Implementation of Partitioned Mixed-Criticality Scheduling on a Multi-Core Platform

ROMAN TRÜB, GEORGIA GIANNOPOULOU, ANDREAS TRETTER, and LOTHAR THIELE, Computer Engineering and Networks Laboratory, ETH Zurich, Switzerland

Recent industrial trends favor the adoption of multi-core architectures for mixed-criticality applications. Although several mixed-criticality multi-core scheduling approaches have been proposed, currently there are few implementations on hardware that demonstrate efficient resource utilization and the ability to bound interference on shared resources. To address this necessity, we develop a mixed-criticality runtime environment on the Kalray MPPA-256 Andey many-core platform. The runtime environment implements a scheduling policy based on adaptive temporal partitioning. We develop models, methods and implementation principles to implement the necessary scheduling primitives, to achieve high platform utilization and to perform a compositional worst-case execution time analysis. The bounds account for scheduling overheads and for the inter-task interference on the platform's shared memory. Using realistic benchmarks from avionics and signal processing, we validate the correctness and tightness of the bounds and demonstrate a high platform utilization.

CCS Concepts: • **Computer systems organization → Embedded software**; *Real-time operating systems*; *Real-time system architecture*;

Additional Key Words and Phrases: multi-core, mixed-criticality, adaptive temporal partitioning, MPPA-256

ACM Reference format:
Roman Trüb, Georgia Giannopoulou, Andreas Tretter, and Lothar Thiele. 2017. Implementation of Partitioned Mixed-Criticality Scheduling on a Multi-Core Platform. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 122 (September 2017), 21 pages.
https://doi.org/10.1145/3126533

1 INTRODUCTION

The prevalence of multi-core architectures in the electronic market has led to an ongoing shift from single-core to multi-core designs even in safety-critical domains, such as avionics and automotive. Safety-critical systems are often mixed-criticality systems, in which applications with different safety criticality levels are hosted on a common platform.

Although theoretic aspects of mixed-criticality multi-core scheduling have been studied, there is a lack of implementations on hardware that demonstrate efficient resource utilization and the ability to bound interference on shared resources. In this work, we describe models, methods and implementation principles that enable the deployment of mixed-criticality applications on

This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2017 and appears as part of the ESWEEK-TECS special issue.
Authors' addresses: R. Trüb, G. Giannopoulou, A. Tretter, and L. Thiele, Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland; emails: {roman.trueb, georgia.giannopoulou, andreas.tretter, lothar.thiele}@tik.ee.ethz.ch.

ACM 1539-9087/2017/09-ART122 $15.00
https://doi.org/10.1145/3126533

multicores with a high degree of *predictability* and *efficiency*, such that real-time guarantees are provided for all applications and a high utilization of the platform resources is possible. We accomplish this based on three key design factors:

Adaptive temporal partitioning. Our multi-core scheduling approach reduces the complexity of interference analysis by applying global temporal partitioning, i.e. by allowing only applications of the same criticality to utilize the platform resources at any time [4, 11, 13]. For efficient resource utilization, the schedule of the temporal partitions can be dynamically adapted at runtime to react to occasionally higher resource demand (from high-criticality applications). Timing predictability is enabled by (i) eliminating interference among applications with different criticality by construction, (ii) restricting interference to statically known task sets, (iii) avoiding complex scheduling primitives with potentially high overheads, like task migrations.

Efficient implementation of scheduling primitives. The successful deployment of mixed-criticality applications on multi-core architectures depends on the ability to implement scheduling primitives, such as inter-core synchronization and dynamic scheduling adaptations, with bounded and low overhead. Sigrist et al. showed that the overhead of mixed-criticality mechanisms can have a substantial influence on schedulability [20]. To avoid this, we describe efficient and predictable primitives for the implementation of adaptive temporal partitioning on the MPPA-256 Andey [7], such as synchronization across cores and time-triggered execution.

Bounded interference on shared resources. Bounding the tasks' mutual delays due to interference on shared resources is highly complex, see [6]. We show how implementing adaptive temporal partitioning on the MPPA-256 Andey can lead to a tight bounding of such delays. In addition, we experimentally validate the usefulness and correctness of the underlying system abstractions.

The adaptive temporal partitioning has been addressed in the work of Giannopoulou et al. [11]. Our work mainly focuses on the last two design factors, the efficient implementation of scheduling primitives and the bounded interference on shared resources. Note that although existing mixed-criticality policies [4, 11] implement adaptive temporal partitioning, so far there has been insufficient empirical evidence on whether these two goals can be achieved on available multicores. By presenting the first deployment of adaptive temporal partitioning on a timing-predictable many-core platform, we show that this approach is indeed a viable solution to mixed-criticality scheduling. Our main contributions can be summarized as follows:

- We propose and compare different implementations of scheduling primitives on the MPPA-256 Andey platform. Based on measurements, we bound the overhead of the different implementations and select the ones with minimal overhead.
- We present a worst-case timing analysis which accounts for the runtime overheads and the worst-case interference on the memory paths of the MPPA-256 Andey platform. This allows us to provide real-time guarantees for scheduled task sets with bounded execution time and resource access requirements.
- With a set of industrial-representative benchmarks, we perform a case study with measurements to demonstrate the efficient implementation of our scheduling framework as well as to evaluate the bounds of the worst-case timing analysis.

2 RELATED WORK

The deployment of mixed-critical applications on multicores remains challenging due to the inter-core interferences on shared platform resources and the difficulty in bounding their effect on the execution time of real-time applications.

One approach is to extend the notion of strict temporal and spatial partitioning of the ARINC-653 standard for single-core systems to multicores. Several works [14, 24] attempt this. Fisher [10]

proposed a (certified) operating system for partition scheduling. Such approaches are restricted to platforms with hardware and OS support for partitioning, unlike adaptive temporal partitioning which achieves it through scheduling, and they often result in resource under-utilization. Anderson et al. [5, 15, 16] implemented the MC^2 framework with different scheduling policies for different criticality levels. This framework allows partitions with different criticality to run in parallel, however interference on shared platform resources is not formally bounded like in our work.

Another line of research allows applications of different criticalities to execute in parallel, but imposes restrictions on resource arbitration to bound the delays that tasks suffer due to contention. Yun et al. proposed software-based memory throttling and a memory bandwidth reservation scheme with online reclamation support [28]. Hardware modifications to shared memory controllers for mixed-criticality systems were proposed in [12]. The authors of [9, 27] implemented data partitioning to disjoint DRAM banks to minimize inter-core interference on bank arbiters. Furthermore, Kim et al. combined bank partitioning with shared last-level cache partitioning [15]. Such mechanisms ensure bounded interference among criticality levels. However, some approaches suffer from limited flexibility, e.g. [28], many require hardware support, e.g. [12], and finally, most approaches target bus-based systems-on-chip rather than hierarchical cluster-based architectures, like the MPPA-256.

A third approach for bounding interference on mixed-criticality multi-core designs is Isolation Scheduling (IS) [13]. The key idea is, instead of fine-grained resource arbitration, to only permit tasks of the same criticality to execute concurrently. This way, IS policies [4, 11, 13] avoid inter-criticality interference, while exploiting parallelism. Adaptive temporal partitioning is a special IS case, where task preemptions and migrations are not permitted. To the best of our knowledge, we present the first implementation of adaptive temporal partitioning on a many-core platform designed for timing predictability. From an implementation perspective, close to our work are the scheduling frameworks of [3, 19]. Unlike our work, the authors eliminate all possible sources of interference on shared resources by adopting spatial and temporal partitioning of cores and memory banks [19] and/or by dedicated execution models with computation and resource access phases [3]. Although the scope of these works is more general, as they utilize several compute clusters of the MPPA-256, eliminating contention on shared resources can have a significant effect on schedulability. Our work shows that a high utilization is feasible by allowing contention among same-criticality applications.

3 SYSTEM AND SCHEDULING MODEL

First, we present the considered application model in general without any constraints on the temporal partitioning. We continue with an explanation of the adaptive temporal partitioning scheduling policy and summarize the task mapping optimization used in this work.

