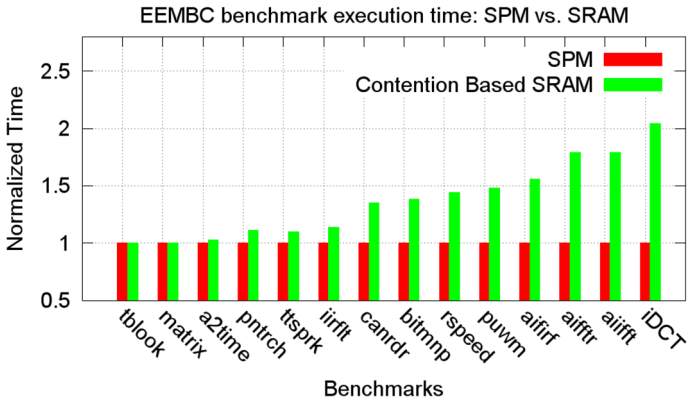

Fig. 12 Experimental execution time for EEMBC benchmarks

7.4 Results of EEMBC benchmarks

Next, we investigate the behavior of EEMBC benchmarks on the selected platform. For this purpose, we have ported and measured the execution time of the full suite of automotive EEMBC benchmarks under two scenarios: traditional contention-based execution from SRAM and proposed SPM-based execution. We have initially explored the stability of the results over time. We observed that the maximum variation across multiple runs of the same configuration was never more than 3 CPU cycles, in spite of our millisecond-scale frame of reference for task activations and execution times. Given the very low variability, the final results were computed by timing 10 runs. The average of the acquired measurements is reported in all the tables and plots. The normalized execution times for all the considered benchmarks are reported in Fig. 12. From the results, we note that computation intensive benchmarks do not benefit from SPM-based execution. Conversely, for memory intensive benchmarks SPM-based execution determines substantial speed-ups (up to 2.1x).

Table 6 shows the execution time of the full suite of EEMBC automotive benchmarks. Furthermore, Table 6 also provides the footprint size of the considered benchmarks. It can be noted that all the considered benchmarks fit into a single scratchpad partition. These results validate the applicability of proposed design in real scenarios.

7.5 Schedulability analysis

For the schedulability evaluation of our approach, we compare our system against the contention-based system, in which cores use caches but are left unregulated when accessing main memory. We consider the platform described in Table 4. For the SPM-centric system, we use the schedulability analysis in Sect. 5, while for the contention-based system, we use response-time analysis for a set of non-preemptive, fixed priority sporadic tasks (Buttazzo 2011). We have considered the applications in Table 6 to generate sets of random tasks (workloads). Given a system utilization, each application

Table 6 Details of EEMBC benchmarks

Benchmark	SPM time (μs)	SRAM time (μs)	Code size (bytes)	Relocatable code size (bytes)	Data size (bytes)
tblook	1013	1015	1804	1892	10,516
matrix	1053	1054	4430	4774	4488
a2time	1002	1029	2175	2538	1704
pntrch	1036	1145	1000	1398	4924
ttsprk	383	425	4124	4772	8160
iirflt	1040	1189	3288	3512	1000
canrdr	1009	1359	1370	1562	12,440
bitmnp	990	1389	3152	3282	1116
rspeed	1012	1457	710	1208	13,212
puwm	1036	1540	1716	2500	2412
aifirf	1005	1564	1554	2286	1552
aifftr	916	1642	3720	4458	8448
aiifft	1170	2092	2796	3540	9224
idct	1045	2126	4498	4690	244

is randomly selected and assigned a random period in the range between 10 ms to 100 ms. The task's utilization is then computed based on the measured execution time of each application and its selected period. At every iteration a new task is randomly generated. The generation stops when the sum of the individual tasks' utilizations reaches the required system utilization. After that, the overhead is added, such as context-switch and DMA setup. For the contention-based system, the execution times reported in SRAM column in Table 6 are used to represent the worst-case execution time including the contention overhead.

