Fixed-Priority Memory-Centric Scheduler for COTS-Based Multiprocessors

Gero Schwäricke
Technical University of Munich, Germany
gero.schwaericke@tum.de

Tomasz Kloda
Technical University of Munich, Germany
tomasz.kloda@tum.de

Giovani Gracioli
Federal University of Santa Catarina, Brazil
giovani@lisha.ufsc.br

Marko Bertogna
Università di Modena e Reggio Emilia, Italy
marko.bertogna@unimore.it

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

Abstract

Memory-centric scheduling attempts to guarantee temporal predictability on commercial-off-the-shelf (COTS) multiprocessor systems to exploit their high performance for real-time applications. Several solutions proposed in the real-time literature have hardware requirements that are not easily satisfied by modern COTS platforms, like hardware support for strict memory partitioning or the presence of scratchpads. However, even without said hardware support, it is possible to design an efficient memory-centric scheduler.

In this article, we design, implement, and analyze a memory-centric scheduler for deterministic memory management on COTS multiprocessor platforms without any hardware support. Our approach uses fixed-priority scheduling and proposes a global "memory preemption" scheme to boost real-time schedulability. The proposed scheduling protocol is implemented in the Jailhouse hypervisor and Erika real-time kernel. Measurements of the scheduler overhead demonstrate the applicability of the proposed approach, and schedulability experiments show a 20% gain in terms of schedulability when compared to contention-based and static fair-share approaches.

2012 ACM Subject Classification Computer systems organization → Embedded systems; Computer systems organization → Multicore architectures; Software and its engineering → Real-time schedulability; Security and privacy → Virtualization and security

Keywords and phrases Schedulability Analysis, Scheduler Implementation, memory-centric Scheduling, Virtualization, Multiprocessor

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.1

Funding *Marco Caccamo*: Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research.

1 Introduction

In commercial-off-the-shelf (COTS) multiprocessor systems, a task's execution time can increase by an order of magnitude due to shared main memory interference, generated by tasks running simultaneously on other cores [25]. Bounding or even eliminating this interference is highly desirable in real-time systems. The *PRedictable Execution Model* (*PREM*) [40,41] and

its extensions [6, 8, 15, 57] attempt to mitigate the memory contention problem by splitting a task's execution into separate phases for memory transactions and pure computation. During a memory phase, task data is fetched from the shared main memory to a fast local memory (private cache partition or scratchpad). During a computation phase, a task performs the computation on the prefetched data without the need to access the shared main memory. A memory-centric scheduler ensures that memory phases of tasks running on different processors are scheduled in non-overlapping time slots.

Many successful implementations of memory-centric schedulers [19, 41, 52, 53, 57] have adopted time-division multiplexing (TDM) as the underlying principle. TDM-based arbitration distributes the resource main memory in a very predictable way. One known downside of this approach is the underutilization of the resource: if a processor has a reserved time slot but does not use it, the slot cannot be offered to another processor [23, 49]. A priority-based memory scheduler may permit to schedule task sets with higher processor utilization due to its work-conserving nature [5].

The prohibitively high preemption cost of memory transactions favors non-preemptive scheduling. This is particularly relevant when using Direct Memory Access (DMA) controllers, which can effectively increase the overall system performance by enabling an overlap of memory transactions and computation [3, 20, 53, 57]. Unfortunately, most COTS processors use cache memories without any architectural support for offloading memory operations (e.g., cache stashing). Nevertheless, the processor can still temporarily suspend and resume its own prefetch operation. The ability to suspend a processors prefetch operation can be exploited to enable global memory preemptions in memory-centric scheduling and thus remove blocking caused by low priority tasks running on other cores.

State-of-the-art approaches for the implementation of memory-centric scheduling rely on specific hardware support (e.g., cache locking [3, 46, 60] or the presence of scratchpads [19, 53, 57]). Such features may not be present on all modern COTS multiprocessor systems. For instance, the ARM Cortex A-57 (used in Nvidia Tegra TX1/TX2 and Exynos 7 Octa) and its more energy-efficient alternative the ARM Cortex A-53 (used in Raspberry Pi 3) have no scratchpads and no explicit locking support for caches.

This paper. To address these shortcomings and enable a contentionless memory scheduling, we make the following contributions:

- design of a fixed-priority preemptive memory-centric scheduler for modern COTS multiprocessor systems that do not feature hardware support for predictable memory management,
- implementation of the memory-centric scheduler within a hypervisor and a real-time operating system kernel that are based entirely on open-source software components,
- evaluation of the effectiveness and the limitations of the proposed approach with a set of microbenchmarks and real-world workloads,
- fixed-priority partitioned multiprocessor schedulability analysis for PREM-compliant task sets backed by schedulability experiments that incorporate the actual memory arbitration overheads in the analytical model and identify a heuristic for task partitioning.

Paper structure. The remainder of the paper is organized as follows. In Section 2, we discuss the design decisions, design alternatives, and rationale behind the proposed memory-centric scheduler. The model of the scheduler, and the system as a whole, are formalized in Section 3, followed by the schedulability analysis in Section 4. The implementation details are given in Section 5, and the evaluation is presented in Section 6. Finally, Section 7 reviews the related work, and Section 8 concludes the paper.

2 System Design and Assumptions

Figure 1 shows an overview of our system design. A hypervisor provides memory-centric scheduling and resource partitioning for real-time operating system (RTOS) guests and controls the communication among RTOSs through interprocessor interrupts (IPIs). The objective is to rule out interference among the RTOSs due to concurrent main memory accesses. In the next subsections, we describe the assumptions required by this work considering the system architecture and briefly discuss our design choices.

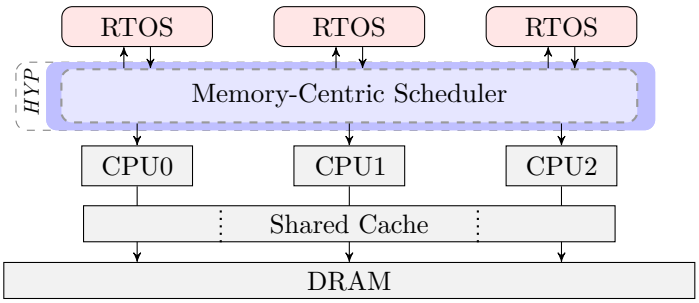


Figure 1 System architecture example for $N = 3$ real-time operating systems.

2.1 Predictable Execution Model

To minimize the amount of main memory interference, the processors service their tasks according to the *PREM*. Such tasks are split into a memory phase and a computation phase. The memory phase prefetches all required task data into a local memory section and is therefore recognizable by a large number of main memory accesses. The computation phase uses only data that was prefetched in a prior memory phase and shows almost no main memory accesses.

2.2 Local Memory

Data that was prefetched during memory phases must be kept in a fast local memory. Two types of local memories are typically used: last-level caches and scratchpads.

When using scratchpads, memory blocks are moved to/from the main memory explicitly in software (usually with the help of a DMA controller). The deterministic nature of scratchpads makes them a suitable choice in real-time systems [43]. However, support for scratchpad memories is rare in commercially available platforms [19, 36].

Last level caches (LLC), as available in most COTS platforms, can store considerable amounts of data copies of frequently used main memory sections. LLCs are self-managed and generally perform well on generic workloads, but introduce some degree of non-determinism because their memory is shared among the processors of a multiprocessor system. The cached data from one core can be evicted by another core if they are using data in memory spaces that are mapped to the same cache location (index). A cache partitioning technique can reduce this interference (see the next subsection). In this work, we consider processor architectures that have a shared LLC and no scratchpads.

2.3 Cache Partitioning

To protect cached data from eviction, cache partitioning assigns a portion of the cache to a given task or core for its exclusive use [19]. It can be done using either a hardware feature (e.g., Intel's Cache Allocation Technology [51], ARM's Lockdown by master [4]) or a software method (e.g., cache coloring [24, 31]). Many embedded platforms do not feature hardware cache partitioning (e.g., platforms based on ARM Cortex-A57 or A72). Thus, in this work, we use software cache partitioning based on cache coloring.

Cache coloring uses virtual memory to take control over the placement of virtually addressed memory in a physically-indexed cache. Memory pages mapped to the same cache sets are said to share the same color. Thus pages with different colors have different locations within the cache. The cache is partitioned by assigning distinct colors to different tasks or cores. The maximal number of distinct colors depends on the cache geometry (size, number of ways, cache line size) and the size of physical pages.

We assume that the hypervisor provides cache coloring to guest OSs, which overcomes many practical problems related to the modification of memory allocators in the OSs. Hardware-assisted virtualization features an additional address translation stage at which intermediate physical addresses (IPA), used by the guests, are translated into actual physical addresses (PA). The page tables at this translation stage are controlled by the hypervisor and can be configured to assign only pages of specific colors to a guest OS [20].

All tasks on a processor can inherit the same set of assigned colors [26], or each task can be assigned a subset of the processor's color set [60]. The latter approach requires a coloring-aware OS to distribute the colors among the tasks [26]. Apart from several academic implementations [19, 31, 62, 64], such a technique is not common. We assume a coloring-unaware OS. The model presented in this paper is also valid for hardware cache partitioning.