3.1 Mixed-Criticality Task Model

We consider a mixed-criticality task set $\tau = \{\tau_1, \ldots, \tau_n\}$ executed on a shared-memory multi-core architecture. We define for simplicity a dual-criticality system, where the criticality levels are denoted as high (HI) and low (LO) [2], although our scheduling framework supports more than two criticality levels. Each task $\tau_i \in \tau$ is characterized by a tuple $\tau_i = \{T_i, D_i, \chi_i, C_i(\text{LO}), C_i(\text{HI})\}$, where $T_i, D_i \in \mathbb{N}^+$ denote the period and relative deadline, $\chi_i \in \{\text{LO}, \text{HI}\}$ the criticality level and C_i, the execution profile. $C_i = (e_i, \mu_i)$ consists of an upper bound on the task's execution time (e_i) and the number of shared memory accesses (μ_i). The execution time e_i is specified when τ_i runs in isolation, i.e. *without* considering the delay it may experience due to contention on the shared memory. The two parameters of the execution profile (e_i and μ_i) can be obtained by static analysis tools [1] or estimated by profiling and measurement [18]. Each task has two execution profiles, at

different assurance levels [25]. For high-criticality tasks, the HI execution profile C_i(HI) is more conservative compared to the LO execution profile C_i(LO) since at a higher assurance level more stringent safety guarantees need to be provided.

The dual-criticality system can operate in two modes. In LO (default) mode of the system, all tasks are scheduled according to their LO execution profile C_i(LO). If at least one high-criticality tasks runs according to its HI-level profile C_i(HI), e.g. because the functionality requires more computation time or memory accesses, from then on the system switches (temporarily or permanently) to HI mode. In HI mode, high-criticality tasks require more resources, i.e. if χ_i = HI: e_i(HI) $\geq e_i$(LO) and μ_i(HI) $\geq \mu_i$(LO). Low-criticality tasks may need to execute in degraded mode, i.e. with reduced functionality, to preserve the system schedulability if the system is in HI mode. Execution in degraded mode is specified by $C_{i,deg} = C_i$(HI), i.e. if χ_i = LO: e_i(HI) $\leq e_i$(LO) and μ_i(HI) $\leq \mu_i$(LO).

In this work, the tasks of one criticality level are grouped together and are executed separated from tasks of different criticality level(s) (explained in more detail in Section 3.2). For simplicity, we assume that the first job (task instance) of all tasks is released at time 0 and that the relative deadline of τ_i is equal to its period, i.e., $D_i = T_i$. Precedence constraints may exist among tasks with equal periods and criticality levels as long as the resulting dependencies are acyclic. Finally, we define the total utilization of a periodic dual-criticality task set τ as the maximum utilization across LO and HI execution mode:

$$U = \max \left\{ U_{\text{LO}}^{\text{LO}}(\tau) + U_{\text{HI}}^{\text{LO}}(\tau), U_{\text{LO}}^{\text{HI}}(\tau) + U_{\text{HI}}^{\text{HI}}(\tau) \right\} \tag{1}$$

where $U_{\text{x}}^{\text{y}}(\tau)$ represents the total utilization of the tasks with criticality level x for their y–level execution time (in isolation), i.e. $U_{\text{x}}^{\text{y}}(\tau) = \sum_{\chi_i=\text{x}} e_i(y)/T_i$.

3.2 Adaptive Temporal Partitioning

Adaptive temporal partitioning is a special case of Isolation Scheduling (IS) [13] for co-hosting tasks of different criticality levels, e.g. mixed-criticality task sets on shared-resource multicores, such that no timing interference among tasks of different criticality levels exists [4, 11, 13]. The rest of the paper is based on a representative policy for adaptive temporal partitioning, namely Flexible Time-Triggered and Synchronization-based (FTTS) scheduling [11], as it is one of the few non-preemptive IS policies and hence can be implemented on the MPPA-256 whose cores do not support context switching/multitasking.

An FTTS schedule repeats over a *scheduling cycle* P equal to the hyperperiod of the tasks in τ. A simple example with two scheduling cycles is depicted in Figure 1. Each cycle consists of fixed-size *frames* (set $\mathcal{F}$) which start periodically at predefined time points. Each frame is divided further into two flexible-length *sub-frames*, the first containing only high-criticality tasks (HI sub-frame) and the second containing only low-criticality tasks (LO sub-frame). The beginning of frames and sub-frames is synchronized among all cores. Within a frame, the HI sub-frame begins immediately. The LO sub-frame begins once all tasks of the HI sub-frame complete execution across all cores. Synchronization for switching from the HI to the LO sub-frame is achieved dynamically via a barrier mechanism, for efficient resource utilization. Namely, if the HI sub-frame completes earlier than statically expected, the platform can be used by the tasks of the following criticality level, without wasting resources. Within the sub-frames, tasks are scheduled sequentially on each core following a predefined order. The mapping of tasks to cores and sub-frames is determined offline, see Section 3.3.

We use $sfLength(f, k, \text{LO})$ (resp. $sfLength(f, k, \text{HI})$) to denote the worst-case length for the k-th sub-frame of frame $f \in \mathcal{F}$, when the tasks execute in their LO (resp. HI)-level execution profile.

At runtime, the FTTS scheduler monitors the actual length of the sub-frames. If the length of the HI-criticality sub-frame does not exceed $sfLength(f, 1, \text{LO})$, it triggers normal execution

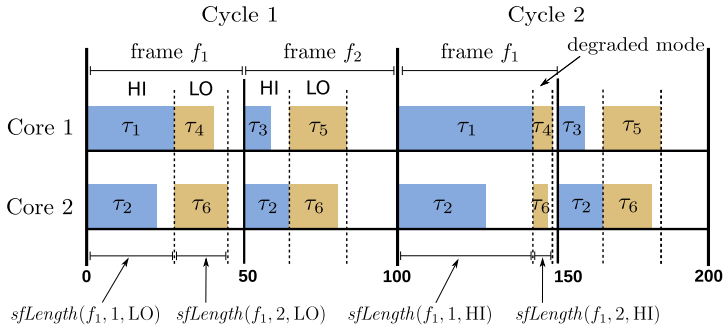


Fig. 1. Two consecutive FTTS scheduling cycles ($P = 100$), with 2 frames ($L_{f_1} = L_{f_2} = 50$) divided into flexible-length HI and LO sub-frames. Frame f_1 runs in LO mode in the first cycle and in HI mode in the second cycle. (For the ease of illustration, we chose to only have 2 frames in each scheduling cycle in this figure.)

of tasks in the LO-criticality sub-frame (LO mode). However, if the length of the HI-criticality sub-frame exceeds $sfLength(f, 1, \text{LO})$, the tasks of the LO sub-frame are triggered in degraded mode (HI mode). A LO to HI mode switch affects the execution of LO-criticality tasks only for the current frame. Note that if an overrun of a single task in the HI-criticality sub-frame occurs but the HI-criticality sub-frame does not exceed $sfLength(f, 1, \text{LO})$, the HI-criticality sub-frame does not overrun and it therefore does not affect the execution mode of the following LO-criticality sub-frame. After the completion of all tasks in the LO-criticality sub-frame the cores remain idle until the next frame starts.

We consider a schedule *feasible*, i.e. all tasks are guaranteed to meet their deadlines, if in every frame $f \in \mathcal{F}$ (with fixed length L_f), the last sub-frame completes by the end of the frame in either LO or HI mode (result from [11]), i.e. if $\forall f \in \mathcal{F}$:

$$\max \left\{ \begin{array}{l} sfLength(f, 1, \text{LO}) + sfLength(f, 2, \text{LO}), \\ sfLength(f, 1, \text{HI}) + sfLength(f, 2, \text{HI}) \end{array} \right\} \leq L_f \tag{2}$$

Finding a feasible FTTS schedule for a given task set τ can be performed by the optimization framework of [11] which we summarize in Section 3.3. A method to determine *sfLength* is presented in Section 6.

Note that the deployment of FTTS on a multi-core platform requires support for global time synchronization for time-triggered frame activation, inter-core barrier synchronization, inter-core communication for the implementation of dynamic scheduling decisions and static per-core schedule tables. Efficient and predictable implementations of scheduling primitives are examined in this paper.

