Implementation of Memory Centric Scheduling for COTS Multi-Core Real-Time Systems

Juan M. Rivas[1]
PARTS Research Centre, Université libre de Bruxelles, Brussels, Belgium
jrivasco@ulb.ac.be

Joël Goossens
PARTS Research Centre, Université libre de Bruxelles, Brussels, Belgium
joel.goossens@ulb.ac.be

Xavier Poczekajlo
PARTS Research Centre, Université libre de Bruxelles, Brussels, Belgium
xavier.poczekajlo@ulb.ac.be

Antonio Paolillo
HIPPEROS S.A., Louvain-la-Neuve, Belgium
antonio.paolillo@hipperos.com

Abstract

The demands for high performance computing with a low cost and low power consumption are driving a transition towards multi-core processors in many consumer and industrial applications. However, the adoption of multi-core processors in the domain of real-time systems faces a series of challenges that has been the focus of great research intensity during the last decade. These challenges arise in great part from the non real-time nature of the hardware arbiters that schedule the access to shared resources, such as the main memory. One solution proposed in the literature is called Memory Centric Scheduling, which defines a separate software scheduler for the sections of the tasks that will access the main memory, hence circumventing the low level unpredictable hardware arbiters. Several Memory Centric schedulers and associated theoretical analyses have been proposed, but as far as we know, no actual implementation of the required OS-level underpinnings to support dynamic event-driven Memory Centric Scheduling has been presented before. *In this paper* we aim to fill this gap, targeting cache based COTS multi-core systems. We will confirm via measurements the main theoretical benefits of Memory Centric Scheduling (e.g. task isolation). Furthermore, we will describe an effective schedulability analysis using concepts from distributed systems.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases real-time, multi-core, memory centric, predictability, implementation, rtos

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.7

Funding *Juan M. Rivas*: This research was funded by Wallonia Region (Belgium) BEWARE grant PARTITA (1610375).

1 Introduction

Advancements in the manufacturing process of integrated electronics, in addition to the sheer size of the general computing market, are increasingly widening the offer and lowering the costs of commercial off-the-shelf (COTS) multi-core processors. While these commercial processors are generally designed with *average* performance in mind, their wide availability and low cost are pushing their adoption into real-time applications where a different set of requirements such as predictability and worst-case guarantees are needed.

[1] Corresponding author

The main challenge of considering COTS multi-core processors for real-time applications can be found in the way some resources are shared between the cores. For instance, the access to the shared main memory is usually scheduled by hardware arbiters that are generally not designed with real-time or predictability considerations in mind. In essence, all the jobs concurrently accessing the shared memory can interfere with each other, thus adding delays to the execution time that are potentially large and very difficult to tightly bound. This interference can be compounded by other inter-dependent sources such as the shared cache: a task could evict a cache line of another task in a different core, which could produce further accesses to the shared memory and interferences down the line. In conclusion, in a multi-core processor the execution time of any task can be affected by any other concurrent task, regardless of priorities or criticalities. This leads to two main problems: (1) the task execution times are inflated due to interferences from other cores, and (2), the worst-case execution time (WCET) estimations tend to be greatly inflated due to the difficulty to predict these inter-core interferences.

This mismatch between the predictability requirements needed in real-time systems and the performance characteristics of available multi-core processors has triggered an intense research effort in the real-time community during the last decade [17]. One line of research proposes additional layers of software and associated mathematical analyses to add predictability to the commercial multi-core platforms. An example of such approach is based on the new task model called the PRedictable Execution Model (PREM) [31]. With PREM, each task code is explicitly divided into coarse grained phases that will access the shared memory (memory phases) and phases that will operate exclusively on cached data and instructions (execution phases). Typically, the memory phases are composed of prefetch or load instructions that copy into the cache the data and instructions needed by a subsequent execution phase, or by write-back instructions to copy updated data from the cache to the main memory.

Leveraging the PREM model, a new type of scheduling scheme called Memory Centric [38] defines high level schedulers for the memory phases with the aim to limit or avoid concurrent memory phases. This way, the contention in the shared memory subsystem can be solved by software, avoiding relying on the low level unpredictable arbiters. Several previous works have proposed different variations of Memory Centric schedulers, a selection of which will be briefly discussed in Section 2. Notwithstanding this body of work, and as far as we know, all these proposals are either theoretical works or implementations relying on static or time triggered scheduling.

