A Real-Time Scratchpad-centric OS for Multi-core Embedded Systems

Rohan Tabish*, Renato Mancuso*, Saud Wasly**, Ahmed Alhammad**, Sujit S. Phatak+, Rodolfo Pellizzoni**
Marco Caccamo*
*University of Illinois at Urbana-Champaign, USA, {rtabish, rmancus2, mcaccamo}@illinois.edu
**University of Waterloo, Canada, {swasly, a2alhamm, rpellizz}@uwaterloo.ca
+ Hitachi America, Ltd., sujit.phatak@hitachi-automotive.us

***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 the 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 the time domain can be enforced. To validate the proposed design, we implemented the proposed OS using a commercial-off-the-shelf (COTS) platform. Experiments show that this novel design delivers predictable temporal behavior to hard real-time tasks, and it improves performance up to 2.1x compared to traditional approaches.

I. 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, including: main memory, buses, caches, and I/O peripherals. Unfortunately, when one or more of these resources is utilized up to its saturation point, severe and unpredictable inter-core interference can heavily affect the system's temporal behavior. 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 [1].

This problem has been approached from different perspectives: a) novel multi-core hardware platforms have been designed [2, 3], b) new OS-level techniques have been developed to perform shared resource partitioning and management on commercial-off-the-shelf (COTS) platforms [4]. 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 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 [5, 6]. 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 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 novel analysis is derived to calculate the response time of real-time tasks under the proposed scheduling strategy.
4) Finally, a full implementation of the proposed OS has been performed using a commercially available multi-core micro-controller. Its design has been validated using a combination of synthetic tasks and EMBC benchmarks.

The rest of the paper is organized as follows. Section II briefly reviews the related work. Next, Section III introduces the considered system model and architectural assumptions. The design of the proposed OS is described in Section IV, while Section V discusses how to conduct task schedulability analysis. We describe the performed implementation in Section VI, and discuss the experimental results in Section VII. Finally, the paper concludes in Section VIII.

II. 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 resource management [2, 3, 4]) 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 [7]. 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 [8], 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 [1] 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 [9]. 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 [9], 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 [10].

The AER model proposed in [10] 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 [10], 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 [11]. 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 [2, 3]. Specifically, the precision timed (PRET) architecture [2] introduces task runtime control and deadline enforcement at the instruction set (ISA) level. Additional hardware modifications allow to achieve better performance without sacrificing predictability. Similarly, in the MERASA project [3, 12], 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 [4, 13, 14, 15] 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. Three main differences exist with respect to the proposed approach. First, the work in [4] 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.

The proposed work is a contribution to existing literature on the usage of scratchpad memories (SPM) for real-time systems [5, 6, 16, 17, 18, 19]. In fact, a number of works have explored the benefits of scratchpad memories over traditional caches for multi-core platforms [5, 6]. Other works on embed-

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

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

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

ded systems exist that propose scratchpad memory allocation strategies targeting real-time applications [20, 21, 22, 23, 24]. In [19] 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 [19] is not predictability but energy efficiency, making its contribution substantially different from the proposed work.

Finally, our design shares some similarities with scratchpad scheduling approaches that have been proposed in [16, 17, 18, 25]. Compared to these works, our approach mainly differs in three 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; and (iii) it is also discussed how I/O traffic issued by different cores is deconflicted.

III. System Model and Assumptions

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

a) Scratchpad Memories: The first assumption we make is the presence of scratchpad memory (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 largest task in the system is at most half the size of the scratchpad memory. Although this assumption may appear restrictive, we make the following considerations. First, modern scratchpad-based micro-controllers provide scratchpad memories that have a size 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 [26, 27] to split a large application into smaller sub-tasks that are individually compliant with the imposed constraint.