3.3 Task Mapping Optimization

In this section, we shortly review the optimization framework that is used to determine the FTTS schedules offline and which has been presented in [11]. The input of the optimization framework consists of the specification of the following components: task set (period, deadlines, criticality levels, worst-case execution profiles (LO and HI)[1] when the tasks run in isolation) of all tasks in the task set, memory interference model (including bounds) of the shared memory architecture, bound on the scheduling overheads (see Section 5), and number of active cores.

[1]See Section 7.1 for an exemplary definition of the execution profile parameters which we use in our case study.

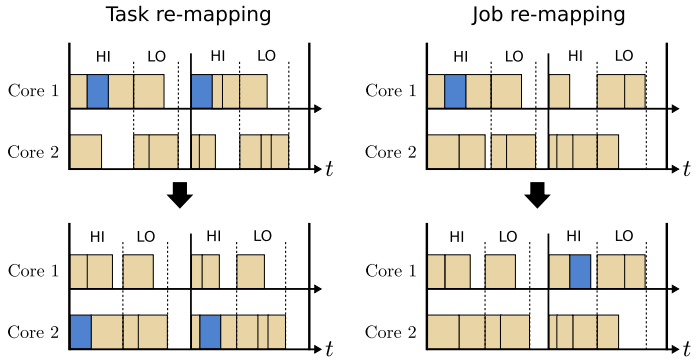


Fig. 2. The two methods of the optimization framework to generate variations of the current solution.

As a result, the optimizer provides FTTS schedules which define the spatial and temporal partitioning of the tasks. The spatial partitioning specifies the mapping of each task to a core and the temporal partitioning defines the mapping of each job to a sub-frame and the execution order of all jobs inside a sub-frame. We use the resulting schedules to execute the task sets in our case study in Section 7.

The optimization targets a solution which satisfies the real-time requirements of all the tasks in both LO and HI mode, respects the dependencies between tasks, and for which the workload is balanced among the cores. Thus, the main objective is to minimize the worst-case sub-frame length of all sub-frames.

The FTTS cycle length and the FTTS frame length can be determined by three methods: (i) they can be chosen by the developer, (ii) they can be determined based on the input of the optimizer or (iii) they can be subject to optimization themselves. For simplicity, in our work, we use the second method, i.e. the FTTS cycle length is equal to the hyperperiod (the least common multiple of the periods) of the tasks. The frame length is the greatest common divisor of the periods of the tasks.

Technically, the optimization is based on simulated annealing. First, a random solution that satisfies the constraints is generated. Starting with this first solution, the design space is explored. For this, a variation of the current solution is generated by using randomly one of the following two methods (depicted in Figure 2):

- **Task re-mapping:** All jobs of a randomly selected task are re-mapped to a randomly selected different core.
- **Job re-mapping:** One job of a randomly selected task is re-mapped to a different sub-frame or to a different position in the same sub-frame.

The final solution is the solution with minimal worst-case sub-frame length out of all solutions explored in the simulated annealing process. It is not guaranteed that the final solution actually satisfies the real-time requirements. Therefore, the optimization framework marks the solution as feasible or infeasible, as defined by Equation (2).

4 KALRAY MPPA-256

We present an overview of the Kalray MPPA-256 with emphasis on the memory system, which is important for the timing analysis in Section 6. More details on the MPPA-256 architecture and runtime environment can be found in [7].

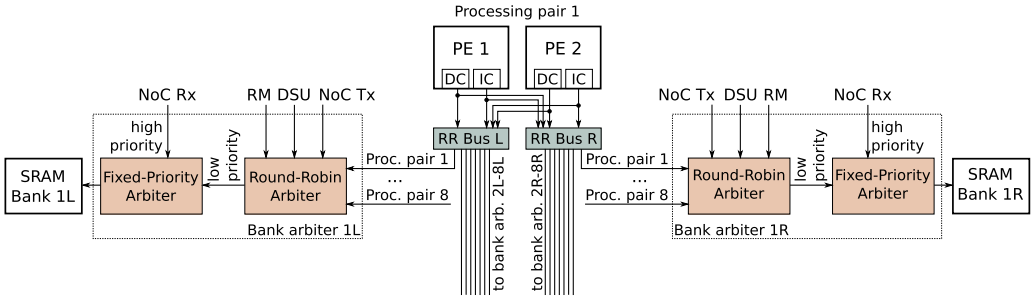


Fig. 3. Memory path from one processing pair to one memory bank on the left or right side of an MPPA-256 Andey compute cluster.

Architecture: The Kalray MPPA-256 Andey processor integrates 256 processing cores (PE) and 32 resource management (RM) cores, which are distributed across 16 compute clusters and four I/O sub-systems. Inter-cluster communication is supported by a network-on-chip (NoC).

A compute cluster includes 16 processing cores and one resource management core, each with a private instruction and data cache. Each of the caches is 2-way set associative and has a size of 8 KB. All cores implement the same VLIW architecture and execute at 400 MHz. Each cluster has a local 2 MB SRAM memory, which is organized in 16 independent banks with a dedicated access arbiter for each bank. The memory banks are arranged in two sides (left, L and right, R) of 8 banks. Each memory bank arbiter is connected to 12 masters: the NoC Rx interface, the NoC Tx DMA engine, a debug support unit (DSU), the resource management core and 8 processing pairs. A processing pair consists of two neighboring processing cores. As shown in Figure 3 (for simplicity, only for processing pair 1), the access path from a processing core to a memory bank passes through three request arbiters. The path starts with a bus shared by the cores of the processing pair. Access to this bus is arbitrated round-robin among the two data caches (DC) and the two instruction caches (IC) of the cores. Each processing pair has two buses, one for each side. This means that if the two cores of the processing pair need to access simultaneously two memory banks on different sides, there is no interference on the bus level. The buses of the processing pairs, along with all other masters are connected to the bank arbiters, which implement a non-preemptive round-robin arbitration scheme with higher priority for NoC Rx (illustrated by the two-stage arbitration on Figure 3).

Runtime Environment: The runtime environment used in this work (Accesscore 1.4) supports two programming modes for the cores of a compute cluster. *OS* mode features a light-weight operating system and precompiled POSIX libraries for thread management and synchronization. Bare-metal mode (*BM*) offers complete low-level control of the platform to the programmer, yet at the cost of increased programming effort. In OS mode, the code is launched on processing core PE1 of a cluster. This core can then spawn POSIX threads on the remaining 15 compute cores, PE2 to PE16 (maximum one thread per core). The resource manager of each cluster hosts the operating system and is involved in the management of interrupts, NoC access requests and system calls. In BM mode, the code is automatically launched on the resource management core of the cluster, so all 16 processing cores PE1 to PE16 can be used for application execution.

5 IMPLEMENTATION OF FTTS SCHEDULING PRIMITIVES

The first goal for a successful deployment is to implement the scheduling primitives with *bounded* and *low* overhead. In this section, we first discuss the basic components and properties of our runtime environment followed by an evaluation of different methods to implement the scheduling primitives.

Table 1. Overview of the Different Implementation Approaches for the Three Scheduling Primitives

	OS	BM
Time-triggered activation	`nanosleep`	custom busy–wait
	`cond_timed_wait`	
	custom busy–wait	
Barrier synchronization	`pthread_barrier_wait`	custom barrier sync
Communication of scheduling decision	`pthread_barrier_wait` + shared variable	custom barrier sync + shared variable

5.1 Overall Structure

Our runtime environment is implemented in the OS as well as in BM programming mode in order to investigate the overhead of each mode. The runtime environment features a *scheduler thread*, mapped on processing core 1 (PE1) of a compute cluster in OS programming mode (resp. on the resource management core in BM mode), and 1 to 15 (resp. 1 to 16) *worker threads*, mapped on processing cores PE2 to PE16 in OS mode (resp. PE1 to PE16 in BM mode). The scheduler thread is responsible for enforcing the time pattern (starting times of FTTS frames and cycles) of an FTTS schedule (timekeeping), synchronizing the beginning of each FTTS frame with the worker threads, performing dynamic decisions at runtime and communicating them to the worker threads. The FTTS schedule is stored in the shared memory of the cluster and can be read by all threads running on the cores of the cluster.