Figure 13 shows the result of the schedulability analysis when using proposed SPM-centric OS versus a contention based SRAM system. The figure shows the results in terms of proportion of schedulable task sets for both approaches. Each point in the graph represents 1000 task sets. The results show that the schedulability of the system increases significantly when the proposed SPM-centric approach is used. Hence, the described SPM-centric OS not only improves the predictability of task execution, but it also improves task set schedulability by hiding the main memory access latency, especially for memory intensive applications.

7.6 Communication latency

We evaluate the communication latency by generating random set of tasks as discussed in Sect. 7.5. From a generated task set, we compute the response time of each task using the analysis in Sect. 5. After that, we generate four random communication flows with 5, 10, 15, and 20 tasks and compute the communication latencies of each flow. Each experiment is repeated 1,000 times and the average worst-case communication latency

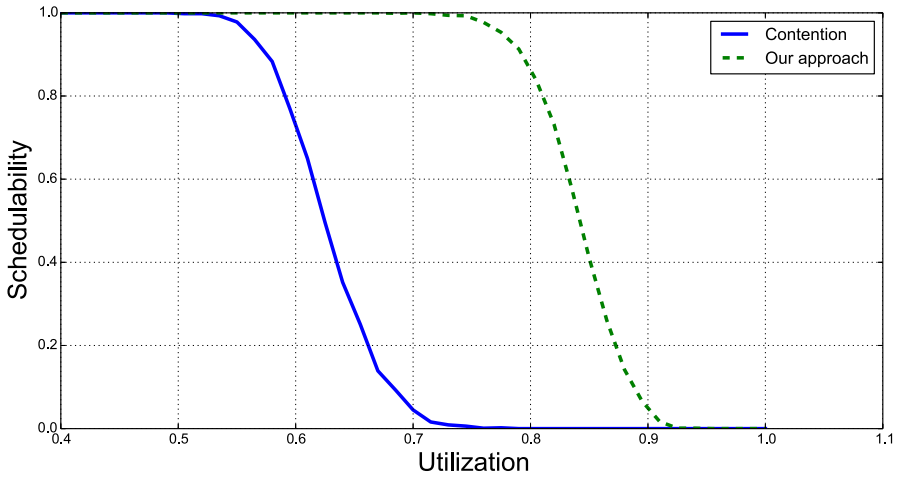

Fig. 13 Schedulability with SPM-based and traditional scheduling models

is reported. Figure 14 shows the estimated worst-case communication latencies for the generated flows comprised of a mix of applications provided in Table 6. As one can observe, there is a slight improvement when the communication tasks are scheduled on the same core. This improvement is because of the DMA phasing. We expect that the improvement should increase with number of application cores. However, exploring this is part of the future work.

In addition, the yellow line in the figure represents the average communication bandwidth, that is, the total amount of data transferred divided by the end-to-end communication latency. For the sake of this experiment, we assumed that each task in a flow will send data equal to the size of its data section as reported in Table 6; hence, the amount of data transferred between any two successive tasks in a flow might be different, which resemble real-life scenarios. Each point in the line is generated by randomly picking 5, 10, 15, or 20 tasks based on the number of tasks in the flow. Then, we compute the total amount of data transferred withing the end-to-end latency window. We repeat the experiment 1,000 times and take the average to capture all the application benchmark in Table 6.

8 Conclusion

In this paper, we presented a novel OS design, namely SPM-centric OS. Proposed SPM-centric OS aims at providing predictability for hard real-time applications on multi-core embedded systems. In order to achieve this goal, we combined resource specialization, high-level scheduling of shared hardware resources as well as a three-phases task execution model. Theoretical results on how to perform schedulability analysis of the proposed scheduling strategy were presented. A complete implementation using a commercially available multi-core platform was also performed to assess the feasibility of our design.