As discussed in Section IV, 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 of the task is written back to main memory during the unload phase.

b) 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 direct memory access (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 scratchpad memory, it is possible to hide task loading/unloading overhead during task execution, as we discuss in Section IV.

c) 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 [28]. 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 [28, 29]. 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 I 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-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 scratchpad memory 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 scratchpad memory. 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 Figure 1.

d) 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

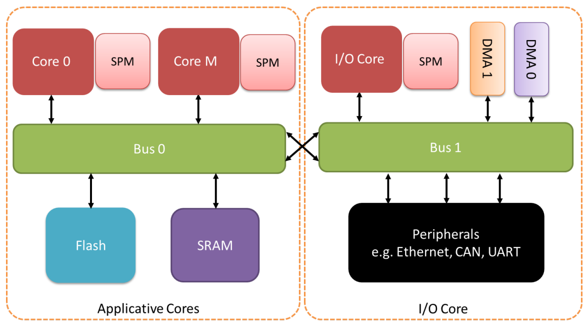


Fig. 1. Multicore architecture satisfying our hardware assumptions

presence of a memory management unit (MMU) is not a necessary requirement. We discuss in Section VI 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 I provides a list of some of the available COTS platforms that satisfy the described assumptions.

TABLE I
SUITABLE COMMERCIAL MULTICORE COTS PLATFORMS

Features	**MPC5777M**	**MPC5746M**	**TMS320C6678**
Scratchpad	✓	✓	✓
DMA engines	✓	✓	✓
Dedicated I/O bus	✓	✓	✗

e) 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, $\{\tau_1, ..., \tau_N\}$, 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 II summarizes the notation used for task parameters. As discussed in Section IV, 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.

IV. Proposed Operating System Design

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

A. 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 Figure 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.

TABLE II
TASK'S PARAMETERS

Term	**Definition**
τ_i	a task in the system
$\tau_i.T$	task's MIT or period(if task is periodic)
$\tau_i.c$	task's execution time including all overheads
σ	TDMA slot size for the DMA operation

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 Section III, 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 Section VI.

In the 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.

B. 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 consumers. For the scope of our design, we employ a time division multiple access (TDMA) scheme to serialize task load/unload operations among M applicative cores. The main advantage of the TDMA scheme lies in its simplicity of implementation. Although in this work we restrict our discussion to TDMA sharing of the applicative DMA, the proposed OS can be extended to consider round-robin policies as well as budget-based schemes.

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, Figure 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 Figure 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

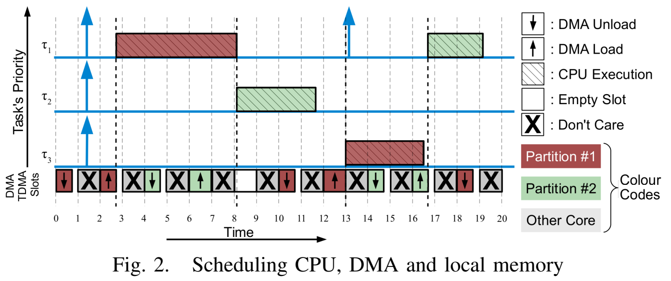


Fig. 2. Scheduling CPU, DMA and local memory

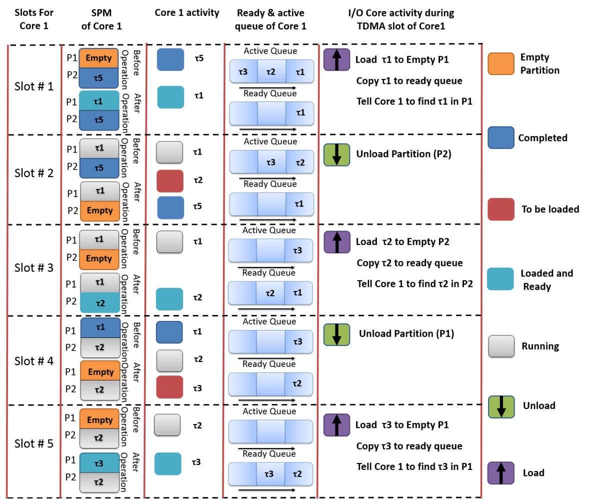


Fig. 3. Interaction between I/O Core and Core 1 for task scheduling.

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 work-flow followed by an applicative core and the I/O core at the boundary of each TDMA slot is depicted in Figure 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.