The worker threads are responsible for executing sequentially the functions that implement the task functionality in each FTTS sub-frame and synchronizing with the scheduler thread when all functions are executed. Each worker thread has access to its schedule table, hence activation of individual task functions is managed without involving the scheduler. The execution of every task function is followed by a data cache flush to ensure cache coherence. The memory address mapping is interleaved, since all threads share libraries and scheduling information and shared-memory inter-task communication is supported, thus making the benefits of bank privatization under sequential address mapping unattainable.

The runtime environment, including a generic empty task, occupies approximately 890 KB out of 2 MB of the cluster memory (OS or BM). The remaining 1150 KB can be used by the tasks that run on the worker threads.

Adaptive temporal partitioning, such as FTTS, is based on three main primitives: (i) the enforcement of a predefined time pattern within a scheduling cycle, (ii) the dynamic synchronization among all cores, e.g. upon completion of their tasks in a partition, (iii) the communication of dynamic decisions of the scheduler to all cores. In the following, we examine different implementations on the MPPA-256 and the corresponding overheads for each of the three necessary scheduling primitives. Table 1 provides an overview of the different implementation approaches.

5.2 Time-Triggered Activation of FTTS Frames

In general, timekeeping can either be done by a scheduler which notifies the worker threads, when to start the execution using barrier synchronization, or timekeeping is done by the individual worker threads which start the next frame independently, based on their schedule. The second approach exhibits less overhead since it does not need barrier synchronization. On the other hand, the clocks of the worker thread cores need to be synchronized and it is more challenging to handle frame overruns. In our implementation we use a scheduler that does timekeeping. In case of a frame

overrun the activation of the next frame is automatically delayed with the barrier synchronization until the overrun task completes execution.

During the time period starting with the end of the last sub-frame of the previous frame until the beginning of the new frame, the scheduler has to wait. We examined the following methods to do this:

OS: For OS mode we examined (i) the POSIX function `nanosleep`, which is called after the completion of the last sub-frame with the remaining time until the end of the frame as input argument, (ii) the POSIX function `cond_timed_wait` which is configured at the beginning of the frame with an absolute value indicating its exact completion time, and (iii) a custom busy-wait function which is called upon completion of the last sub-frame and performs assembler `nop` operations for the remaining time until the next frame starts.

BM: In BM mode the POSIX functions are not available and therefore we only implemented the custom busy-wait function.

The call of the POSIX functions often leads to the scheduler "waking up" hundreds of cycles later than expected. In contrast, the custom busy-waiting approach is highly accurate, resulting in negligible offsets (in the order of tens of clock cycles as estimated based on measurements) from the expected frame completion time. Therefore, we use the custom busy-wait implementation for OS and BM programming mode.

5.3 Barrier Synchronization

We use barrier synchronization to synchronize between the scheduler and the worker threads. In our case, a thread has to stop at a barrier until all other threads, i.e. scheduler and worker threads, have reached the barrier. A barrier synchronization is performed at the beginning of the HI-criticality sub-frame and upon completion of each sub-frame. Our runtime environment supports two alternative implementations:

OS: In OS mode we make use of the POSIX function `pthread_barrier_wait`.

BM: In BM mode, we implemented a custom function using hardware event lines which connect every processing core directly to the resource management core of a cluster.

Figure 4(a) and Figure 4(b) depict the statistical distribution (box-plot) of the overhead of the two implementations as a function of active cores, based on measurements. For the measurements we used an FTTS schedule, consisting of two frames, with two sub-frames each. In each sub-frame, one or two instances of a single task run on every worker thread. The task performs busy waiting for 2 μs. The FTTS schedule was executed on the MPPA-256 for 100,000 scheduling cycles (in total 200,000 frames). For each FTTS sub-frame, we measured the interval between the completion of the last task of the sub-frame and the completion of barrier synchronization as seen by the scheduler thread (o_{sync} in Figure 5). Repetitions of the experiment with other schedules (with different busy waiting interval or different tasks from Section 7) produced similar results. Therefore, we consider it a realistic upper bound for the barrier synchronization overhead in our experiments. Note that the overhead of the custom implementation (BM) is an order of magnitude lower than that of the POSIX function call (OS).

5.4 Communication of Scheduling Decision

This operation is performed upon completion of the HI sub-frame in a frame. After barrier synchronization, the scheduler checks the elapsed time since the beginning of the HI sub-frame. If the sub-frame duration surpasses a statically determined value (function *sfLength*), the scheduler notifies all worker threads to run the tasks of the following LO sub-frame in degraded mode. Our runtime provides two alternative implementations for this notification:

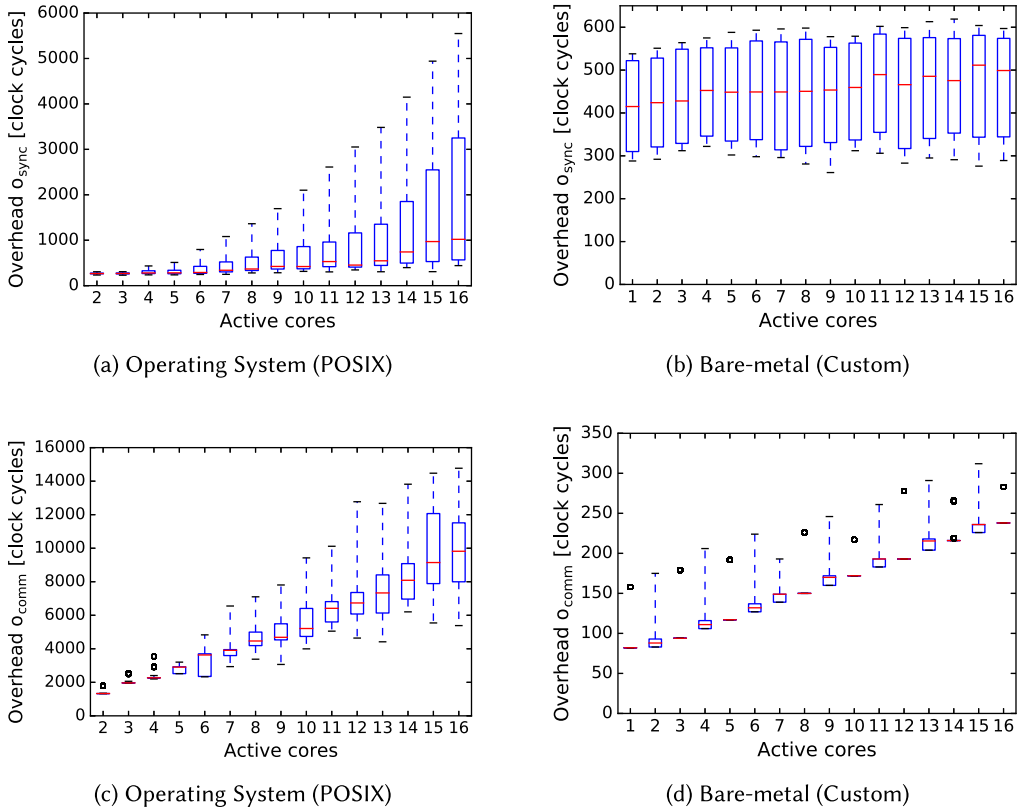


Fig. 4. Runtime overheads of scheduling primitives on MPPA-256 (200,000 measurements). On each box, the central mark, bottom and top edges indicate the median, 25th and 75th percentiles, respectively. The whiskers extend to 10 times the interquartile range. Outliers are denoted with circles.

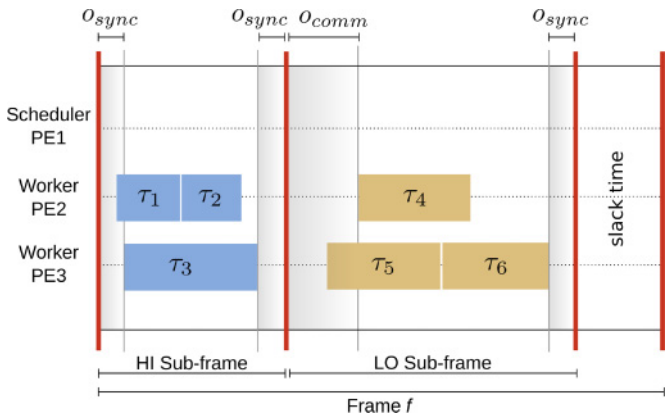


Fig. 5. Timing diagram for one FTTS schedule frame.