Figure 2 illustrates cache coloring implemented in a virtualized environment hosting two guests such that the first guest (CPU0) is allocated to the upper two and the second guest (CPU1) to the lower two sets of a 4-way set-associative L2 cache.

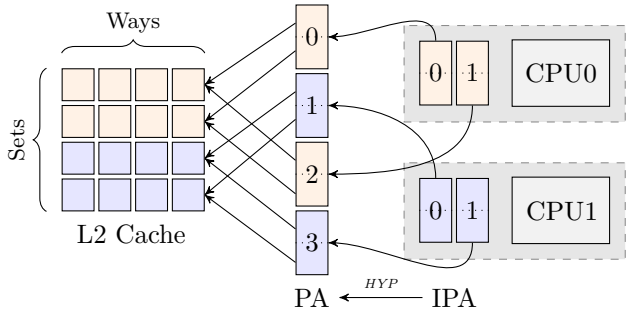


Figure 2 Cache coloring at the hypervisor level for a 4-way set-associative L2 cache.

2.4 Memory Prefetching

Several scheduling frameworks [20, 48, 53] take advantage of DMA offloading to parallelize memory prefetching of newly released tasks alongside ongoing computation. Writing directly to cache with a DMA controller (e.g., ARM's cache stashing) is featured on certain embedded platforms (e.g., Freescale P4080, ARM Cortex-A9, new Cortex-A75, and A55). However,

this mechanism is not widely available [60] (as is the case for Cortex-A57). Without this "passive prefetching", the cache partition must be loaded actively by the processor executing prefetch instructions (such as the `prfm` assembly instruction in ARM architectures).

When prefetching data into cache-based local memory, the cache replacement policy must be taken into account to avoid the problem of self-eviction [40, 41]. Since congruent addresses contend for the cache lines of the same set, the cache replacement policy decides in which cache way of the set the data is placed.

A popular replacement policy, pseudo-random (as in Cortex-A57), works as follows: if there is at least one way in the target set whose cache line is empty (invalid) then the new data is simply copied into that empty cache line; otherwise, the controller randomly selects a way and unloads its cache line to make space for the new data. To prevent self-evictions during prefetching, a sufficient number of unused cache lines must be invalidated before [28, 36].

2.5 Multiprocessor Scheduler and Preemptions

Global non-preemptive multiprocessor scheduling has recently received increasing attention [3, 33, 39, 58]. On top of that, cache-aware scheduling algorithms were designed, guaranteeing that two running tasks' cache spaces do not overlap at any time [21]. These works assume that either all tasks can fit into the cache or cache partitions can be reassigned arbitrarily at runtime. Yet, efficiently implementing dynamic cache partition reallocation can have very high overheads [42, 63]. Also, in direct-mapped and set-associative physically indexed caches, the memory of two different tasks can be mapped to the same cache sets. Without full control of way-placement (e.g., ARM's Lockdown by way, as in [34]), the concurrent execution of such tasks can cause mutual interference. Because of the mentioned limitations, lower runtime overheads, static isolation, and easier extension to heterogeneous platforms [7, 13], we opt for partitioned multiprocessor scheduling.

Tasks assigned to the same processor use the same cache partition. In non-preemptive task scheduling, each task can use the entire partition. In preemptive task scheduling, the partition must be divided among the tasks, leading to less local memory for each task. To maximize the task data footprint, we consider a non-preemptive scheduler. If some degree of preemption is required, limited preemption [9], stack-sharing techniques, or cache-cognizant scheduling [54] can be used to increase memory efficiency.

Since the processors have non-overlapping cache partitions, tasks running on one processor cannot evict the prefetched data of the other processors. Processors can also stop their ongoing cache prefetching operation at any time. Therefore, a processor can preempt another processor during its memory phases using interprocessor interrupts (as will be shown in Section 6.1).

2.6 Priority-based Scheduling

We consider fixed-priority scheduling due to its widespread usage in real-time operating systems (e.g., VxWorks, FreeRTOS, Erika). As stated before, tasks follow the PREM. However, we performed a slight modification to the original model: we allow global memory preemptions during the memory phases (a task can be preempted during its memory phase by tasks running on another processor, but cannot be preempted by tasks running on the same processor). This modification helps to reduce the blocking time caused by low priority tasks [9].

Global memory preemptions during memory phases may lead to uncontrolled *priority inversion* [44, 50]. For instance, consider the case illustrated in Figure 3. A high priority task τ_H is waiting for a low priority task τ_L scheduled on the same processor P_1 due to local

non-preemptive scheduling. Task τ_M, with medium priority, running on processor P_2, can preempt the low priority task τ_L during its memory phase and increase the blocking time of task τ_H. Priority inversion can be countered by adopting a synchronization protocol [44, 50], where each protocol comes with a different implementation complexity and inversion bound.

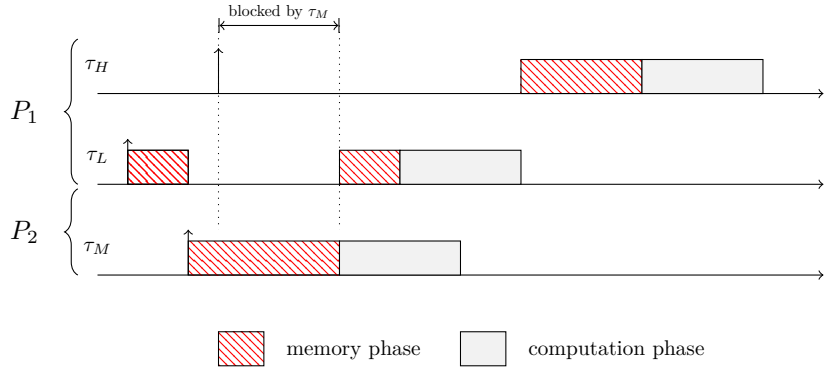


Figure 3 Priority inversion in fixed-priority inter-processor preemptive and intra-processor non-preemptive memory-centric scheduling.

To avoid this problem, our design enforces a strict priority order between the tasks allocated to two different processors. Each processor has a statically assigned unique memory access priority that is inherited by all tasks allocated to it. Our approach comes with the advantage of low runtime overhead compared to a priority inheritance protocol: a registered memory access request does not have to be updated if a new task is released on the same processor. The main drawback lies in additional complexity in the task-to-processor allocation, which we will discuss in Section 6.3.

3 System Model

We consider a multiprocessor platform with shared last level cache and shared main memory on which a partitioned scheduler executes a set of sporadic tasks on each processor. The tasks are composed of a single memory and a single computation phase.

Locally, the scheduler uses a fixed-priority non-preemptive policy. As an exception, memory phases can be preempted by tasks running on processors of higher priority, as described in Section 2.6.

3.1 Processors

The main memory is shared among N processors. Each processor is assigned a unique static priority. The priorities govern the contentionless access to the main memory. Processors are indexed in priority order with P_1 having the highest priority and P_N the lowest priority. We say that $P_h > P_l$ if processor P_h has a higher priority than processor P_l.

3.2 Tasks

We consider a partitioned system in which each task is statically assigned to a single processor. We say that $\tau_i \in P$ if τ_i is allocated to the processor P, and we denote by $P(i)$ the processor to which task τ_i is allocated. Each task τ_i gives rise to a potentially infinite sequence of jobs. Task τ_i releases jobs sporadically after the minimum inter-arrival time T_i (period), and each job of τ_i must be completed within a fixed time interval from its release given by a relative deadline D_i. We assume that tasks have constrained deadlines, i.e., $D_i \leq T_i$.

Each task is composed of two phases: a memory phase and a computation phase. During its memory phase, task τ_i fetches data from the main memory to the local cache partition. The prefetch operation is supposed to fully occupy the processor that is unavailable to perform other work during that time. In its computation phase, the task performs the computation on the prefetched data. Thus the task does not access the main memory. The maximal time to complete the memory phase, with exclusive memory access, is m_i (no memory interference from the other processors), and the maximal time to complete the computation phase, executed in isolation from the other tasks, is c_i (no other tasks running on the same processor). The total worst-case execution time of task τ_i is given as $e_i = m_i + c_i$. All the above parameters are positive integers. The total memory utilization of all the tasks allocated to processor P is given as $U^m(P) = \sum_{\tau_j \in P} \frac{m_j}{T_j}$ and the total utilization as $U(P) = \sum_{\tau_j \in P} \frac{e_j}{T_j}$.

The tasks are assumed to be independent and do not share resources other than the processor and the local memory partition. The worst-case execution time includes preemption, context switch, and scheduler overheads. Once a task starts, it will not voluntarily suspend its execution. Tasks are scheduled on each processor by a fixed-priority non-preemptive scheduler and are indexed in priority order with τ_1 having the highest priority and τ_n the lowest priority where n is the number of tasks allocated to the processor under study. Each task has a unique priority. We introduce the notation $hp(i)$ and $lp(i)$ for the set of tasks with priorities, respectively, higher than and lower than the priority of task τ_i, which are running on the same processor ($hp(i) \subseteq \{\tau_j \,|\, \tau_j \in P(i)\}$ and $lp(i) \subseteq \{\tau_j \,|\, \tau_j \in P(i)\}$). The global fixed-priority preemptive scheduler schedules the accesses to the main memory: a task running in its memory phase can be preempted by the memory phase of a task running on a processor with higher priority. Only one task can access the main memory at a time. If $P_h > P_l$, then all tasks allocated to processor P_h have higher memory access priority than all tasks allocated to processor P_l.