The main objective of this paper is to complement those previous approaches, by implementing dynamic Memory Centric Scheduling in an actual Real-time Operating System (RTOS). While the basic ideas to sustain our implementation do not depend on any particular RTOS, we choose to target HIPPEROS [28]. HIPPEROS is built from the ground-up to support multi-core processors and employs an asymmetric master-slave architecture in which the schedulers run on a dedicated core, called the *master core*. By exploiting this architecture the overheads of executing the schedulers are concentrated in the *master core*, while the *slave cores* can execute tasks with a *baremetal level* of overheads.

To tease the outcome of this paper, Figure 1 shows the execution times of a periodic task during a span of 40 seconds. From 0 to 10 seconds the task runs alone in the system, and then a new task is added to a new core every 10 seconds (each core has at most one task). The blue line (contention) shows how the execution time of the task increases any time a new task is added, indicating that it is being affected by interferences from other cores. The orange line (labelled as "this paper") shows the execution times of the same task, but using the implementation we provide in this paper. We can see that now the task execution time remains constant, thus achieving a level of isolation from other tasks.

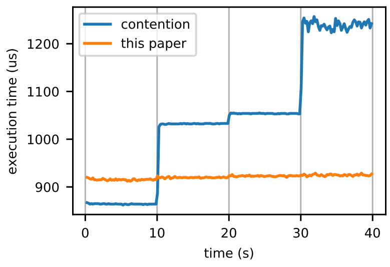

Figure 1 Execution times of a task with different number of competing tasks.

Contributions of this paper

- **Implementation of dynamic Memory Centric Scheduling in HIPPEROS**. This includes a kernel-level scheduler and a user-level API. We target fixed task priority and preemptive memory phases, that can be dynamically invoked either synchronously or asynchronously.
- **An evaluation study** to confirm with measurements the theoretical benefits of Memory Centric Scheduling, namely an execution free of interferences in the main memory.
- **A schedulability analysis** defined by exploiting the similarities between the task model used in Memory Centric Scheduling and the distributed task model [36]. We compare the analytical results with actual measurements.

The paper is organized as follows: Section 2 provides more background on works related to handling interferences in the main memory of multi-core systems, with a focus on Memory Centric Scheduling, and a brief introduction to the HIPPEROS RTOS main relevant features. Section 3 presents a description of the system model and general hardware characteristic assumptions adopted throughout the paper. Section 4 presents our implementation of Memory Centric Scheduling in HIPPEROS. Section 5 describes our schedulability analysis based on distributed systems. Section 6 shows the results of the extensive evaluation performed. Finally, Section 7 provides the main conclusions we have reached in this work, and hints at some general research paths that we could follow in the future.

2 Background and Related Work

In this section we will contextualize this paper in relation to other similar previous works, focusing on approaches that tackle the problem of handling the interferences in the main memory. In Section 2.1 we will briefly describe the basic elements of Memory Centric Scheduling, also providing a selection of papers that develop it. Section 2.2 is dedicated to discuss other approaches similar to Memory Centric Scheduling. Finally, Section 2.3 provides a description of the main relevant characteristics of the RTOS we target in this paper, HIPPEROS.

2.1 Memory Centric Scheduling

In the context of this paper we define Memory Centric Scheduling (MCS) as a scheduling framework of real-time tasks that complies with the following characteristics:

(a) The tasks follow a PREM'like model [31]: tasks have two possible coarse grained states or phases: memory phases (M-Phases) in which the task will access the shared main memory; and execution phases (E-Phases) in which the task operates on cached data and instructions with no access to the shared memory.

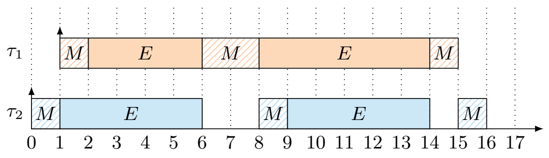

Figure 2 Simple example of Memory Centric Scheduling with two tasks.