C. 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 Section III, 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 scratchpad. A device buffer is further divided into a *input device buffer* and a *output device buffer*. The input (output) device buffer represents the position in memory where data produced by devices (tasks) is accumulated before being dispatched to tasks (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.

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*. 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 Section V for the calculation of tasks' response time can be reused to reason about end-to-end delay of I/O-related events.

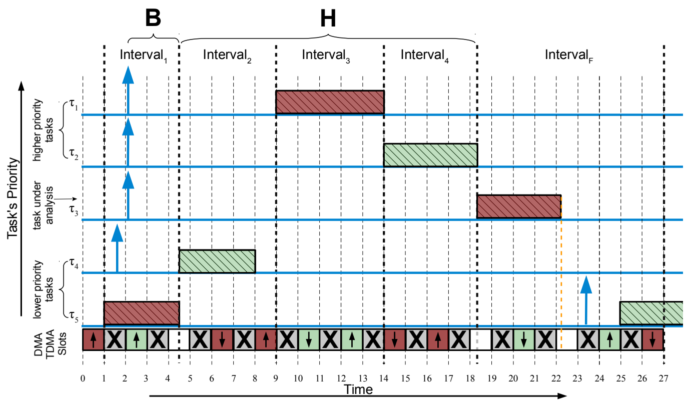


Fig. 4. SPM-centric OS task scheduling. Scheduling intervals are highlighted.

V. Schedulability Analysis

Given the scheduling strategy described in Section IV, we can calculate a safe bound on the worst case execution time based on all tasks' parameters in an approach similar to [25]. Note that we assume that the task's execution time, $\tau_i.c$, 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 4 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 [25], we consider the busy period as composed by a sequence of *scheduling intervals* $Interval_1, Interval_2, Interval_3, Interval_4$ (each bounded by bold vertical lines in the figure), followed by a *final interval* $Interval_F$. 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 *I/O-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.

A. Response Time Calculation

Building on the above-mentioned definitions, we can use the same algorithm detailed in [25] to compute the worst case response time for the task under analysis, by showing that the problem is equivalent to the one in [25]. The algorithm in [25] computes 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 computation of the task under analysis ($\tau_i.c$). The algorithm builds a list of DMA times and computation times for tasks executed in H, then it derives a provably safe bound on the length of H using standard response time iteration.

Compared to [25], our solution differs in three aspects. First, in this work we use fixed-size DMA operations, while [25] employs dynamic-size DMA operations. Therefore, we need to discuss how to compute the length of the DMA operations that are inserted in the list of DMA times. 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 [25], we consider the task under analysis finished when the task is unloaded, at the end of $Interval_F$. Consequently, we can use the same algorithm to compute the worst case response by replacing $\tau_i.c$ with the length of $Interval_F$. We address each point in sequence.

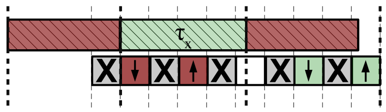
B. Scheduling Intervals in The Busy Period (H)

When the system is busy with both SPM partitions occupied and at least one pending 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 I/O-bound interval, such as for $Interval_3$ in the figure. 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 can induce 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 $\max(\tau.c, 4 \cdot \sigma)$ or $\max(\tau.c, 5 \cdot \sigma)$. 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 ($4 \cdot \sigma$).

Lemma 1: 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. By considering the figure above, the execution of τ_x in the middle interval is greater than $4 \cdot \sigma$, and it induces an empty slot in the next interval. Since both partitions must be full during the execution of intervals in H, it follows that the interval to which τ_x belongs must include both a load and an unload operation; in the worst case showed in the figure, the interval could start with the unload operation. Still, clearly if the execution time of τ_x is reduced to less than $4 \cdot \sigma$, the interval will finish before the next slot assigned to the core under analysis is reached, and hence no empty slot will be induced. Note that if the execution of τ_x is reduced further, it could make the interval into an I/O-bound interval, which would end right after finishing the load of the next task; thus, the next interval would not suffer from an empty induced slot either. ■

Based on Lemma 1, we can construct the list of DMA times 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 in the figure). The remaining DMA times in the list are equal to $4 \cdot \sigma$.