3.3 Memory

Memory is a globally shared resource and can be accessed by all processors incurring the same memory access latencies for each processor. We assume that the processor waits synchronously for every prefetch instruction caused either by a cache miss or an explicit prefetch instruction. We assume the order in which the memory controller serves memory requests issued simultaneously from different processors to be unknown. We assume that the local cache memory has much higher bandwidth and lower latency than the main memory. Each processor has a dedicated cache partition of a fixed size. The largest memory footprint of any real-time task allocated to a processor can fit entirely into the processor's local memory partition. The partitions are non-overlapping.

4 Schedulability Analysis

In the following, we give details of the schedulability analysis for two-phase *PREM*-compliant tasks running under the fixed-priority memory-centric scheduler described in the previous sections. We first identify the different sources of interference and show how the interference can be upper bounded. We then integrate the obtained factors into the response time analysis.

Figure 4 shows a sample execution of a task set on two processors. Processor P_1 has higher priority than processor P_2. Tasks are indexed in decreasing priority order and execute non-preemptively. Tasks may experience blocking from low priority tasks allocated to the

same processor (e.g., task τ_2 blocked by task τ_4 at $t = 0$). However, their memory phases can be preempted by the memory phases of tasks running on the other processors (e.g., task τ_2 preempted by task τ_1 at $t = 4$).

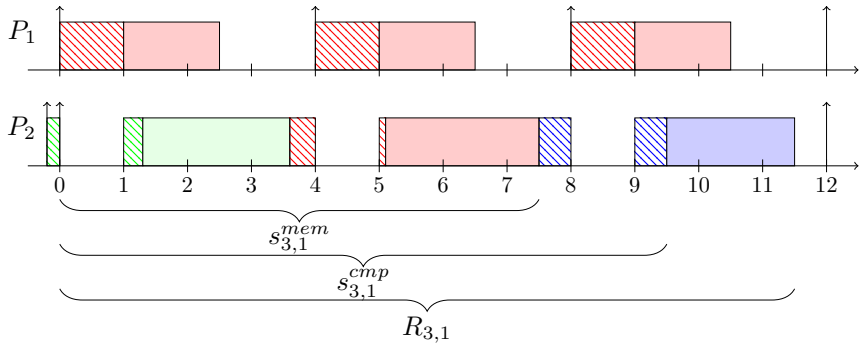


Figure 4 Sample schedule for two processors. P_1 has a higher priority than P_2. Task running on P_1: $\tau_1 : (m_1 = 1.0, c_1 = 1.5, T_1 = 4)$. Tasks running on P_2: $\tau_2 : (m_2 = 0.5, c_2 = 2.4, T_2 = 12)$, $\tau_3 : (m_3 = 1.0, c_3 = 2.0, T_3 = 12)$, and $\tau_4 : (m_4 = 0.5, c_4 = 2.3, T_4 = 24)$.

Sources of interference. The worst-case response time R_i of task τ_i is composed of the following factors:

- *Blocking* due to a lower priority task on the same processor that starts its execution just before the release of task τ_i.
- *Intra-processor interference* due to the tasks with higher priority than task τ_i scheduled on the same processor. This interference can delay the start of a task running on $P(i)$.
- *(Inter-processor) memory interference* due to the memory phases of tasks scheduled on higher priority processors. This interference can only delay the execution of memory phases running on the processor under analysis.

4.1 Interference Calculation

We characterize the interference that task τ_i running on processor $P(i)$ can experience during an interval of length $\Delta > 0$ due to blocking, intra-processor interference, and inter-processor memory interference.

Formulas for blocking and intra-processor interference can be obtained straightforwardly. The blocking factor B_i is the longest worst-case execution time among all tasks with priority lower than τ_i allocated to the same processor. The maximum of an empty set is assumed to be zero.

$$B_i = \max_{\tau_j \in lp(i)} e_j \tag{1}$$

Intra-processor interference $I_i(\Delta)$ is the sum of the worst-case execution times of all higher priority task instances that can be released on the same processor within the time interval Δ:

$$I_i(\Delta) = \sum_{\tau_j \in hp(i)} \left\lceil \frac{\Delta}{T_j} \right\rceil e_j \tag{2}$$

The problem of estimating the inter-processor memory interference cannot be solved with the above formula and requires additional analysis. This is due to the fact that i) the

memory interference of the two-phase model can exhibit a jitter behavior, and ii) within the processor busy period preemption from the other processors can occur only during memory phases, which are interleaved with the non-preemptive computation phases.

The jitter behavior is evidenced by the fact that the time between the start of the memory phases of two consecutive instances of the same task can be shorter than its period. Consider the example from Figure 5. We try to characterize the memory interference generated by the memory phases of task τ_1. The first instance of task τ_1 is blocked by the lower priority task τ_2. The first memory phase of τ_1 starts at time instant 2. Every following memory phase of τ_1 does not start earlier than at its next release, that is, at time instants 4, 8, 12, ... The time distance between the start times of the first and the second memory phase is shorter than between the second and the third memory phase.

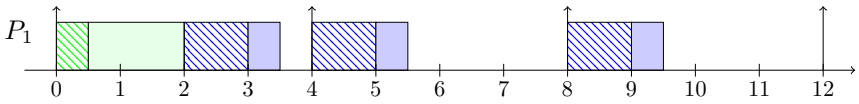


Figure 5 Inter-processor memory interference from a single processor. Tasks running on processor P_1: $\tau_1 : (m_1 = 1.0, c_1 = 0.5, T_1 = 4)$ and $\tau_2 : (m_2 = 0.5, c_2 = 1.5, T_2 = 12)$.

This is equivalent to the problem of *inherited release jitter* [55]. When estimating memory interference generated by a task from another processor, we may assume that its release jitter is equal to its latest start time. The latter can be safely upper bounded by $R_i - e_i$ as proposed in [11] for remote blocking. According to the above approach, the inter-processor memory interference from tasks running on processors with a higher priority than processor $P(i)$ on which τ_i is running can be expressed as:

$$\alpha_i(\Delta) = \sum_{P_h > P(i)} \sum_{\tau_j \in P_h} \left\lceil \frac{\Delta + R_j - e_j}{T_j} \right\rceil m_j \tag{3}$$

Not every external memory phase within the busy period of the processor under analysis can interfere with the memory phases that this particular processor runs. Consider the example shown in Figure 6. In time interval $[0, 9]$, task τ_1 running on processor P_1 can release up to 3 memory phases. However, not all of these memory phases can interfere with the execution of a task running on processor P_2. Indeed, processor P_2 is running a computation for a long time (from time instant 2 to 9), which is not affected by the memory interference from processor P_1.

Evaluating the number of times the processor is exposed to the memory interference from the higher priority processors can reduce the pessimism stemming from the assumption that every external memory phase is interfering. For instance, in the example from Figure 6, processor P_2 executes only one memory phase in the interval $[0, 9]$. Therefore it can be assumed that it does not experience the memory interference from processor P_1 more than once.

The number of memory phases released on processor $P(i)$ during the busy-period of task τ_i with length $\Delta > 0$, such that there is a pending instance of τ_i, can be calculated as (we consider higher priority tasks, previous instances of task τ_i and one lower priority task that may block τ_i if τ_i is not the lowest priority task):

$$\mathcal{N}_i(\Delta) = \sum_{\tau_j \in hp(i)} \left\lceil \frac{\Delta}{T_j} \right\rceil + \left\lfloor \frac{\Delta}{T_i} \right\rfloor + [lp(i) \neq \emptyset] \tag{4}$$

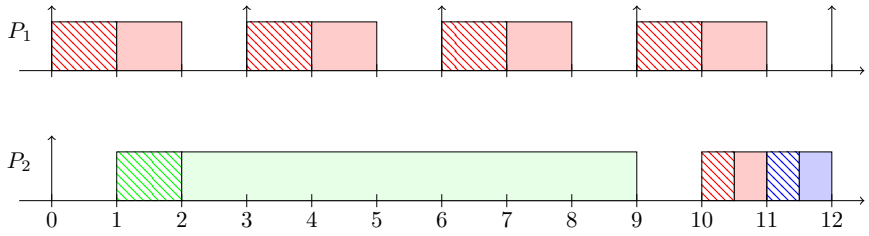


Figure 6 Exposure to inter-processor memory interference in a two processor system. Processor P_1 has a higher priority than processor P_2. Task running on P_1: $\tau_1 : (m_1 = 1.0, c_1 = 1.0, T_1 = 3)$. Tasks running on P_2: $\tau_2 : (m_2 = 0.5, c_2 = 0.5, T_2 = 24)$, $\tau_3 : (m_3 = 0.5, c_3 = 0.5, T_3 = 24)$, and $\tau_4 : (m_4 = 1.0, c_4 = 8.0, T_4 = 24)$.