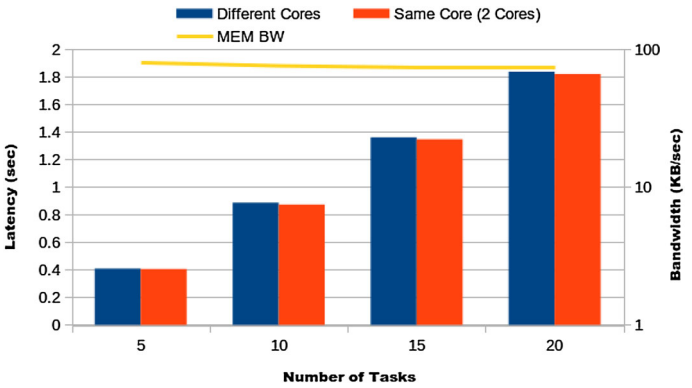

Fig. 14 End-to-end communication latency

Finally, in order to validate the proposed OS design, we have combined experimental results from synthetic and automotive EEMBC benchmarks on the considered platform. In addition to the strong temporal predictability achieved by enhancing inter-core isolation, we are able to exploit the performance benefits of scratchpad memories. Hence, a schedulability improvement over traditional contention-based approaches was obtained. As part of our future work on SPM-centric OS, we plan to investigate the following aspects: support for task preemption, and compliance with standard application interfaces (e.g. AUTOSAR, POSIX).

Acknowledgements The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under Grant Numbers CNS-1646383, NSERC 402369-2011 and 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 NSF and other sponsors.

References

Bai K, Lu J, Shrivastava A, Holton B (2013) CMSM: an efficient and effective code management for software managed multicores. In: Hardware/software codesign and system synthesis (CODES+ ISSS), 2013 international conference on, IEEE, pp 1–9

Betti E, Bak S, Pellizzoni R, Caccamo M, Sha L (2013) Real-time I/O management system with COTS peripherals. IEEE Trans Comput 62(1):45–58

Bui D, Lee EA, Liu I, Patel H, Reineke J (2011) Temporal isolation on multiprocessing architectures. In: Design automation conference (DAC), pp 274 – 279

Buttazzo GC (2011) Hard real-time computing systems: predictable scheduling algorithms and applications, vol 24. Springer, New York

Chattopadhyay S, Roychoudhury A, Rosén J, Eles P, Peng Z (2014) Time-predictable embedded software on multi-core platforms: analysis and optimization. Found Trends Electron Des Autom 8(3–4):199–356

Deverge J-F, Puaut I (2007) WCET-directed dynamic scratchpad memory allocation of data. In: Real-time systems, 2007. ECRTS'07. 19th Euromicro Conference on, IEEE, pp 179–190

Durrieu G, Faugere M, Girbal S, Perez DG, Pagetti C, Puffitsch W (2014) Predictable flight management system implementation on a multicore processor. In: ERTSS'14

FAA position paper on multi-core processors, CAST-32 (rev 0). http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast32.pdf. Accessed 26 Jan 2015

Falk H, Kleinsorge JC (2009) Optimal static WCET-aware scratchpad allocation of program code. In: Proceedings of the 46th annual design automation conference, ACM, pp 732–737

Farshchi F, Valsan PK, Mancuso R, Yun H (2018) Deterministic memory abstraction and supporting multicore system architecture

Flodin J, Lampka K, Wang Y (2014) Dynamic budgeting for settling dram contention of co-running hard and soft real-time tasks. In: Industrial embedded systems (SIES), 2014 9th IEEE international symposium on, IEEE, pp 151–159

Girbal S, Jean X, Le Rhun J, Perez DG, Gatti M (2015) Deterministic platform software for hard real-time systems using multi-core COTS. In: Digital avionics systems conference (DASC), 2015 IEEE/AIAA 34th, IEEE, pp 8D4–1

Jean X, Faura D, Gatti M, Pautet L, Robert T (2012) Ensuring robust partitioning in multicore platforms for ima systems. In: Digital avionics systems conference (DASC), 2012 IEEE/AIAA 31st, IEEE, pp 7A4–1

Kai L, Adam L (2016) Resolving contention for networks-on-chips: Combining time-triggered application scheduling with dynamic budgeting of memory bus use. In: International GI/ITG conference on measurement, modelling, and evaluation of computing systems and dependability and fault tolerance, Springer, pp 137–152

Lampka K, Giannopoulou G, Pellizzoni R, Zheng W, Stoimenov N (2014) A formal approach to the wcrt analysis of multicore systems with memory contention under phase-structured task sets. Real Time Syst 50(5–6):736–773