C. Critical Instant and Blocking Time (B)

At the beginning of the example schedule in Figure 4, 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. 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 2: 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 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. Finally, 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 (i.e., two slots before τ_i arrives, which is slot [0:1] in Figure 4) maximizes the amount of execution of the task within the busy period. ■

Based on Lemma 2, the worst case blocking time B can be obtained as the length of $Interval_1$ minus σ, where the length of $Interval_1$ is bounded by $\max(\tau_u.c, 2\cdot\sigma)$; here, $\tau_u.c$ 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 in the second slot of the interval (slot [2:3] in the figure). Similar to [25], since we can make no assumption on which lower priority tasks execute in $Interval_1$ and $Interval_2$, the algorithm simply considers the two lower priority tasks with the longest execution times.

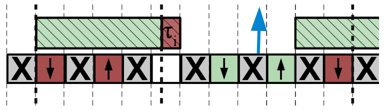
D. *Final Interval* (F)

The length of the final interval $Interval_F$ can be computed as $\max(\tau_i.c+5\cdot\sigma, 7\cdot\sigma)$, where τ_i is the task under analysis. In the example depicted in Figure 4, the length of $Interval_F$ is $\tau_3.c+5\cdot\sigma$. The other case can happen when $\tau_3.c$ 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 3: The length of $Interval_F$ is upper bounded by $\max(\tau_i.c+5\cdot\sigma, 7\cdot\sigma)$.

Proof: Similar to scheduling intervals in H, we need to consider two cases: (1) the length of the interval is bounded by $\tau_i.c$ 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 partition before unloading $\tau_i.c$. For the first case, note that differently from scheduling intervals in H, there might be no pending 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 release 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 Figure 4. 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 Figure 4 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 $\tau_i.c+5\cdot\sigma$.

The following figure shows 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 $\tau_i.c+5\cdot\sigma$ 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.



∎

TABLE III
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

VI. IMPLEMENTATION

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

A. *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 III. The chip includes four processors: two E200Z710 application cores operating at 300 MHz and a single E200Z425 I/O core. An additional non-programmable core is included for delayed lockstep operation.

Each core features private, globally accessible scratchpads for instructions and data, with a size of 16 KB and 64 KB respectively. 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.

B. *Implementation of SPM-centric OS using Erika Enterprise*

The proposed SPM-centric OS was implemented using Evidence Erika Enterprise[4]. Erika Enterprise is an open-source RTOS that is compliant with the AUTOSAR[5] (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.

In order to implement our SPM-centric OS, we have augmented Erika Enterprise to support position-independent (relocatable) tasks. We rely on the compiler[6] 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

[4]http://erika.tuxfamily.org/drupal/
[5]http://www.autosar.org/
[6]Applications and OS are compiled using the WindRiver Diab Compiler version 5.9.4 - http://www.windriver.com/products/development-tools/

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 Section IV.

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.

VII. 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.

A. SPM-Centric OS Overhead Evaluation

A crucial parameter of the 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, 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 IV.

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 IV. Similarly, Table IV reports the measured context-switch overhead of the implemented scheduler.

B. Results of Achievable I/O Bandwidth

The performance of the proposed I/O subsystem (see Section IV-C) depends on the frequency of load/unload operations. In order to measure the achievable I/O bandwidth of the 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.

TABLE IV
Details of OS Parameters

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

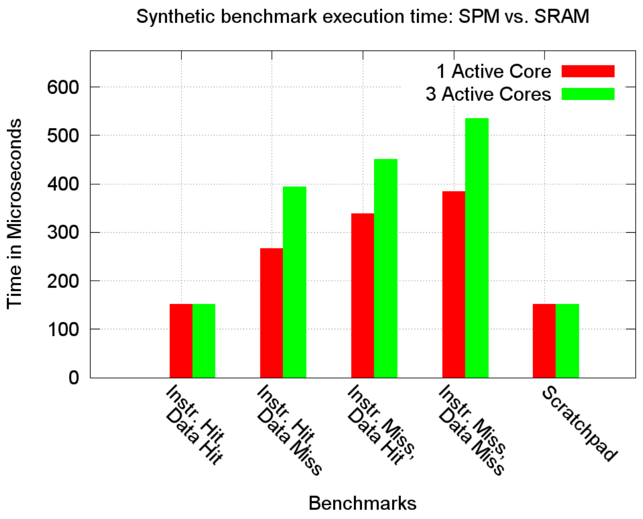


Fig. 5. Experimental execution time for synthetic benchmarks.