where $[\cdot]$ is the Iverson bracket evaluating to one for a true expression and to zero otherwise. The maximum interference that can affect one of these memory phases released on processor $P(i)$ is not greater than the least positive fixed-point of the following recurrent equation:

$$\varepsilon_{P(i)} = \alpha_i(\varepsilon_{P(i)} + \widehat{m}_{P(i)}) \quad \text{where} \quad \widehat{m}_{P(i)} = \max\{m_j \,|\, \tau_j \in P(i)\} \tag{5}$$

The term $\widehat{m}_{P(i)}$ designates the longest memory phase among all the memory phases of tasks running on processor $P(i)$. The fixed-point iteration converges if and only if $\sum_{P_h > P(i)} U^m(P_h) < 1$. All in all, the total memory interference from the higher priority processors within interval $\Delta > 0$ is not greater than the number of memory phases released during the busy period times the maximum interference from the higher priority processors that can affect one of these memory phases:

$$\beta_i(\Delta) = \mathcal{N}_i(\Delta) \cdot \varepsilon_{P(i)} \tag{6}$$

This approach can overapproximate the actual memory interference as well. It can happen when memory phases on the processor under analysis are scheduled more frequently than the memory phases of the higher priority processors (e.g., in Figure 6 from time instant 10 to 11.5). If that is the case, an approach in which each external memory phase is considered to be interfering can give a less pessimistic estimation.

Based on these two approaches, we can now formulate a safe upper bound on the inter-processor memory interference from the tasks scheduled on higher priority processors:

$$\min\{\alpha_i(\Delta), \beta_i(\Delta)\} \tag{7}$$

4.2 Response Time Analysis

The worst-case response time of task τ_i running on processor $P(i)$ can be determined by combining the expressions for interference presented above with the analysis for single processor non-preemptive scheduling [9, 14].

The latest start time $s^{mem}_{i,k}$ of the memory phase of the k-th instance of task τ_i is bounded by the least positive fixed-point of the following recurrent equation:

$$s^{mem}_{i,k} = B_i + I_i(s^{mem}_{i,k}) + (k-1) \cdot e_i + \min\left\{\alpha_i(s^{mem}_{i,k}), \beta_i(s^{mem}_{i,k})\right\} \tag{8}$$

Now we compute the latest start time of its computation phase. At $s^{mem}_{i,k}$, task τ_i starts its memory phase. It cannot be delayed by the tasks from the same processor (non-preemptive

local scheduling). From that point on, a further delay can only be caused by the memory interference from tasks running on processors with higher priority (preemption in memory phase):

$$s_{i,k}^{cmp} = B_i + I_i(s_{i,k}^{mem}) + m_i + (k-1)\cdot e_i + \min\left\{\alpha_i(s_{i,k}^{cmp}), \beta_i(s_{i,k}^{mem}) + \alpha_i(s_{i,k}^{cmp} - s_{i,k}^{mem})\right\} \quad (9)$$

Proof. Let $s_{i,k}^{mem}$ and $s_{i,k}^{cmp}$ be respectively, the start and the end of the memory phase of the k-th instance of task τ_i. To ease the notation, we replace $s_{i,k}^{mem}$ with t_s and $s_{i,k}^{cmp}$ with t_e. At t_s, the memory phase of task τ_i starts and cannot be delayed by the tasks running on the same processor due to the non-preemptive scheduling. The interference from higher priority processors within the interval $[0, t_e]$ is upper bounded by $\alpha_i(t_e)$.

First, suppose that $\alpha_i(t_s) > \beta_i(t_s)$. The interference from the higher priority processors within the interval $[0, t_s]$ is upper bounded by $\beta_i(t_s)$ and within the interval $[t_s, t_e]$ by $\alpha_i(t_e - t_s)$. Consequently, within the interval $[0, t_e]$, it cannot be greater than $\min\{\alpha_i(t_e), \beta_i(t_s) + \alpha_i(t_e - t_s)\}$. Now suppose that $\alpha_i(t_s) \leq \beta_i(t_s)$. It must be that $\alpha_i(t_e) \leq \beta_i(t_s) + \alpha_i(t_e - t_s)$ since $\alpha_i(t_e) \leq \alpha_i(t_s) + \alpha_i(t_e - t_s) \leq \beta_i(t_s) + \alpha_i(t_e - t_s)$. Consequently, $\min\{\alpha_i(t_e), \beta_i(t_s) + \alpha_i(t_e - t_s)\} = \alpha_i(t_e)$. ◀

Once started, the computation phase executes non-preemptively and is not subject to any type of interference. The worst-case response time of the k-th instance of task τ_i is:

$$R_{i,k} = s_{i,k}^{cmp} + c_i - (k-1)\cdot T_i \quad (10)$$

To determine the worst-case response time of task τ_i it is necessary to calculate the response time of each instance within the i-level busy period, i.e., the longest time processor $P(i)$ can be busy executing the jobs of tasks with a priority higher than or equal to the priority of task τ_i [30]:

$$L_i = B_i + I_{i-1}(L_i) + \min\left\{\alpha_i(L_i), \beta_i(L_i) + \widehat{m}_{P(i)}\right\} \quad (11)$$

Note that $I_{i-1}(L_i)$ includes the jobs $hp(i)$ as well as the jobs of τ_i. Moreover, $\widehat{m}_{P(i)}$ must be added to $\beta_i(L_i)$ to account for the last instance of τ_i. The fixed-point iteration can be solved for all the tasks of processor $P(i)$ if and only if:

$$U(P(i)) + \min\left(\sum_{P_h > P(i)} U^m(P_h), \sum_{\tau_j \in P(i)} \frac{\varepsilon_{P(i)}}{T_j}\right) < 1 \quad (12)$$

The largest $R_{i,k}$ calculated for the instances of τ_i released within interval L_i gives the worst-case response time:

$$R_i = \max_k R_{i,k} \quad \text{where} \quad k \in \left[1, \left\lceil \frac{L_i}{T_i} \right\rceil\right] \quad (13)$$

Figure 4 shows the scenario corresponding to the worst-case response time of task τ_3.

4.3 Schedulability Test

The task set is said schedulable if all jobs of every task meet their deadlines, i.e., $\forall i : R_i \leq D_i$. Before computing the worst-case response time of task τ_i, the worst-case response times of all higher priority tasks running on the other processors must be calculated first. This is necessary to account for their memory interference in Equation (3). Also note that the factor $\varepsilon_{P(i)}$ from Equation (5) can be computed only once, before starting the response time analysis of the first task on processor $P(i)$. Moreover, the values of the external memory interference given by Equation (7) can be stored and reused to speed up the computation for all the tasks allocated to the same processor.

5 System Implementation

In this section, we describe the implementation of the proposed fixed-priority memory-centric scheduling approach. According to the system design presented in Section 2, the full system has three parts: application tasks, one or more RTOSs, and a hypervisor containing the memory access scheduler. The structure of this section follows these three parts, beginning with a description of requirements for application tasks, followed by implementation details for the support in an RTOS (i.e., Erika), and the implementation of the memory-centric scheduling in a hypervisor (i.e., Jailhouse). At last, we depict the interaction between an RTOS and the hypervisor and describe the concrete implementation that we used for the evaluation.

5.1 Application Task Requirements

Tasks must follow the two-phase PREM as specified in Section 3. The code structure of a suitable PREM task is typically formed by i) a call to a function to prefetch data (memory phase), ii) a call to a function signaling the end of the memory phase (here: `end_memory_phase()`), and iii) the code using the prefetched data (computation phase). All tasks on all RTOSs in the system need to follow this structure to ensure proper guarding of all memory accesses.

Data prefetching is performed using assembly instructions (`prfm` for the ARM architecture). On platforms with random replacement policy, a sufficient number of data cache lines need to be cleaned and invalidated before the data prefetching (`cisw` for the ARM architecture).

5.2 Real-Time Operating System Support

The RTOS provides typical task states (ready, running, waiting) with the addition of a state for tasks that were suspended by the memory arbitration (suspended). When a task enters the ready state, and no prior task is in the suspended state, the RTOS performs a hypercall (trap to hypervisor mode, here: `enable_request`) to request memory access permission from the memory-centric scheduler implemented in the hypervisor. If the memory access is not granted or suspended tasks exist, the task is suspended. Otherwise, it runs the task (begin of the memory phase). At the end of the memory phase (call to `end_memory_phase()`), the RTOS revokes its memory access permission through a hypercall (here: `disable_request`) and disables all interrupts to run the subsequent computation phase without interference. When a task's computation phase ends, the RTOS enables the interrupts again and performs an `enable_request` hypercall for the highest priority task in the suspended state (if any).

The hypervisor can pause and resume a task's memory phase at any point in time due to the preemptive property of the memory-centric scheduling. Pausing and resuming is signaled to the RTOS by interprocessor interrupts (IPIs). When receiving an IPI to pause the memory access, the RTOS suspends the running task. Since tasks in the suspended state are also non-preemptive, the prefetched data will still be in the cache partition when the task is resumed by another IPI.