(b) There exists a system wide scheduler for the memory phases (memory phases scheduler, MPS) that dynamically decides which memory phase can execute, hence effectively controlling the access to the shared memory.

If the MPS limits the number of concurrent memory phases to 1, the low level non real-time hardware arbiters that control the access to the shared memory are effectively bypassed, avoiding any unpredictable interference delays they could produce. Figure 2 shows an example of MCS with 2 tasks, in which each task executes in a different core, memory phases and execution phases are labeled with M and E respectively, and vertical arrows indicate the task activation. Note that at any time there is at most one memory phase executing.

Memory Centric Scheduling was first introduced by Yao et al [38], proposing a TDMA schedule for the memory phases. That paper also introduces the concept of memory promotion, by which the memory phases are given a higher priority than the execution phases to seek a better schedulability. The work in [9] studies with simulations different partitioned scheduling policies for MCS, and reaches the conclusion that non-preemptive least-laxity first memory phases is the best approach. The papers [3, 22] propose static schedules of the memory and execution phases. MCS with global scheduling has also been studied [2, 39].

The authors of [37] propose a mechanism to hide the latencies of the memory phases by executing them as background DMA transfers. This initial work targets single-core systems, but is later extended for multi-core systems in [4]. These two papers assume special hardware such as double ported scratchpad memories, or a cache based system that supports DMA transfers from the main memory to the cache. The paper [35] presents an integration of MCS in an RTOS, relying on scratchpad memory and implementing the memory phases with DMA transfers scheduled with TDMA.

In this paper we provide an actual implementation of Memory Centric Scheduling in an RTOS (HIPPEROS), where the memory phases can be dynamically invoked. We will also describe latency hiding techniques that do not rely on special hardware. Additionally we will adapt existing analysis techniques for real-time distributed systems, and compare it with our measurements.

2.2 Alternative approaches

Memory bandwidth regulators are usually mentioned as an alternative to MCS. This approach assigns per-core memory bandwidth reservation budgets that do not overload the memory bus. Therefore, each core can run with a guaranteed memory bandwidth, independently on the number of other active cores. A clear advantage of this approach is that it is transparent to the tasks, i.e., no task modifications are needed. On the other hand, with MCS, once a task is granted access to the memory, it can enjoy full access to its bandwidth.

Examples of this approach implemented in software are MEMGUARD [42] or the Multi-Resource Server [10]. A challenge of these techniques is to find an optimal budget assignment per core. A framework called Single Core Equivalence [20] proposes a static and even budget assignment, which is complemented by a mechanism to lock in cache the most frequently used pages [19], and a tool called PALLOC [41] that allocates pages to specific memory banks to allow a certain level of parallelism in the main memory. Subsequent works allow uneven memory bandwidth budget allocations [21, 40, 1, 8]. Recently ARM presented the Memory Partitioning and Monitoring (MPAM) feature in the Armv8.4-A specification [7], which allows to partition the memory bandwidth among different system components.

2.3 Multi-core real-time operating system: HIPPEROS RTOS

HIPPEROS [33] is a multi-core real-time operating system targeting high performance and safety-critical applications running on embedded systems.

The HIPPEROS kernel is a micro-kernel written to natively support multi-core and parallel systems. The kernel software architecture is distributed and asymmetric. This means that, when entering kernel mode, different cores are running different kernel code. In this case, only one core has a different code path than the others, and it is called the *master core*. The *master core* is responsible of the heavy kernel operations such as scheduling, memory management and task states. The other cores are the *slave cores* and have a lighter kernel mode of operations. The rationale behind that asymmetric design is that the *master core*, being dedicated to heavy management operations, is freeing the *slave cores* of that burden, letting them focus on user-mode task execution with few interferences. The *master core* can orchestrate task context switches remotely on the *slave cores* through inter-core interrupts. When a *slave core* requires to invoke the scheduler (for example when a task is completed and exits), it can do so by calling a *remote system call*, triggering an inter-core interrupt to the *master core* which is in turn effectively calling the scheduler module. The *slave core* waits the system call response from the *master core* by spinning the task in user-mode, making it easy to preempt the task when executing a context switch ordered by the *master core*.

This software architecture has the following advantages.