OS: In OS mode, we use the POSIX function `pthread_cond_wait` along with conditional variables and mutex locks for blocking the worker threads until a decision is made by the scheduler.

BM: In BM mode, we implement the functionality using a custom function which makes use of the hardware event lines. The same mechanism is used as for barrier synchronization in BM mode.

The scheduler writes the decision in a global variable in memory and broadcasts a signal to all processing cores afterwards. Due to the lack of hardware support for cache coherence, the data structures that are shared between the scheduler and the worker threads as well as other data structures that can be modified at runtime are non-cachable, i.e. they are loaded/stored by bypassing the local cache.

The statistical distribution (box-plot) of the overhead of the two implementations for the same experiment as before (over 100,000 scheduling cycles of the FTTS schedule), is depicted in Figure 4(c) and Figure 4(d). The overhead is defined as the measured interval between the completion time of barrier synchronization at the scheduler thread and the latest starting time of a task in the LO sub-frame (o_{comm} in Figure 5).

The difference is significant, with the OS implementation having up to 52 times higher maximal overhead than the BM implementation. One possible factor for the higher overhead is related to the data cache flush, which automatically follows the call of POSIX synchronization functions due to the lack of hardware cache coherence protocols. In BM mode, such a flush operation is unnecessary since the scheduler writes explicitly its decision in shared memory (bypassing the cache) and similarly, the worker threads read it directly from the memory. Note also the variance of the measured OS overhead, spanning a range of more than 9000 clock cycles for 16 active cores.

6 WORST-CASE TIMING ANALYSIS

The previous section focused on the first goal which is addressed in this work, i.e. the ability to (experimentally) bound the overhead of scheduling primitives. In this section, we investigate also the second goal, i.e. the ability to bound the worst-case delays that tasks experience due to memory contention. The fact that MPPA-256 fulfills [7] the property of timing compositionality [26] enables bounding the delays due to contention on the shared cluster memory.

In the following, we propose a method for the computation of function $sfLength$, i.e. for the offline estimation of the worst-case sub-frame lengths of a given FTTS schedule. The timing analysis accounts for the runtime overheads of the scheduling primitives and for the task execution delays on the shared memory path of an MPPA-256 cluster. In order to determine the $sfLength$ function, the following parameters are required to be known: the tasks' profiles (measured in isolation), the worst-case delay of the memory path and the scheduling overheads.

6.1 Impact of Runtime Scheduling Overheads

Figure 5 presents a timing diagram of an FTTS schedule for a single frame with two sub-frames. The thick lines indicate the start and the completion of barrier synchronization as seen by the scheduler thread at the beginning of the frame, at the end of each sub-frame and at the end of the frame. We identify the following runtime overheads that have an impact on the actual length of the FTTS sub-frames: (1) *barrier synchronization* denoted as o_{sync} and (2) *communication of scheduling decision* denoted as o_{comm}. Barrier synchronization overhead must be considered in both sub-frames, since barrier synchronization is used to signal the start (HI sub-frame only) and to detect the completion of the task execution (HI and LO sub-frame). The communication overhead includes the time to compute and store the scheduling decision in a shared variable, the time to signal the start of the task execution and the time to read the scheduling decision by all computing threads. It has be considered only in the LO sub-frame.

Let $\text{WCET}_i(\ell)$ denote the worst-case execution time of a task at profile $\ell \in \{\text{LO,HI}\}$, when it executes in parallel with other tasks (considering all possible interference delays), m the number of active cores in an FTTS schedule, and $S(c, f, k) \subseteq \tau$ the set of tasks executing on core c in the k-th sub-frame of frame f. The worst-case length of sub-frame $k \in \{\text{LO,HI}\}$ in f can be expressed as:

$$sfLength(f, k, \ell) = o(k) + \max_{1 \le c \le m} \left\{ \sum_{\tau_i \in S(c,f,k)} \text{WCET}_i(\ell) \right\} \tag{3}$$

where the overhead $o(k)$ is defined by

$$o(k) = \begin{cases} 2 \cdot o_{sync} & \text{if } k \equiv \text{HI} \\ o_{sync} + o_{comm} & \text{if } k \equiv \text{LO} \end{cases}$$

6.2 Impact of Interference on Shared Memory

The problem of bounding the worst-case execution time of tasks under FTTS scheduling and memory contention on the MPPA-256 was investigated in [11]. However, their shared memory model differs from the model of Section 4 by not accounting for timing interference on the shared bus between the cores of a processing pair. Here, we extend the timing analysis of [11] to account for this additional source of interference and guarantee safe execution time bounds (assuming that the task execution profiles are safe, too). Note that Skalistis et al. [21] used a similar memory model as in Section 4 for WCET analysis, yet assuming dataflow-based scheduling, where certain tasks cannot interfere due to precedence constraints. For our analysis, we assume enabled data caches and interleaved memory address mapping, which is the default configuration in our runtime environment. The latter leads to conservative WCET bounds, which can be refined if sequential address mapping is applied with a known allocation of data to memory banks. Additionally, we assume that there are no memory access requests generated by the NoC Rx, NoC Tx, the debug support unit or the resource management core of a compute cluster during the task execution phase. In our framework we achieve this by not using the corresponding components.

Under these assumptions and given the memory model of the MPPA-256 Andey architecture, every memory access request of a task can be delayed at two points: (i) on the shared bus of the processing pair where the task is executed, by pending requests from the other three caches (in the worst-case, all requests target banks on the same side), (ii) on the round-robin arbiter of the target bank, by pending requests from the other seven processing pairs (see Figure 3).

Following the previous discussion, the $\text{WCET}_i(\ell)$ of task τ_i which runs in processing pair $1 \le p \le 8$ can be upper-bounded by:

$$\text{WCET}_i(\ell) = e_i(\ell) + \mu_i(\ell) \cdot \left(N_{caches}(p) \cdot \sum_{j=1}^{8} active(j) - 1 \right) T_{acc} \tag{4}$$

where the first term is the worst-case execution time and the second term specifies the worst-case delay due to interference on the memory path. $N_{caches}(p)$ denotes the number of active caches (data and instruction caches) in the processing pair p to which τ_i is mapped. If there is at least one task running on the neighboring core in the same sub-frame as τ_i, then $N_{caches}(p) = 4$, otherwise $N_{caches}(p) = 2$. Function $active(p)$ returns 1 if there is at least one task running in processing pair p in the same sub-frame as τ_i. $N_{caches}(p)$ and $active(p)$ are derived based on the FTTS schedule which is obtained by the task mapping optimizer. On the MPPA-256, in the worst-case all four caches of the processing pair are active ($N_{caches}(p) = 4$) and at least one core is active ($active(p) = 1$) in each of the seven interfering processing pairs. In this case, the term in the brackets in Equation (4) evaluates to 31 which means that a memory access is delayed by 31 other

memory accesses until it can be served in the worst-case. Finally, T_{acc} is an upper bound on the latency of a single memory access (under no contention).

Recall that the execution profile $C_i(\ell)$ of a task τ_i contains its worst-case execution time $e_i(\ell)$ when τ_i runs in isolation and the maximum number of memory access requests $\mu_i(\ell)$ that it can generate. The corresponding numbers for the different execution profiles of our benchmark tasks are discussed in Section 7.1.

If several jobs of τ_i are scheduled in different sub-frames of an FTTS schedule, then multiple $\text{WCET}_i(\ell)$ values exist depending on the tasks that run in parallel in each sub-frame. By using these values in Equation (3), we can bound the worst-case sub-frame lengths in LO and HI execution mode and validate the feasibility of any given schedule according to Equation (2).

Empirical Estimation of Single Memory Access Latency. In order to derive the memory access latency bound T_{acc} on the Kalray-MPPA 256 Andey processor, different assembly code snippets were executed in bare-metal mode with interleaved memory address mapping. The execution time of the code was measured using a hardware cycle counter. All experiments were run twice in a row, with a first run to load all code into the instruction cache and the second run for the actual measurement.

We conclude from the performed benchmark that a memory access has a latency of 10 cycles, however loading an entire cache line requires 14 cycles. If the accesses are executed too quickly in a row (less than 4 cycles between two accesses), the processor stalls until the last cache line operation is finished, i.e. for 14 cycles. In this way, we derived the memory access latency $T_{acc} =$ 14 cycles, which is used in the empirical evaluation (Section 7).