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 4 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[7]. 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.

C. 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 patterns. Figure 5 depicts the runtime for such benchmarks on one of the two applicative cores. The

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

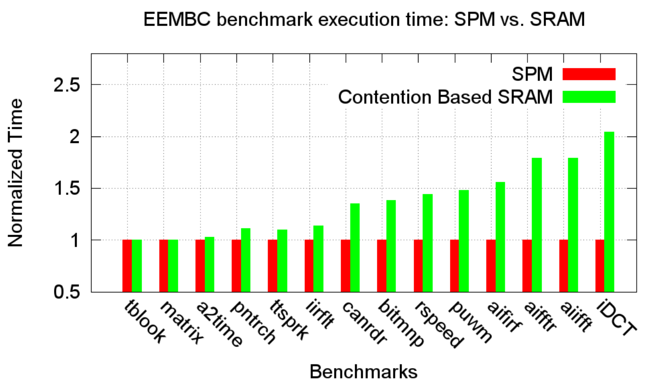


Fig. 6. Experimental execution time for EEMBC benchmarks.

first cluster of bars refer to the runtime of the benchmark that exhibits good 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.

D. 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 the proposed SPM-based execution. The results of normalized execution times are shown in Figure 6. 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 V shows the execution time of the full suite of EEMBC automotive benchmarks. Furthermore, Table V 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 the proposed design in real scenarios.

E. 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. Standard response time analysis is applied on both our system and the contention-based system for the same

TABLE V
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	10516
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	12440
bitmnp	990	1389	3152	3282	1116
rspeed	1012	1457	710	1208	13212
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

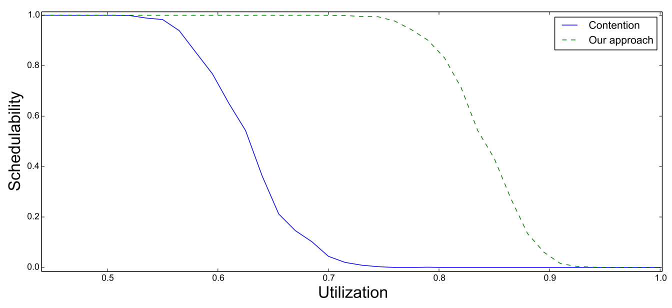


Fig. 7. Schedulability with SPM-based and traditional scheduling models.

simulated workload. We have considered the applications in Table V to generate sets of random tasks (workloads). Given a system utilization, each application 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 V are used to represent the worst-case execution time including the contention overhead.

Figure 7 shows the result of the schedulability analysis when using the 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.

VIII. 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.

Finally, in order to validate 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, inter-process communication, and compliance with standard application interfaces (e.g. AUTOSAR, POSIX).

IX. Acknowledgment

The material presented in this paper is based upon work supported by Hitachi America Ltd. under contract Hitachi 2013-07132, the National Science Foundation (NSF) under grant numbers CNS-1302563, and CNS-1219064, NSERC DG 402369-2011 and CMC Microsystems. 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

[1] FAA position paper on multi–core processors, CAST32 (rev 0). http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast32.pdf. Accessed: 2015-01-26.

[2] D. Bui, E.A. Lee, I. Liu, H. Patel, and J. Reineke. Temporal isolation on multiprocessing architectures. In *Design Automation Conference (DAC)*, pages 274 – 279, June 2011.

[3] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. MERASA: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010. ISSN 0272-1732.