Li L, Gao L, Xue J (2005) Memory coloring: a compiler approach for scratchpad memory management. In: Parallel architectures and compilation techniques, 2005. PACT 2005. 14th international conference on, IEEE, pp 329–338

Lu J, Bai K, Shrivastava A (2013) SSDM: smart stack data management for software managed multicores (SMMs). In: Proceedings of the 50th annual design automation conference, ACM, pp 149

Mancuso R, Dudko R, Betti E, Cesati M, Caccamo M, Pellizzoni R (2013) Real-time cache management framework for multi-core architectures. In: Real-time and embedded technology and applications symposium (RTAS), 2013 IEEE 19th, IEEE, pp 45–54

Mancuso R, Pellizzoni R, Caccamo M, Sha Lui, Yun Heechul (2015) WCET(m) estimation in multi-core systems using single core equivalence. In: Real-time systems (ECRTS), 2015 27th Euromicro conference on, pp 174–183

Metzlaff S, Guliashvili I, Uhrig S, Ungerer T (2011) A dynamic instruction scratchpad memory for embedded processors managed by hardware. In: Architecture of computing systems-ARCS 2011, Springer, pp 122–134

Pellizzoni R, Betti E, Bak S, Yao G, Criswell J, Caccamo M Kegley R (2011) A predictable execution model for COTS-based embedded systems. In: Proceedings of the 2011 17th IEEE real-time and embedded technology and applications symposium, RTAS '11, IEEE Computer Society, Washington, DC, USA, pp 269–279

Puau I, Pais C (2007) Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In: Design, automation & Test in Europe conference & exhibition, 2007. DATE'07, IEEE, pp 1–6

Schranzhofer A, Pellizzoni R, Chen Jian-Jia, Thiele L, Caccamo M (2010) Worst-case response time analysis of resource access models in multi-core systems. In: Proceedings of the 47th design automation conference, ACM, pp 332–337

Software techniques for scratchpad memory management. http://memsys.io/wp-content/uploads/2015/09/p98-sebexen.pdf. Accessed 26 Jan 2015

Suhendra V, Roychoudhury A, Mitra T (2010) Scratchpad allocation for concurrent embedded software. ACM Trans Program Lang Syst 32(4):13

Tabish R, Mancuso R, Wasly S, Alhammad A, Phatak SS, Pellizzoni R, Caccamo M (2016) A real-time scratchpad-centric os for multi-core embedded systems. In: Real-time and embedded technology and applications symposium (RTAS), 2016 IEEE, IEEE, pp 1–11

Takase H, Tomiyama H, Takada H (2010) Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In: Design, automation & test in Europe conference & exhibition (DATE), 2010, IEEE, pp 1124–1129

Ungerer T, Cazorla F, Sainrat P, Bernat G, Petrov Z, Rochange C, Quinones E, Gerdes M, Paolieri M, Wolf J, Casse H, Uhrig S, Guliashvili I, Houston M, Kluge F, Metzlaff S, Mische J (2010) MERASA: multicore execution of hard real-time applications supporting analyzability. IEEE Micro 30(5):66–75

Wasly S, Pellizzoni R (2013) A dynamic scratchpad memory unit for predictable real-time embedded systems. In: Real-time systems (ECRTS), 2013 25th Euromicro Conference on, IEEE, pp 183–192
Wasly S, Pellizzoni R (2014) Hiding memory latency using fixed priority scheduling. In: Real-time and embedded technology and applications symposium (RTAS), 2014 IEEE 20th, IEEE, pp 75–86
Whitham J, Audsley NC (2012) Explicit reservation of local memory in a predictable, preemptive multitasking real-time system. In: Real-time and embedded technology and applications symposium (RTAS), 2012 IEEE 18th, IEEE, pp 3–12
Whitham J, Davis RI, Audsley NC, Altmeyer S, Maiza C (2012) Investigation of scratchpad memory for preemptive multitasking. In: Real-time systems symposium (RTSS), 2012 IEEE 33rd, IEEE, pp 3–13
Wilding MM, Hardin DS, Greve DA (1999) Invariant performance: a statement of task isolation useful for embedded application integration. In: dcca, IEEE, p. 287
Wolf J, Gerdes M, Kluge F, Uhrig S, Mische J, Metzlaff S, Rochange C, Cassé H, Sainrat P, Ungerer T (2010) RTOS support for parallel execution of hard real-time applications on the MERASA multicore processor. In: Object/component/service-oriented real-time distributed computing (ISORC), 2010 13th IEEE international symposium on, IEEE, pp 193–201
Yun H, Mancuso R, Wu ZP, Pellizzoni R (2014) 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, IEEE, pp 155–166
Yun H, Yao G, Pellizzoni R, Caccamo M, Sha L (2013) Memguard: memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In: Real-time and embedded technology and applications symposium (RTAS), 2013 IEEE 19th, IEEE, pp 55–64