7 CASE STUDY

This section presents the experiments that were performed on the MPPA-256 to empirically validate the worst-case timing analysis of Section 6 and to evaluate the schedulability loss due to the temporal partitioning (FTTS) constraints, the scheduling overheads and the mutual delays of co-running tasks on the shared memory.

The experimental framework enables: (1) specification of a mixed-criticality application, (2) automatic generation of a feasible FTTS schedule, as defined by Equation (2) (if one exists), (3) deployment of the FTTS schedule on an MPPA-256 cluster for a given number of scheduling cycles, (4) post-processing and statistical analysis of collected profiling data. The framework that executes the FTTS schedule repeatedly on the MPPA-256 and collects and post-processes profiling data has been built upon an existing C++ runtime environment for static dataflow applications [22].

The timestamps for profiling are collected by the scheduler thread at the beginning and end of each frame and sub-frame and by the worker threads at the beginning and end of each executed task. The timestamps are obtained by using the performance and monitoring counter (PMC). The offset which the PMC counters of the different cores exhibit among each other is compensated by measuring and subtracting the offset from the obtained timestamps.

7.1 Benchmark Applications

For the evaluation, we used the following four different sets of benchmarks:

- The flight management system (FMS) [8] is a safety-critical application. We used an industrial-representative implementation of an FMS sub-set, consisting of one task (sens_c1) which periodically reads (hard-coded) sensor data and four tasks (loc_cx) which compute the current aircraft location based on the data. The tasks communicate through shared memory, but their read/write operations are non-blocking. This means that they read the previous data if no new data is available.

Table 2. Specification and Execution Profiles of the Benchmark Tasks

App	Task τ_i	Crit. χ_i	Period T_i (ms)	Execution Cycles e_i	DCache misses μ_i
FMS	sens_c1	HI	5	14803	105
	loc_c1	HI	5	8335	85
	loc_c2	HI	40	2361	43
	loc_c3	HI	40	11134	95
	loc_c4	HI	40	2241	53
ROSACE	engine	HI	5	1294	9
	elevator	HI	5	1280	11
	aircraft_dynamics	HI	5	9232	34
	h_filter	HI	10	1302	13
	az_filter	HI	10	1341	12
	Vz_filter	HI	10	1334	12
	q_filter	HI	10	1289	12
	Va_filter	HI	10	1321	12
	altitude_hold	HI	20	1269	9
	Vz_control	HI	20	1275	12
	Va_control	HI	20	1303	11
StreamIt	matmult	LO	5	56477	157
	fft	LO	5	19668	103
	bitonic_sort	LO	5	200198	95
	insertion_sort	LO	5	77203	135
	dct_2D_coarse	LO	5	59833	138
	idct_2D_coarse	LO	5	56222	168
	fm	LO	5	36726	889
	filter_bank	LO	5	1540483	240
	autocorrelation	LO	5	2896	13
Synth.	busy_wait_HI	HI	5	80079	7
	busy_wait_LO	LO	5	1600061	7

- The ROSACE application [17, 19] is an open-source avionic benchmark, implementing a longitudinal flight controller. It consists of three tasks that perform environment simulation (pilot instructions) and eight tasks that implement the controller logic.
- For the class of lower-criticality applications, we used nine benchmarks from the StreamIt suite [23] since most of them represent computation- and memory-intensive applications.
- Finally, we implemented two synthetic tasks, which simply perform busy waiting for 200 μs and 4 ms, respectively.

All benchmarks were implemented such that their code and data fit into the memory of a compute cluster, without the need to access the memory space of other compute clusters or I/O subsystems. Their inputs are set during an initialization phase prior to their first execution.

The specification of all tasks is presented in Table 2. The task periods were assigned according to their specification for high-criticality tasks (the FMS periods being down-scaled in order to get hyperperiods of acceptable length) or randomly for low-criticality tasks. The execution profiles were obtained through measurements on the MPPA-256. Table 2 presents the *maximum* execution time and cache misses over 10,000 executions of each benchmark in isolation. The statistical

distribution of the measured values is presented in Section 7.2. The worst-case parameters specify the LO-level execution profile C_i(LO) of all tasks. For high-criticality tasks (χ_i = HI), we assume C_i(HI) = C_i(LO) and for low-criticality tasks (χ_i = LO), we assume C_i(HI) = ($e_i = 0$ and $\mu_i = 0$).

7.2 Profiling of Benchmarks

For the derivation of the execution profiles of the benchmarks used in Section 7, we performed extensive measurements of the tasks' execution times and data cache misses, with the tasks running in isolation. Specifically, we used FTTS schedules, where either a single task or a sequence of tasks under analysis ran in every sub-frame on a single core. Every execution is followed by a cache flush. The assumption that the tasks start with an empty cache is realistic since the cache is flushed during the schedule execution in the experiments in Section 7.3, too.

When constructing the FTTS sub-frames, we took into account that some of the tasks exhibit their worst-case execution time depending on the output of previously executed tasks (implicit data dependencies) by considering all possible execution scenarios. The FTTS schedules were executed for at least 10,000 scheduling cycles, and the maximum observed execution time and data cache misses in Table 2 were extracted from the collected profile data. Figure 6 presents the statistical distribution of the execution time and cache miss measurements over 10,000 executions in the form of a box-whisker-plot. Note that the measured execution times include the cost of data cache flush, which takes place at the end of each task execution, for data coherence. This operation costs approximately 1100 cycles. Execution time variance is relatively low for all benchmarks except "fm", which executes for 4.53 times its mean execution time on every 1000th execution. Cache miss variance is also low or zero for most benchmarks, with the exception of "idct2DCoarse" and "fm". The large number of iterations, over which the profiling data was collected, and the consideration of different inputs for tasks with input-dependent functionality provides confidence that the obtained maximal values represent reliable upper bounds.

7.3 Metrics

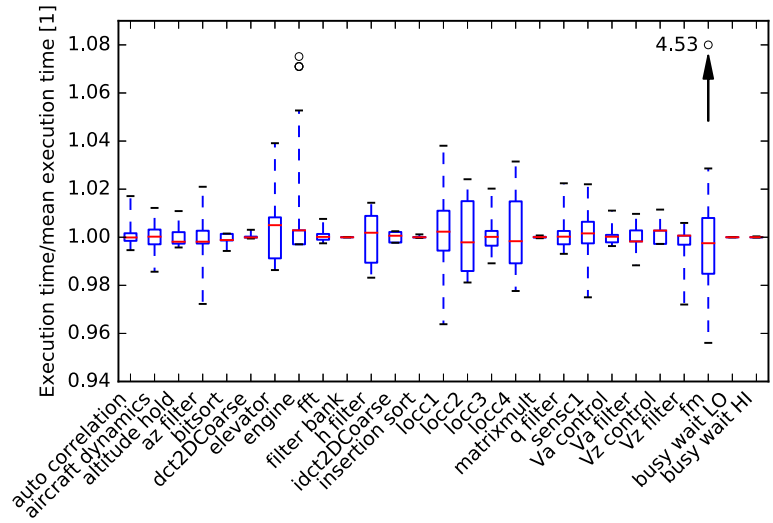
In the following, we experiment with several combinations of the benchmarks with a two-fold objective: (i) to validate whether the timing analysis of Section 6 bounds the FTTS sub-frame lengths at runtime, and (ii) to investigate the practical limits of our scheduling approach in terms of maximum achievable utilization. We evaluate every deployed FTTS schedule based on two criteria:

Number of frame violations: It is defined as the portion of FTTS frames in which the last sub-frame was not completed by the end of the frame. If a frame violation happens for a *feasible* schedule, this implies that either our worst-case timing analysis is incorrect or the considered upper bounds for scheduling overheads, single memory access latency, task execution and accessing parameters are not safe. With our experiments, we cannot guarantee safe parameters since *all* of them were acquired through measurements. For the deployment of a safety-critical system, more rigorous methods, such as static analysis, need to be applied.