Figure 7 shows the task states and their transitions in the RTOS: the memory phase can only be accessed once the permission to access the main memory was acquired, either through a hypercall (from the ready state) or an interrupt (from the suspended state). The hypervisor is notified through a hypercall to revoke the access request once the memory phase is completed.

To reclaim execution time when the memory arbitration suspends a task, the RTOS can optionally implement an aperiodic server running preemptively at the lowest priority.

The server's tasks should access only very little memory or work entirely on non-cacheable memory. The effect of the server's memory accesses should be added in the inter-processor memory interference computation for the other processors.

5.3 Hypervisor Support

The hypervisor implements the memory-centric scheduler. The implementation is based on a kind of token passing algorithm. There exists one token in the system, which permits the RTOS possessing it to freely access the main memory. Each RTOS can register (and deregister) one request for the token. Requests have a priority, which is equal to the requesting processor's priority.

The scheduling occurs when one of the hypercalls `enable_request` and `disable_request` is executed. The first registers and the second revokes a token request. The first hypercall's return value informs the calling RTOS if the token was acquired. When registering a request, the hypervisor reclaims the token if it is assigned to a core of lower priority than the request priority by sending an IPI (see Listing 1). When revoking a request, the hypervisor finds the highest priority pending request and passes the token to the associated processor by sending an IPI (see Listing 2).

Listing 1 Enable memory request hypercall.

```
bool enable_request(int request_prio,
                    int request_proc):
  spin_lock(&mem_arbiter)
  memory_requests[request_proc] ←
      request_prio
  if request_prio > token_prio:
    if token_owner is not undefined:
      notify_pause(token_owner)
    endif
    token_owner ← request_proc
    token_prio ← request_prio
  endif
  bool token_acquired = (token_owner ==
                         request_proc)
  spin_unlock(&mem_arbiter)
return token_acquired
```

Listing 2 Disable memory request hypercall.

```
void disable_request(int request_proc):
  spin_lock(&mem_arbiter)
  memory_requests[request_proc] ←
      undefined
  token_owner ← undefined
  token_prio ← undefined
  foreach pending_prio in memory_requests:
    if pending_prio is not undefined and
        (token_prio is undefined or
         pending_prio > token_prio):
      token_owner ← index(pending_prio)
      token_prio ← pending_prio
    endif
  endforeach
  if token_owner is not undefined:
    notify_resume(token_owner)
  endif
  spin_unlock(&mem_arbiter)
return
```

A spinlock is used to control the access to shared scheduler data structures used from the different cores in the arbitration hypercalls. Consequentially, a processor calling the hypercall `enable_request` may have to wait. The proposed implementation assumes Linux's *ticket lock* (first in first out queue-based mechanism). Thus, the spinning processor cannot be bypassed twice by another processor. The worst-case time to register the memory request can therefore be bounded by $wcet(hypercall) + N \cdot wcet(arbitration)$. After winning the arbitration, the token may need to be reclaimed from a different processor. This event may induce a delay Δ_{IPI} needed for the interprocessor interrupt propagation. The maximal time to acquire exclusive memory access is therefore upper bounded by:

$$wcet(hypercall) + N \cdot wcet(arbitration) + \Delta_{IPI} \tag{14}$$

To enable the usage of an OS that does not provide the required support for the model, the hypervisor can optionally suspend all processors of the said OS when the token is assigned and resume the processors otherwise.

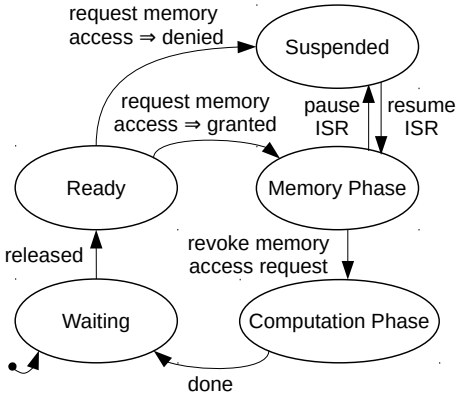


Figure 7 The states and transitions of a PREM task in the RTOS.

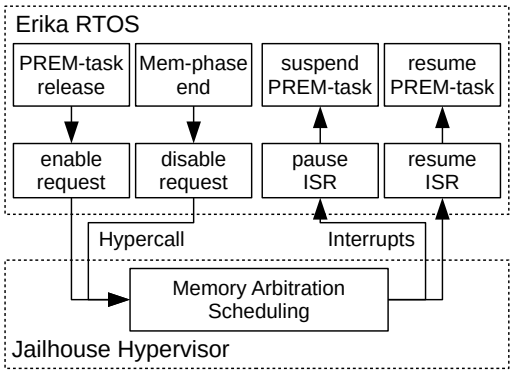


Figure 8 Communication of Erika and Jailhouse in the states of a PREM-task.

5.4 Our Implementation

As RTOS, we use Erika Enterprise version 3, which is an OSEK/VDX certified open-source/commercial RTOS [17]. Erika supports fixed-priority scheduling and runs on several embedded multicore platforms, such as Xilinx Ultrascale+ and NVIDIA Jetson TX2.

We restrict our implementation to PREM-tasks. We use function calls at task release and task completion instead of including the memory access arbitration into the scheduler to reduce implementation complexity. This simplification introduces only a small overhead in case a task is scheduled and then immediately suspended at the `enable_request` hypercall. We added a new task queue to the Erika kernel for tasks suspended by the memory-centric scheduler, since Erika does not support suspending tasks from an interrupt.

As hypervisor, we use *Jailhouse*, which is a partitioning hypervisor based on Linux [27] (required for bootstrapping) that runs on Intel and ARM 64-bit processors featuring hardware-assisted virtualization. Jailhouse statically splits the system into isolated partitions called cells. Each cell runs one guest OS and has full control over its statically assigned resources (e.g., CPUs, memory regions, and PCI devices). We have chosen Jailhouse due to its simplicity and low overhead that favors real-time applications [20, 45].

We implemented the memory-centric scheduler in Jailhouse. Local memory partitions for the RTOSs are created using cache coloring in Jailhouse [20]. We restrict our implementation by not coloring the Jailhouse code. The detrimental effect on the cores' cache partitions can be assumed as low. Figure 8 summarizes the functions in Erika and Jailhouse and the flow of calls.

6 Evaluation

In this section, we provide experimental results showing the overheads incurred by the memory-centric scheduling. We first measure different factors of overheads in microbenchmarks. Then, with a set of benchmarks, we evaluate the overall performance of the implemented system under stress. Finally, we conduct experiments with randomly generated workloads to measure the effectiveness of the schedulability analysis.

The evaluations in Subsection 6.1 and 6.2 are carried out on an Nvidia Tegra TX2 with 4 Cortex-A57 cores, 8GB 128-bit LPDDR4 RAM @ 1866Mhz, 2MB 16-way set-associative shared L2 cache with a pseudo-random replacement policy. For all measurements, Linux was

halted to remove interference from Linux itself or its DMAs on the memory subsystem. The remaining three cores each run Erika RTOS, allowing measurements for low, medium, and high processor priorities. The L2 cache is partitioned into four equal sizes by Jailhouse.

6.1 Microbenchmarks

We measured the duration of hypercalls, IPIs, and the memory arbitration in the memory-centric scheduler using a set of benchmarks that we crafted specifically for this purpose. The benchmarks are implemented as bare-metal applications and *Jailhouse* patches to measure the cost of particular operations using the ARM physical counter-timer. The following benchmarks were executed:

Hypercall The transition from the OS to the hypervisor and back to the OS without any computation in the hypervisor. It measures the cost of a single hypercall (`hvc` instruction).

IPI Issuing an interprocessor interrupt from one processor to another at the hypervisor level. It measures the cost of a single IPI.

Arbitration Arbitration within the hypervisor to find the processor with the highest memory access priority. Four processors ($N = 4$) were assumed.

Table 1 presents the results of *Hypercall*, *IPI*, and *Arbitration* microbenchmarks. Similar evaluations for *KVM* and *Xen* are available in [12]. We collected 1 000 000 samples for each benchmark and extracted the average execution time (AVG), standard deviation (STD), worst-case execution time (WCET), and best-case execution time (BCET) from these samples. Given the times from Table 1, the overhead incurred by the memory-centric scheduler (see Formula (14) with $N = 4$) is less than 6.028 μs. Compared to the worst-case execution times of the memory phases in the real-world application benchmarks (in the following section), the overhead represents less than 6.5%. Moreover, the hypervisor-related overheads have a small impact on the schedulability, as will be shown in Section 6.3.

Table 1 Jailhouse hypercall, Interprocessor Interrupt (IPI), and arbitration overheads (in ns).

	AVG	STD	WCET	BCET
Hypercall	1265.83	191.50	3129	709
IPI	979.65	80.95	1999	935
Arbitration	127.79	8.94	225	64

6.2 Real-World Benchmarks

We performed benchmarks for the full system model in two stages. First, we analyze the maximum interference of RTOSs on the highest priority RTOS with and without our approach. After that, we apply our system model and perform response time benchmarks for two application tasks on each of the cores.