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

[5] I. Puau and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*, pages 1–6. IEEE, 2007.

[6] S. Metzlaff, I. Guliashvili, S. Uhrig, and T. Ungerer. A dynamic instruction scratchpad memory for embedded processors managed by hardware. In *Architecture of Computing Systems-ARCS 2011*, pages 122–134. Springer, 2011.

[7] Wilding, M. M. and Hardin, D. S. and Greve, D. A. Invariant performance: A statement of task isolation useful for embedded application integration. In *dcca*, page 287. IEEE, 1999.

[8] Jean, X. and Faura, D. and Gatti, M. and Pautet, L. and Robert, T. Ensuring robust partitioning in multicore platforms for ima systems. In *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st*, pages 7A4–1. IEEE, 2012.

[9] Girbal, S. and Jean, X. and Le Rhun, J. and Perez, D. G. and Gatti, M. Deterministic platform software for hard real-time systems using multi-core COTS. In *Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th*, pages 8D4–1. IEEE, 2015.

[10] Durrieu, G. and Faugere, M. and Girbal, S. and Pérez, D. G. and Pagetti, C. and Puffitsch, W. Predictable flight management system implementation on a multicore processor. *ERTSS'14*, 2014.

[11] S. Chattopadhyay, A. Roychoudhury, J. Rosén, P. Eles, and Z. Peng. Time-predictable embedded software on multi-core platforms: Analysis and optimization. *Foundations and Trends in Electronic Design Automation*, 8(3-4):199–356, July 2014.

[12] J. Wolf, M. Gerdes, F. Kluge, S. Uhrig, J. Mische, S. Metzlaff, C. Rochange, H. Cassé, P. Sainrat, and T. Ungerer. RTOS support for parallel execution of hard real-time applications on the MERASA multi-core processor. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*, pages 193–201. IEEE, 2010.

[13] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 55–64. IEEE, 2013.

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

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

[16] S. Wasly and R. Pellizzoni. A dynamic scratchpad memory unit for predictable real-time embedded systems. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 183–192. IEEE, 2013.

[17] J. Whitham, R.I. Davis, N.C. Audsley, S. Altmeyer, and C. Maiza. Investigation of scratchpad memory for preemptive multitasking. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 3–13. IEEE, 2012.

[18] J. Whitham and N.C Audsley. 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*, pages 3–12. IEEE, 2012.

[19] Takase, H. and Tomiyama, H. and Takada, H. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 1124–1129. IEEE, 2010.

[20] Deverge, J-F and Puaut, Isabelle. WCET-directed dynamic scratchpad memory allocation of data. In *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, pages 179–190. IEEE, 2007.

[21] Lu, J. and Bai, K. and Shrivastava, A. SSDM: smart stack data management for software managed multicores (SMMs). In *Proceedings of the 50th Annual Design Automation Conference*, page 149. ACM, 2013.

[22] Bai, Ke and Lu, Jing and Shrivastava, Aviral and Holton, Bryce. CMSM: an efficient and effective code management for software managed multicores. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*, pages 1–9. IEEE, 2013.

[23] Suhendra, Vivy and Roychoudhury, Abhik and Mitra, Tulika. Scratchpad allocation for concurrent embedded software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(4):13, 2010.

[24] Falk, H. and Kleinsorge, J. C. Optimal static WCET-aware scratchpad allocation of program code. In *Proceedings of the 46th Annual Design Automation Conference*, pages 732–737. ACM, 2009.

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

[26] Software techniques for scratchpad memory management. http://memsys.io/wp-content/uploads/2015/09/p98-sebexen.pdf. Accessed: 2015-01-26.

[27] L. Li, L. Gao, and J. Xue. Memory coloring: A compiler approach for scratchpad memory management. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 329–338. IEEE, 2005.

[28] E. Betti, S. Bak, R. Pellizzoni, M. Caccamo, and L. Sha. Real-time I/O management system with COTS peripherals. *Computers, IEEE Transactions on*, 62(1):45–58, 2013.

[29] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. 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, pages 269–279, Washington, DC, USA, 2011. IEEE Computer Society.