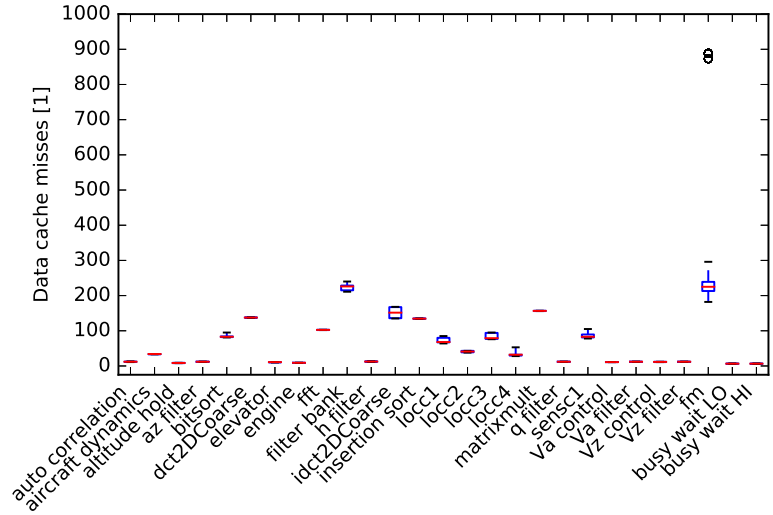
Availability: It expresses how many computational resources are available, if needed for incremental design, and it is defined as:

$$A = (16 - N_a) + N_a \cdot \frac{\sum_{\forall f \in \mathcal{F}} \left(L_f - \sum_{k=1}^{2} sfLength(f, k, \text{LO}) \right)}{P} \tag{5}$$

where N_a is the number of active cores (implementing the schedule) out of the 16 processing cores of an MPPA-256 cluster and P is the period of the FTTS scheduling cycle. Availability is a combined expression of the number of currently inactive cores and the aggregate portion of time when all active cores are idle, which happens in each frame between the completion of the last sub-frame and the frame end. In an ideal 16-core platform, without scheduling or memory



(a) Execution time vs mean execution time ratio. For ease of presentation, 150 outliers of "fm" at 4.53 have been omitted.



(b) Data cache misses

Fig. 6. Profiling of benchmarks: Statistical distribution of measured values over 10,000 executions in isolation. On each box, the central mark, bottom and top edges indicate the median, 25th and 75th percentiles, respectively. The whiskers extend to 10 times the interquartile range. Outliers are denoted with circles.

interference overheads and without the constraint of temporal partitioning, $A = 16 - U$ should hold for any application with utilization U as defined by Equation (1). By comparing the availability of our deployed FTTS schedules to the "ideal" availability, we get a measure of the schedulability loss due to the temporal partitioning constraint and the overheads of implementation on a real platform.

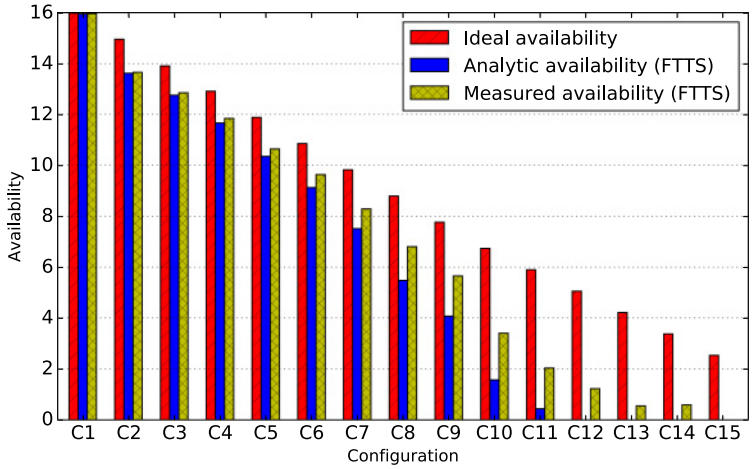
Table 3. Benchmark Configurations and Deployment on the MPPA-256. The FTTS Schedules with (*) were Deemed *Infeasible* at Design Time

Conf.	Task instances	U	Active cores OS	Active cores BM	Frame Viol. OS	Frame Viol. BM
C1	69	0.02	2	1	0%	0%
C2	141	1.05	3	2	0%	0%
C3	282	2.10	4	3	0%	0%
C4	399	3.09	5	4	0%	0%
C5	532	4.12	6	5	0%	0%
C6	665	5.15	7	6	0%	0%
C7	798	6.18	9	8	0%	0%
C8	931	7.20	11	9	0%	0%
C9	1064	8.23	12	11	0%	0%
C10	1197	9.26	15	13	0%	0%
C11	1213	10.10	16	15	0%	0%
C12	1229	10.94	16 (*)	16	0%	0%
C13	1245	11.78	16 (*)	16	0%	0%
C14	1261	12.62	16 (*)	16 (*)	0%	0%
C15	1277	13.46	16 (*)	16 (*)	100%	100%

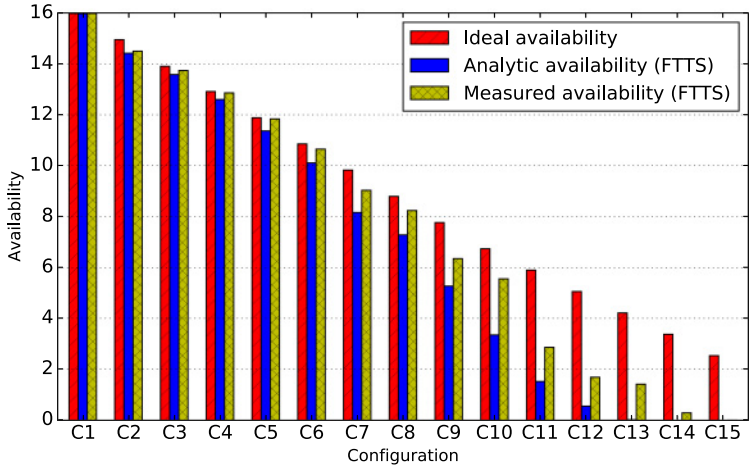
7.4 Configurations

We consider 15 experimental configurations, with different combinations and number of instances (replications) of the benchmarks. With every configuration, we increase the number of task instances in order to gradually increase the utilization of the system. We do this to evaluate the maximum achievable utilization with our framework. The goal of the task selection was to obtain configurations which include tasks with varying execution and communication demands as well as a large range of periods. Configuration 1 (C1) contains only the high-criticality benchmarks, FMS and ROSACE. Configurations 2 and 3 contains the FMS, ROSACE and StreamIt benchmarks, with one and two instances of each benchmark, respectively. Configurations 4–10 contain the FMS, ROSACE and StreamIt benchmarks except "fm", with three to nine instances of each benchmark. "fm" has been excluded due to its large memory footprint. Configurations 11–15 contain nine instances of the FMS, ROSACE, StreamIt benchmarks and respectively $\{1, 2, 3, 4, 5\}$ instances of the synthetic benchmarks, which have a very low memory footprint. The total number of task instances and the ideal utilization (as defined by Equation (1)) of each specific configuration are listed in Table 3. Columns 4 and 5 of the same table show the number of active cores which have been used as input for the FTTS optimization framework. The FTTS optimizer was able to find feasible FTTS schedules for configurations C1 to C11 for OS deployment and C1 to C13 for BM deployment. For the remaining configurations, it returned the best found solutions even if feasibility under the timing analysis of Section 6 could not be guaranteed. All FTTS schedules have a cycle period $P = 40$ ms, consisting of 8 frames with a fixed length of 5 ms each. Note that the OS schedule employs one more active core compared to the corresponding BM schedule. This is because in OS mode processing core PE1 is used exclusively by the scheduler thread, whereas in BM mode this core can be used for task execution.

Using the MPPA-256 runtime environment, we deployed the best found FTTS schedule (even if it was deemed infeasible by the optimizer) for each configuration and each execution mode (OS or BM) on one compute cluster.



(a) Operating System Implementation



(b) Baremetal Implementation

Fig. 7. Availability of different configurations on the MPPA-256.