To analyze the interference of parallel prefetching operations, we let two cores prefetch 448 KB of memory (equivalent to the memory size of the benchmarks used below and small enough to fit into a cache partition, which has 512 KB). The prefetching frequency for the high priority core is 300 Hz. The low priority core runs an interfering task whose main loop consists only of prefetching to generate the highest possible interference from a PREM-ized task set running on an RTOS. The memory prefetching time is measured on the high priority core using the ARM physical counter-timer. The experiment is run for 90 000 task activations.

Figure 9 shows the worst-case prefetching time on the high priority core. The results are shown broken down by the enabled (*ours*) or disabled (*legacy*) memory-centric scheduler, and

with the low priority core being idle (*solo*) or generating worst-case interference (*stress*). With our approach preventing unpredictable interference, the prefetching time is bounded, and the performance is not significantly impacted by the memory arbitration related overheads.

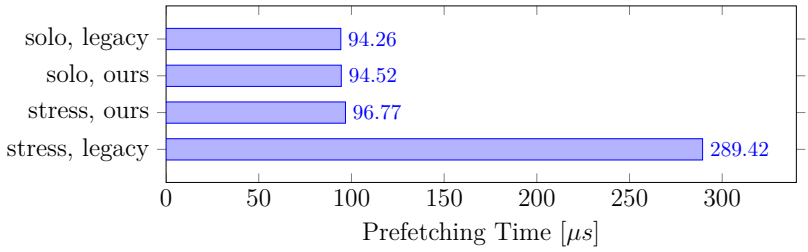
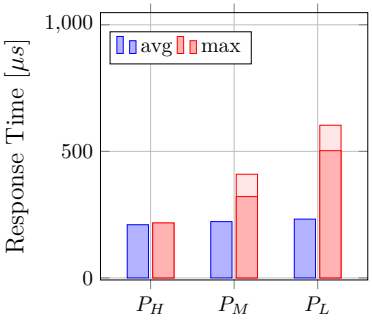


Figure 9 Comparison of prefetching times for a task running on the highest priority core.

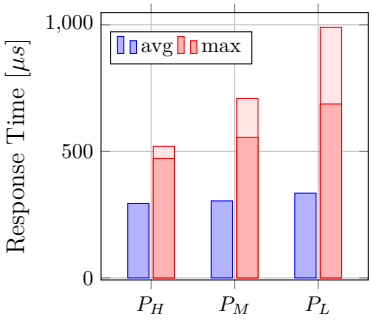
For the second stage of our experiments, we run a task set consisting of two application benchmarks from *TACLeBench* [18], which are compliant with the two-phase PREM. The first task computes an average of integer values and has a high priority (in the task set). It is released with a frequency of 300 Hz. The second task, with a low priority (in the task set), computes a SHA-1 hash and is activated with a frequency of 100 Hz. Both tasks are released synchronously and use 448 kB of data. We run the benchmark on three cores, each with a different processor priority. The measurement is taken identically to the first experiment over 90 000 task activations for the high priority task and 30 000 task activations for the low priority task.

When the response time is measured on the high priority core, the medium priority core runs the interference task described in the first experiment while the low priority core is idle. When the measurement is done on the medium priority core, the high priority core runs the benchmark task set, while the low priority core runs the interference task. The high and medium priority cores both run the benchmark task set when the response time is measured on the low priority core.

In Figure 10, the average and worst-case response time for the high and low priority task is shown. The analytical worst-case response time for the high priority task takes the synchronous release of the tasks into account and thus does not incorporate the blocking from the low priority task scheduled on the same processor.



(a) High priority task.



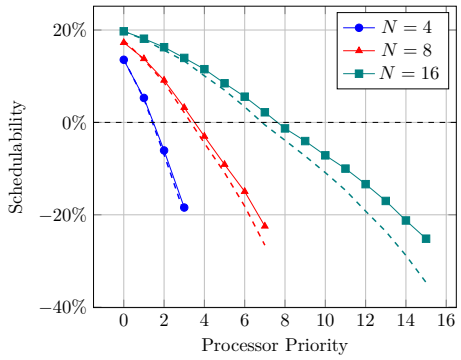
(b) Low priority task.

Figure 10 Response times for the *TACLeBench* benchmark task set executed on the high (P_H), medium (P_M), and low priority (P_L) core. The figures show the average (blue), measured (red), and analytical (light red) worst-case response time.

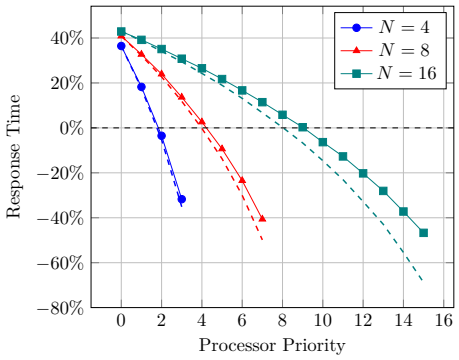
6.3 Schedulability Analysis Evaluation

We generate a sporadic task set with a given number of tasks and a given utilization. We randomly generate period values in the range $[10\,000, 100\,000]$ microseconds with log-uniform distribution, as suggested in [16]. We also use the Emberson et al. unbiased method [16] to generate random task-utilization values that sum to the requested total utilization. Based on the application profiles provided in [61], the ratio between memory and computation phases is generated randomly with a uniform distribution in the range $[0.05, 0.20]$. Tasks are assumed to have implicit deadlines and are assigned priorities based on the Rate-Monotonic priority assignment policy. We consider platforms with $N \in \{4, 8, 16\}$ processors. We compare fixed-priority scheduling to contention-based scheduling [59] (i.e., each processor is assigned $1/N$ of the memory bandwidth) and round-robin scheduling with the quantum size 1 for each processor (bandwidth is shared equally among all processors having pending memory requests).

Schedulability and response time. We investigate the impact of the processor priority on the task's worst-case response time and the improvement of the task schedulability. Each processor is assigned 8 tasks with a total utilization of 0.6. The memory utilization is scaled down proportionally to the processor count (i.e., the bandwidth utilization in the system is constant). 10 000 task sets were randomly generated. If a task was not schedulable under a given policy, its worst-case response time was considered double of the worst-case response time under any other scheduling policy. The results are shown in Figure 11. The baseline represents contention-based scheduling [59]. Round-robin is omitted in Figure 11: for schedulability, its improvement is 1%, and for the reduction of worst-case response times 2%. Additionally, we integrated the scheduler overheads (see Formula 14 and Section 6.1) in the response time analysis by inflating the generated WCET of memory phases with the overheads shown in Table 1. The dashed and the plain lines represent the results, respectively, with and without overheads. As expected, the fixed-priority policy improves the schedulability and reduces the response times on the high priority processors at the expense of the low priority processors. The tasks running on the low priority processors suffer more from the scheduling overheads as their memory phases are preempted more frequently.



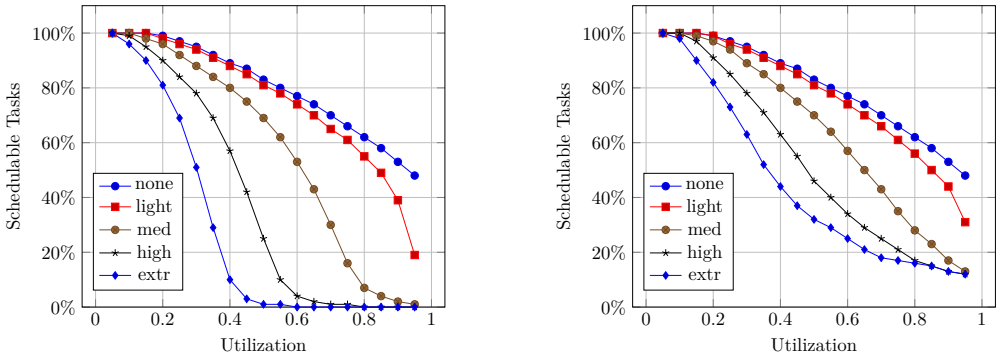
(a) Schedulability improvement.



(b) Response time reduction.

Figure 11 Percentage improvement in task schedulability and response time with respect to contention-based scheduling as a function of processor priority. The dashed lines take into account the measured memory arbitration overheads shown in Table 1.

Memory interference. We measure the impact of the task memory stall (i.e., the ratio between a task's memory phase duration and WCET - m_i/e_i) on the task schedulability for an $N = 4$ processor system. For each utilization within the range $[0.05, 1]$, we generate 5 types of sets with 4 tasks each. The types differ in the range of random stall distributions [61]: $[0.00, 0.00]$ (none), $[0.00, 0.05]$ (light), $[0.05, 0.20]$ (med), $[0.20, 0.40]$ (high) and $[0.40, 0.65]$ (extr). For instance, a task with a memory stall of 0.2 spends 20% of its WCET in the memory phase and 80% in the computation phase. Then, we take 4 sets with the same utilization and the same stall distribution, assign each set to one processor and test the schedulability with contention-based and fixed-priority memory-centric scheduling. Our test covers more than 550 randomly generated task sets per utilization and memory stall type. The percentage of schedulable tasks for contention-based scheduling and for fixed-priority scheduling are shown respectively in Figure 12(a) and Figure 12(b). Both algorithms can schedule only a very limited number of memory-intensive tasks. In fixed-priority scheduling, the tasks running on the highest priority processor are isolated from the interference generated by tasks running on the low priority processors. For this reason, the curves on Figure 12(b) converge to a higher value than the curves on Figure 12(a). In the next experiment, we show how grouping and prioritizing of high-rate memory-bound tasks can improve the overall schedulability of the system.