- For a kernel developer perspective, the design and the code base is *much easier to write and maintain*. It avoids having to lock every single structure which is shared among cores in a symmetric kernel design.
- As *slave cores* do not directly interact with each other but only with the *master core*, the asymmetric design has the effect of *reducing peak contention* on spinlocks when cores are trying to acquire them. In fact, the only spinlocks required are to implement the communication mechanisms between *slaves* and *master* to trigger the remote system call procedure and the context switches. Less contention allows for an *improved scalability* (in the worst case) when increasing the number of cores.
- The asymmetric design naturally partitions the data among cores, automatically making *a better use of private caches*. As the state of the system regarding tasks and scheduling is only maintained by the *master core*, this data must not be shared among cores and can stay in the private caches of the *master core*, allowing for both a faster execution of the *master core* kernel path and less cache misses on the *slave* side.

The concept of an asymmetric kernel for multi-core real-time systems was previously studied and validated in prior work [12]. Regarding HIPPEROS, it has been the target of previous contributions, such as its own parallel micro-kernel design [28], power-aware real-time scheduling [30], mixed-criticality scheduling [29] and hardware acceleration for embedded image processing applications [34, 16]. In this paper, the HIPPEROS RTOS is used to showcase an *in-kernel* implementation of the Memory Centric Scheduling approach.

3 System Model

3.1 Hardware Assumptions

We consider a typical commercial off-the-shelf (COTS) multi-core processor, composed of M identical cores with one or more levels of cache connected to the main memory via a shared memory bus. We assume that the last-level cache (LLC) is shared among the cores and employs a write-back policy: an LLC miss produces a share memory access to load a cache line, and possibly another memory bus access to write-back an evicted LLC line. Therefore we can establish that accesses to the shared bus only occur during LLC misses. We also assume that the shared LLC cache can serve concurrent hits from several cores with negligible interference delays. We will see in Section 6 that this assumption holds in our measurements. Additionally, we consider that only the cores can trigger a memory bus access (e.g. no peripheral DMA transfers are allowed). Figure 3 depicts a typical COTS multi-core processor, with 4 cores, private L1 caches and shared L2 cache.

3.2 Task Model

The cores execute a set of N preemptive sporadic tasks $\Gamma = \tau_1, ..., \tau_N$. In this paper we consider fixed task priorities (FTP) partitioned scheduling. We use a modified PREM task model that, in addition to the Execution Phases, defines two types of memory phases, called Prefetch Phases and Write-Back Phases. These phases operate in the following manner:

1. Prefetch Phase (P-Phase): tasks start with a memory phase called Prefetch Phase that prefetches (i.e. copies) the necessary data and instructions from the main memory to the cache.
2. Execution Phase (E-Phase): the task operates on data and instructions cached during the previous phase. As a result, no accesses to the main memory are triggered during this phase.
3. Write-back Phase (W-Phase): this is a memory phase that executes after an E-Phase to copy any updated data from the cache to the main memory. Additionally, the W-Phase also flushes all the previously prefetched cache lines, so any subsequent P-Phase could start with a known clean state.

For a predictable execution, cache lines prefetched during a P-Phase should only be evicted in a controlled manner during a W-Phase. Otherwise, any cache line that is inadvertently evicted could later trigger main memory accesses during the E-Phases, thus breaking the assumptions of the PREM model. Accidental cache line evictions can be produced by the task to itself (self-eviction), by other tasks in the same core (intra-core eviction) or from a different core (inter-core eviction). In this paper we propose to evade intra-core and

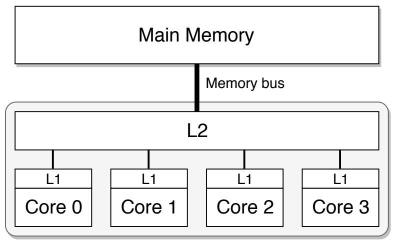

Figure 3 Simplified diagram of a typical commercial quad-core processor with 2 levels of cache.

inter-core evictions by assigning each task a partition of the shared cache. Cache partitioning is a commonly used solution that has been extensively studied [11, 23, 6]. Formally, we define A_i as the set of shared cache partitions assigned to τ_i, and BA_i the total size of the cache partitions assigned to τ_i. In this paper we assume that the cache partition mapping is performed offline. Section 4.4 provides more details about how we implemented cache partitioning for our particular platform.