Rohan Tabish works as a graduate student in the Department of Computer Science at the University of Illinois at Urbana-Champaign, IL, USA. His research interests are in real-time and embedded systems with a focus on the use of OS-level techniques in multi-core processors to achieve predictability and strong timing guarantees such that they can be deployed in safety-critical automotive and avionics applications. He is also interested in wireless communications and sensor networks. He is an IEEE student member.

Renato Mancuso is an assistant professor at the department of Computer Science at Boston University (BU) since September 2017. He received his Ph.D. from the University of Illinois at Urbana-Champaign (UIUC) the same year. His research focuses on real-time and embedded systems, with specific focus on OS-level multi-core resource management technologies for high-performance, safety-critical avionics and automotive systems. Renato is also interested in applications and methodologies to design, deploy and analyze Cyber-Physical Systems (CPS) in general, and autonomous terrestrial and aerial vehicles in particular. He is a member of the IEEE.

Saud Wasly is an Assistant Professor in the Department of Electrical and Computer Engineering at King Abdulaziz University, Saudi Arabia. He received his Ph.D. in 2018 from the Department of Electrical and Computer Engineering at the University of Waterloo, Canada. His research interests include embedded systems architecture, real-time operating systems, timing analysis, and architectural simulation. He is a member of the IEEE.

Rodolfo Pellizzoni is Associate Professor in the Department of Electrical and Computer Engineering at the University of Waterloo. He received his master degree from Scuola Superiore Sant'Anna in 2005 and his Ph.D. from the University of Illinois at Urbana-Champaign in 2010. Rodolfo's main research interests are in real-time systems and timing analysis, with a particular focus on hardware/software architectures for timing predictability and safety certification.

Marco Caccamo studied Computer Engineering at University of Pisa (Italy). Following his Master degree in computer engineering in July 1997, he earned his Ph.D. in computer engineering from Scuola Superiore SantAnna (Italy) in 2002. Shortly after graduation, he joined University of Illinois at Urbana-Champaign as assistant professor in Computer Science and was promoted to associate professor at the age of 36, then he became a full professor in 2014. Since 2018, Prof. Caccamo has been appointed to the chair of Cyber-Physical Systems in Production Engineering at TUM. He has chaired Real-Time Systems Symposium and Real-Time and Embedded Technology and Applications Symposium, the two IEEE flagship conferences on Real-Time Systems. He also has served as General Chair of Cyber Physical Systems Week. In 2003, he was awarded an NSF CAREER Award. He is a recipient of the Alexander von Humboldt Professorship and he is an IEEE Fellow.

Affiliations

Rohan Tabish[1] · Renato Mancuso[2] · Saud Wasly[3] · Rodolfo Pellizzoni[4] · Marco Caccamo[5]

Renato Mancuso
rmancuso@bu.edu

Saud Wasly
saudalwasli@gmail.com

Rodolfo Pellizzoni
rpellizz@uwaterloo.ca

Marco Caccamo
mcaccamo@tum.de

1 University of Illinois at Urbana-Champaign, Champaign, IL, USA

2 Boston University, Boston, MA, USA

3 Kind AbdulAziz University, Jeddah, Saudi Arabia

4 University of Waterloo, Waterloo, ON, Canada

5 Technical University of Munich, Munich, Germany