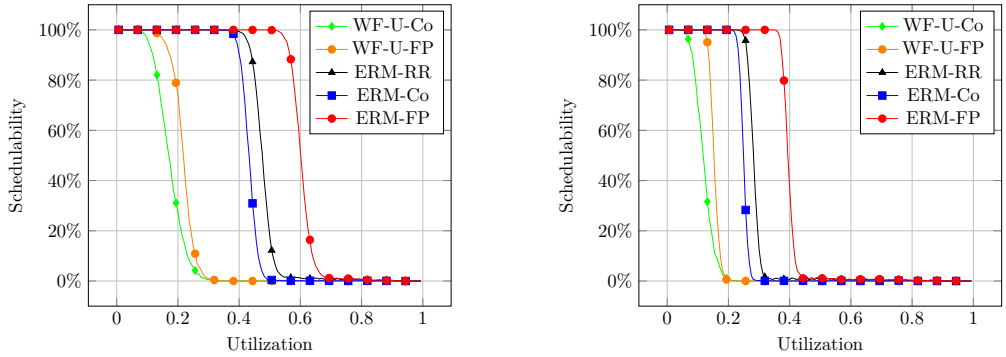
(a) Contention-based scheduling.

(b) Fixed-priority memory-centric scheduling.

Figure 12 Schedulability ratios for contention-based and fixed-priority memory-centric scheduling.

Heuristics for task assignment. We run our schedulability test for randomly generated task sets partitioned among the processors using various heuristics. We assume a multiprocessor system of $N = 16$ processors and a set of $N \cdot 8 = 128$ tasks. More than 5000 task sets are generated for each utilization factor within $[0, N]$ with a step size of 0.1. To partition the tasks among the processors, we consider many widely used heuristics and propose a new heuristic matching to our scheduling model. The standard heuristics include *First Fit (FF)*, *Next Fit (NF)*, and *Worst Fit (WF)*, each combined with an initialization step sorting the tasks in decreasing or increasing order by utilization factor or period, or without sorting. The best results among the standard heuristics are obtained for *WF* with tasks sorted in decreasing order of utilization. The heuristic that we propose first sorts the tasks in increasing order of their periods and then allocates the tasks using *FF* with the processors capacity reduced to U/N to ensure the proper load balancing (similarly to [63]). This heuristic clusters the tasks with shorter periods on the processors with higher priorities: the tasks do not suffer interference in memory phases from the low priority processors, and their blocking factor (due to the non-preemptive execution on the same processor) is not increased by the tasks that may have longer periods and longer worst-case execution times. The latter are

placed on the processors with lower priority since they can potentially accommodate more interference in their memory phases within the longer deadlines. The intuition is analogous to *Rate Monotonic* priority assignment.



(a) Normal task memory stall ($[0.05, 0.20]$). **(b)** High task memory stall ($[0.20, 0.40]$).

Figure 13 Schedulability ratio for different task partitioning heuristics.

In our evaluation, the fixed-priority policy performs best in every case, with the exception of the case where all policies fail for *FF* at $1/N$ due to the overcharge on the first processors. The results with the normalized utilization are shown in Figure 13. Due to space limitations, we report only *WF* with the utilization sorted in decreasing order (*WF-U*), which is the best among all the standard heuristics, and our Rate Monotonic like heuristic with equally distributed load (*ERM*) for round-robin (*RR*), contention-based (*Co*) and Fixed-Priority (*FP*) scheduling. The latter shows a maximal gain of 20% in the schedulability compared to the former two policies.

7 Related Work

Scratchpad-based memory-centric scheduling. Scratchpad-based systems allow overlapping of DMA memory transactions and CPU execution, resulting in better overall schedulability compared to cache-based systems. This approach was combined with partitioned [20, 48, 53, 57] and global scheduling [3]. Melani et al. [37] proposed exact response time analysis for fixed-priority scheduling on a single core with a fully preemptive DMA engine.

Cache-based memory-centric scheduling. The *PREM* was first introduced in [40, 41] to co-schedule memory accesses from CPU and I/O on a single core with multi-level caches. In [59], the *PREM* was applied for partitioned multiprocessor systems under TDM main memory accesses. To better utilize the assigned TDM-slot, the scheduling policy of each processor raises the priority of its pending memory phases above the priority of the computation phases. The concept of prioritizing the memory phases was also applied in global scheduling [60].

Schedulability analysis of global fixed-priority non-preemptive scheduling was extended for PREM compliant tasks [2, 33]. Non-preemptive execution permits to share the same cache partition among different tasks, but the scheduling algorithms proposed in the cited works are unaware of the tasks' cache footprints. Gaun et al. [21] integrated this constraint into the schedulability analysis for global scheduling. Nonetheless, without explicit cache locking, it is not sufficient to check that the tasks' data footprints are smaller than the available cache space to guarantee that their cache regions are not overlapping. In the partitioned scheduling considered in our work, the inter-core cache interference can be avoided by design with the proposed assignment of tasks to processors.

Matějka et al. [36] generate a static offline schedule for PREM-compliant tasks. The authors target COTS platforms with very similar characteristics to ours but recognize to disregard the inter-core eviction in L2 shared cache. The inter-core cache interference can be accounted for in the analysis proposed by Xiao et al. [58].

The closest related work was presented recently in [46]. The key differences to our work are that we do not consider hardware cache partitioning, we implement memory arbitration at the hypervisor-level, and our implementation, as well as schedulability analysis, are not limited to a single task per processor.

Cache coloring. Cache-coloring-based cache management frameworks for multiprocessor systems leverage virtualization [26, 32, 35, 38]. Our approach for cache partitioning is based on cache coloring implemented in *Jailhouse* hypervisor [20, 28]. Coloring can also be used for controlling DRAM bank allocation [64]. However, on platforms with address randomization (e.g., Nvidia TX1), this solution may be impracticable [36].

Time division multiplexing. Wandeler and Thiele [56] find an optimal TDM time slot and cycle assignment for systems characterized by arrival and service curves. Our model could be plugged into their framework by providing such curves. Hamann and Ernst [22] apply an evolutionary algorithm to optimize the time slot assignments of tasks mapped to TDM-scheduled resources. Worst-case execution time analysis can be extended to take the TDM parameters into account [47]. Several dynamic arbitration schemes based on TDM were recently proposed to avoid or reduce the resource contention in multiprocessor [1, 23] and network-on-chip [29] systems.

8 Conclusions, Limitations, Future Work

Conclusions. In this work, we proposed a memory-centric fixed-priority scheduler. Our solution does not rely on hardware features for predictable memory management of any kind, which is the case for many COTS multiprocessor platforms today. Furthermore, we were able to go beyond the static allocation of bus mastership as in the classical TDM approach, reducing the worst-case response time for high priority tasks. Our implementation shows that it is possible to achieve dynamic memory arbitration at the hypervisor level with relatively small overheads. Moreover, to the best of our knowledge, we are the first to propose a task-to-processor assignment algorithm integrated with the PREM.

Limitations. We have ignored any interference effects due to I/O devices using the shared memory hierarchy. This may not be of concern on COTS-based platforms that feature a dedicated I/O bus (e.g., MPC5777M, MPC5746M). Otherwise, the I/O interference should be taken into account into the worst-case memory phase time or other approaches for isolating I/O (e.g., [41]) or accelerators (e.g., GPU [10]) should be integrated. Since the OS and its tasks share the same cache partition, a minimal degree of cache pollution can occur. Additionally, the memory region of the hypervisor is not colored. We made the assumption that task data cannot exceed the size of the cache partition. Prior works [33,37,53,57,60] made a similar assumption. While the proposed implementation can already support multi-segment tasks, the schedulability analysis should be revisited.

Future work. As part of our ongoing work, we are investigating limited preemption for tasks assigned to the same processor and extending our model for three-phase PREM tasks.

References

1 A. Agrawal, G. Fohler, J. Freitag, J. Nowotsch, S. Uhrig, and M. Paulitsch. Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, pages 2:1–2:22, 2017.

2 A. Alhammad and R. Pellizzoni. Schedulability Analysis of Global Memory-predictable Scheduling. In *2014 Inter. Conference on Embedded Software (EMSOFT)*, pages 1–10, October 2014.

3 A. Alhammad, S. Wasly, and R. Pellizzoni. Memory Efficient Global Scheduling of Real-Time Tasks. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 285–296, April 2015.

4 ARM. Primecell Level 2 Cache Controller (PL310) - Technical Reference Manual, Revision: r2p0. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0246c/index.html`. Accessed: 2019-10-07.