7.5 Experimental Results

The last two columns of Table 3 show the ratio of frame violations detected during deployment. Figure 7 illustrates the availability metric. For comparison purposes, three computations of availability are depicted: (i) the "ideal" availability, considering no overheads, interferences or partitioning constraints, which is equal to $16 - U$, (ii) the "analytic" availability of an FTTS schedule, computed by the FTTS optimization framework Equation (5) based on our worst-case timing analysis, and (iii) the "measured" availability which corresponds to the minimum availability of a deployed FTTS schedule, based on the profile data of our runtime environment after 10,000 scheduling cycles. In the last case, L_f and $\sum_{\forall f \in \mathcal{F}} sfLength(f, k, \text{LO})$ in Equation (5) are substituted by the actual frame length and the completion time of the last sub-frame of a frame in each scheduling cycle.

Frame Violations: For the FTTS schedules that were deemed feasible offline, i.e. up to C11 (OS) and up to C13 (BM), there was *no* frame violation at runtime. Frame violations occurred only

for infeasible schedules at high utilizations (C14, C15 in Table 3). For feasible BM schedules, we detected in total 9 HI sub-frame overruns over 130,000 scheduling cycles (1,040,000 frames), leading to 9 skipped LO sub-frames (0.009‰). Sub-frame overruns occurred only in BM mode. We consider an unusually high overhead of the scheduling primitives in BM mode as the most probable cause. Note that the runtime reaction to the HI sub-frame overruns prevented frame violations.

By comparing the analytic versus measured availability in Figure 7, we observe that the pessimism of the timing analysis increases with increasing utilization, especially for OS schedules. Note that no feasible FTTS schedules could be found for configurations C12, C13 and C14 (OS), although the best found solutions were deployed successfully without any frame violations. Namely, the maximum achievable utilization with guaranteed schedulability is 10.10 (63.1%, C11), while a utilization of 12.62 (78.9%, C14) is practically possible without frame violations.

The pessimism in our timing analysis can stem from the over-estimation of (i) task execution parameters, (ii) task memory accesses, (iii) scheduling overheads or (iv) memory interference. We suspect that the scheduling overhead contributes a major part since the variance of the scheduling overheads is high. The increasing pessimism with increasing number of active cores can be explained partly by the increasing variance of the overhead parameters and the increasing memory interference.

Maximum Achievable Schedulability: In order to evaluate the schedulability loss that is caused by temporal partitioning and the runtime overheads/interferences, we compare the "ideal" to the empirical availability in Figure 7. As expected, the difference between the two metrics is higher for OS than for BM schedules. This is because in BM mode, there is one more processing core available for running a worker thread and the worst-case runtime scheduling overheads are significantly lower. These two factors lead to a higher availability of the platform, closer to the ideal bound. The maximum achievable utilization of 11.78 (73.6%, C13) with guaranteed schedulability or 12.62 (78.9%, C14) for a practically feasible solution is a significant result.

8 CONCLUSION

We presented a runtime environment for adaptive temporal partitioning on many-core platforms. Applicability was demonstrated based on the flexible time-triggered and synchronization-based (FTTS) scheduling policy for periodic mixed-criticality applications. We proposed alternative implementations of the scheduling primitives and evaluated them w.r.t. runtime overhead. Additionally, we proposed a worst-case timing analysis methodology for evaluating the feasibility of FTTS schedules, by accounting for scheduling overheads and the interference of co-running tasks on the shared resources of a compute cluster. Using industrial-representative benchmarks, we were able to validate runtime adherence to the analytic worst-case execution time bounds. We also showed that by optimizing the implementation of the scheduling primitives, an effective utilization of 73.6% (analytically guaranteed) or 78.9% (empirically feasible) can be achieved on the 16 cores of an MPPA-256 cluster. Adaptive temporal partitioning seems, therefore, a promising solution for efficient and predictable safety-critical many-core systems by enabling sufficient isolation among applications with different criticality, yet at a low schedulability cost.

ACKNOWLEDGMENTS

The authors would like to thank Thales Research and Technology, France for providing the FMS use case, which was used for evaluation of the runtime environment, Oded Maler and Pranav Tendulkar for access to runtime environment [22], and the anonymous reviewers for their constructive feedback.

REFERENCES

[1] AbsInt.2015. aiT Worst-Case Execution time Analyzers. https://www.absint.com/ait/. (2015).

[2] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo DAngelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. 2012. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE, 145–154.

[3] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, and T. Nolte. 2016. Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform. In *ECRTS*. 14–24.

[4] A. Burns, T. Fleming, and S. Baruah. 2015. Cyclic Executives, Multi-core Platforms and Mixed Criticality Applications. In *ECRTS*. 3–12.

[5] M. Chisholm, B. C. Ward, N. Kim, and J. H. Anderson. 2015. Cache Sharing and Isolation Tradeoffs in Multicore Mixed-Criticality Systems. In *RTSS*. 305–316.

[6] D. Dasari, B. Akesson, V. Nélis, M. A. Awan, and S. M. Petters. 2013. Identifying the sources of unpredictability in COTS-based multicore systems. In *SIES*. 39–48. DOI: https://doi.org/10.1109/SIES.2013.6601469

[7] Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhiès, and Guillaume Lager. 2014. Time-critical Computing on a Single-chip Massively Parallel Processor. In *DATE*. 97:1–97:6.

[8] Guy Durrieu, Madeleine Faugere, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. 2014. Predictable flight management system implementation on a multicore processor. In *ERTS*.

[9] Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. 2014. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *RTCSA*. 1–10.

[10] Stuart Fisher. 2013. Certifying applications in a multi-core environment: The world's first multi-core certification to SIL 4. *SYSGO white paper* (2013).

[11] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, Lothar Thiele, and Benoît Dupont de Dinechin. 2016. Mixed-Criticality Scheduling on Cluster-Based Manycores with Shared Communication and Storage Resources. *Real-Time Systems* 52, 4 (2016), 399–449.

[12] M. Hassan and H. Patel. 2016. Criticality- and Requirement-Aware Bus Arbitration for Multi-Core Mixed Criticality Systems. In *RTAS*.

[13] Pengcheng Huang, Georgia Giannopoulou, Rehan Ahmed, Davide Basilio Bartolini, and Lothar Thiele. 2015. An Isolation Scheduling Model for Multicores. In *RTSS*.

[14] Jung-Eun Kim, Man-Ki Yoon, R. Bradford, and Lui Sha. 2014. Integrated Modular Avionics (IMA) Partition Scheduling with Conflict-Free I/O for Multicore Avionics Systems. In *COMPSAC*. 321–331.

[15] N. Kim, B. C. Ward, M. Chisholm, C. Y. Fu, and others. 2016. Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. In *RTAS*.

[16] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, J. A. Scoredos, and others. 2010. Mixed-criticality real-time scheduling for multicore systems. In *ICCIT*. 1864–1871.

[17] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron. 2014. The ROSACE case study: From Simulink specification to multi/many-core execution. In *RTAS*. 309–318.

[18] Rodolfo Pellizzoni, Bach D. Bui, Marco Caccamo, and Lui Sha. 2008. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In *Real-Time Systems Symposium, 2008*. IEEE, 221–231.

[19] Q. Perret, P. Maurere, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet. 2016. Temporal Isolation of Hard Real-Time Applications on Many-Core Processors. In *RTAS*.

[20] Lukas Sigrist, Georgia Giannopoulou, Pengcheng Huang, Andres Gomez, and Lothar Thiele. 2015. Mixed-Criticality Runtime Mechanisms and Evaluation on Multicores. In *RTAS*. 194–206.

[21] Stefanos Skalistis and Alena Simalatsar. 2016. Worst-Case Execution Time Analysis for Many-Core Architectures with NoC. In *FORMATS*. 211–227.

[22] Pranav Tendulkar, Peter Poplavko, Jules Maselbas, Ioannis Galanommatis, and Oded Maler. 2015. *A Runtime Environment for Real-time Streaming Applications on Clustered Multi-cores*. Technical Report TR-2015-6. Verimag Research Report.

[23] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *CC*. 179–196.

[24] Domiţian Tămaş-Selicean and Paul Pop. 2015. Design Optimization of Mixed-Criticality Real-Time Embedded Systems. *ACM TECS* 14, 3 (2015), 1–29.

[25] S. Vestal. 2007. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS*. 239–243.

[26] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. 2009. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 7 (2009), 966–978.

[27] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. 2014. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *RTAS*. 155–166.
[28] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. 2013. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS*. 55–64.

Received April 2017; revised June 2017; accepted June 2017