5 S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo. Memory-Aware Scheduling of Multicore Task Sets for Real-Time Systems. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 300–309, August 2012.

6 M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, and T. Nolte. Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24, July 2016.

7 B. B. Brandenburg and M. Gül. Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 99–110, November 2016.

8 P. Burgio, A. Marongiu, P. Valente, and M. Bertogna. A memory-centric approach to enable timing-predictability within embedded many-core accelerators. In *2015 CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*, pages 1–8, October 2015.

9 G. C. Buttazzo, M. Bertogna, and G. Yao. Limited Preemptive Scheduling for Real-Time Systems. A Survey. *IEEE Transactions on Industrial Informatics*, 9(1):3–15, February 2013.

10 N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna. SiGAMMA: Server Based Integrated GPU Arbitration Mechanism for Memory Accesses. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems (RTNS)*, pages 48–57, 2017.

11 J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, D. de Niz, and G. von der Brüggen. Many Suspensions, Many Problems: A Review of Self-Suspending Tasks in Real-Time Systems. *Real-Time Systems*, 55(1):144–207, January 2019.

12 C. Dall, S.W. Li, J. T. Lim, and J. Nieh. ARM Virtualization: Performance and Architectural Implications. *SIGOPS Oper. Syst. Rev.*, 52(1):45–56, August 2018.

13 R. I. Davis and A. Burns. A Survey of Hard Real-time Scheduling for Multiprocessor Systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

14 R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, April 2007.

15 G. Durrieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch. Predictable Flight Management System Implementation on a Multicore Processor. In *Embedded Real Time Software (ERTS)*, February 2014.

16 P. Emberson, R. Stafford, and R.I. Davis. Techniques For The Synthesis Of Multiprocessor Tasksets. In *WATERS at the Euromicro Conference on Real-Time Systems*, pages 6–11, July 2010.

17 Evidence. Erika Enterprise RTOS v3, October 2018. Accessed: 2019-10-16. URL: `http://www.erika-enterprise.com/`.

18 H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R.B. Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A Benchmark Collection to

Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 2:1–2:10, 2016.

19 G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *ACM Comput. Surv.*, 48(2):32:1–32:36, November 2015.

20 G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 27:1–27:25, 2019.

21 N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware Scheduling and Analysis for Multicores. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, pages 245–254, 2009.

22 A. Hamann and R. Ernst. TDMA Time Slot and Turn Optimization with Evolutionary Search Techniques. In *Design, Automation and Test in Europe*, pages 312–317, March 2005.

23 F. Hebbache, M. Jan, F. Brandner, and L. Pautet. Shedding the Shackles of Time-Division Multiplexing. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 456–468, December 2018.

24 R. E. Kessler and M. D. Hill. Page Placement Algorithms for Large Real-indexed Caches. *ACM Trans. Comput. Syst.*, 10(4):338–359, November 1992.

25 H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding Memory Interference Delay in COTS-based Multi-Core Systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, April 2014.

26 H. Kim and R. Rajkumar. Real-Time Cache Management for Multi-Core Virtualization. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10, October 2016.

27 J. Kiszka, V. Sinitsyn, H. Schild, and contributors. Jailhouse Hypervisor. Siemens AG on GitHub, `https://github.com/siemens/jailhouse`, 2018. Accessed: 2019-10-10.

28 T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14, April 2019.

29 A. Kostrzewa, S. Saidi, L. Ecco, and R. Ernst. Flexible TDM-Based Resource Management in on-Chip Networks. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS)*, page 151–160, 2015.

30 J. P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 201–209, December 1990.

31 J. Liedtke, H. Haertig, and M. Hohmuth. OS-Controlled Cache Predictability for Real-Time Systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 213–, 1997.

32 R. Ma, W. Ye, A. Liang, H. Guan, and J. Li. Cache Isolation for Virtualization of Mixed General-purpose and Real-time Systems. *J. Syst. Archit.*, 59(10):1405–1413, November 2013.

33 C. Maia, G. Nelissen, L. Nogueira, L. M. Pinho, and D. G. Pérez. Schedulability Analysis for Global Fixed-Priority Scheduling of the 3-Phase Task Model. In *IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, August 2017.

34 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-Time Cache Management Framework for Multi-core Architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, April 2013.

35 J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, pages 3:1–3:14, 2020.

36 J. Matějka, B. Forsberg, M. Sojka, Z. Hanzálek, L. Benini, and A. Marongiu. Combining PREM Compilation and ILP Scheduling for High-performance and Predictable MPSoC Execution. In

Proc. of the 9th Inter. Workshop on Programming Models and Applications for Multicores and Manycores (PMAM), pages 11–20, 2018.

37 A. Melani, M. Bertogna, R. I. Davis, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo. Exact Response Time Analysis for Fixed Priority Memory-Processor Co-Scheduling. *IEEE Transactions on Computers*, 66(4):631–646, April 2017.

38 P. Modica, A. Biondi, G. Buttazzo, and A. Patel. Supporting Temporal and Spatial Isolation in a Hypervisor for ARM Multicore Platforms. In *2018 IEEE International Conference on Industrial Technology (ICIT)*, pages 1651–1657, February 2018.

39 M. Nasri, G. Nelissen, and B. B. Brandenburg. A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, pages 9:1–9:23, 2018.

40 R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, and M. Caccamo. Predictable Execution Model: Concept and Implementation. Technical report, University of Illinois at Urbana-Champaign, June 2010. URL: `http://hdl.handle.net/2142/16605`.

41 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 *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, April 2011.

42 L. T. X. Phan, M. Xu, and I. Lee. Cache-aware Interfaces for Compositional Real-time Systems: Invited Paper. *SIGBED Rev.*, 13(3):52–55, August 2016.

43 I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Design, Automation and Test in Europe Conference and Exposition (DATE)*, pages 1484–1489, 2007.

44 R. Rajkumar. Real-Time Synchronization Protocols for Shared Memory Multiprocessors. In *Proceedings.,10th International Conference on Distributed Computing Systems*, pages 116–123, May 1990.

45 R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer. Look Mum, no VM Exits! (Almost). *Proceedings of the 13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, May 2017.

46 J. M. Rivas, J. Goossens, X. Poczekajlo, and A. Paolillo. Implementation of Memory Centric Scheduling for COTS Multi-Core Real-Time Systems. In *31st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 7:1–7:23, 2019.

47 J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *28th IEEE International Real-Time Systems Symposium (RTSS)*, pages 49–60, December 2007.

48 B. Rouxel, S. Skalistis, S. Derrien, and I. Puaut. Hiding Communication Delays in Contention-Free Execution for SPM-Based Multi-Core Architectures. In *31st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 25:1–25:24, 2019.

49 M. Schoeberl, L. Pezzarossa, and J. Sparsø. A Multicore Processor for Time-Critical Applications. *IEEE Design Test*, 35(2):38–47, April 2018.

50 L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

51 V. Shivappa. x86: Intel Cache Allocation Technology support. `https://lwn.net/Articles/622893/`. Accessed: 2019-10-07.

52 M. R. Soliman, G. Gracioli, R. Tabish, R. Pellizzoni, and M. Caccamo. Segment Streaming for the Three-Phase Execution Model: Design and Implementation. In *IEEE Real-Time Systems Symposium (RTSS)*, December 2019.

53 R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, April 2016.

54 C. Tessler and N. Fisher. BUNDLE: Real-Time Multi-threaded Scheduling to Reduce Cache Contention. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 279–290, November 2016.

55 K. Tindell and J. Clark. Holistic Schedulability Analysis for Distributed Hard Real-time Systems. *Microprocess. Microprogram.*, 40(2-3):117–134, April 1994.

56 E. Wandeler and L. Thiele. Optimal TDMA Time Slot and Cycle Length Allocation for Hard Real-Time Systems. In *Asia and South Pacific Conference on Design Automation, 2006.*, pages 6 pp.–, January 2006.

57 S. Wasly and R. Pellizzoni. Hiding Memory Latency Using Fixed Priority Scheduling. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 75–86, April 2014.

58 J. Xiao, S. Altmeyer, and A. Pimentel. Schedulability Analysis of Non-preemptive Real-Time Scheduling for Multicore Processors with Shared Caches. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 199–208, December 2017.

59 G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48(6):681–715, November 2012.

60 G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo. Global Real-Time Memory-Centric Scheduling for Multicore Systems. *IEEE Trans. on Computers*, 65(9):2739–2751, September 2016.

61 G. Yao, H. Yun, Z. P. Wu, R. Pellizzoni, M. Caccamo, and L. Sha. Schedulability Analysis for Memory Bandwidth Regulated Multicore Real-Time Systems. *IEEE Transactions on Computers*, 65(2):601–614, February 2016.

62 Y. Ye, R. West, Z. Cheng, and Y. Li. COLORIS: A Dynamic Cache Partitioning System Using Page Coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392, August 2014.

63 Y. Ye, R. West, J. Zhang, and Z. Cheng. MARACAS: A Real-Time Multicore VCPU Scheduling Framework. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 179–190, November 2016.

64 H. Yun, R. Mancuso, Z.P. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, April 2014.