Regarding the problem of self-evictions, in the context of this paper we can establish that they occur when the size of the data/instructions prefetched in a P-Phase exceeds the size of the cache partition assigned to the task. To handle this situation, we allow tasks to have several sequential batches of P-E-W phases, each one targeting a different portion of the memory with a size up to the size of the task cache partition. Formally, we define B_i as the memory requirement of τ_i, that is, the total size of data and instructions that τ_i needs to execute. If the memory requirement is larger than the task cache partition ($B_i > \mathrm{BA}_i$), the task must invoke several P-E-W phases to cover its memory requirement. We call each P-E-W phase triplet a `Section`.

We define $S_{i,j}$ as the j-th `Section` of task τ_i. Additionally, $P_{i,j}$, $E_{i,j}$ and $W_{i,j}$ are the P-Phase, E-Phase and W-Phase in section $S_{i,j}$, respectively. To avoid self-evictions, each `Section` operates on data/instructions with a size up to BA_i. Thus, the number of `Sections` needed by task τ_i is at least $\lceil \frac{B_i}{BA_i} \rceil$. Furthermore, we assume that the memory requirements of the tasks are fully available at task release, and that they can be partitioned. An example of a PREM task with two `Sections` is shown in Figure 4.

In this paper we target fixed priority scheduling of the memory phases. Accordingly, the tasks have two fixed priority values: its fixed task priority (with a core local scope) and its fixed memory phase priority (with a global scope). In HIPPEROS the task priority can be defined by an external configuration file, or by using the standard API's such as `OpenMP` or `pthreads`. To set the memory phase priority we have implemented a new system call (`memphase_priority_set`).

In single-core systems, the worst-case execution time (WCET) of a task is usually defined as an upper bound of its execution time when it runs *alone* in its core. This definition cannot be maintained in multi-core systems due to inter-core interference delays [17]. Accordingly, in this paper we define the WCET of task τ_i as an upper bound of its execution time when it is running alone in its core, and a known set of tasks are running in other cores. Similarly, we define the worst-case response time (WCRT) of τ_i as an upper bound of its execution time when it runs with a known set of tasks, in the same core and others. Thus, the WCRT includes the WCET of the task, and possible scheduling delays due to tasks in the same core.

This paper will define the system calls to start memory phases, but is not concerned about how to generate the code of the tasks, or how to determine the memory addresses to prefetch. We assume that the memory phases are either defined manually, or using some automatic tool [31, 18].

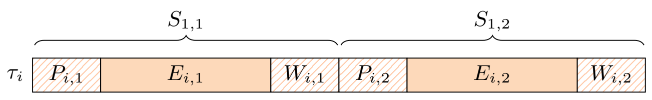

Figure 4 PREM task τ_i with 2 sections.

4 Implementation of Memory Centric Scheduling

In this section we describe the main contribution of this paper, which is a full implementation of Memory Centric Scheduling in an RTOS. In the next subsection we first lay-out the goals and intended behavior which will later shape the implementation.

4.1 Overview and Goals

We identify that an implementation of Memory Centric Scheduling is composed of two main interconnected components: (1) a scheduler for the memory phases and (2) an API for tasks to invoke the start and end of memory phases.

An initial approximation could understand a memory phase as a critical region protected by a mutex located in shared memory, in which the mutex lock and unlock functions would signal the start and end of the memory phases. While this approach can indeed assure that only one memory phase is executing at a time, it restricts their behavior to be non-preemptive. Consequently, any task could be temporarily blocked while requesting the start of a memory phase, independently of any priority assignment.

We propose an architecture similar to that of a mutex but with a *preemptive* nature to avoid those blocking times. Also, the memory phases are scheduled according to their fixed priorities and only one memory phase is allowed to execute at a time. By removing the interference in the shared memory and cache, and prioritizing the memory phases, the target of our implementation is that the response time of any task would only depend on the number of higher priority tasks executing in the same core, and the number of higher priority memory phases system-wide.

It is worth noting that the underlying hardware platform could force the use of non-preemptive memory phases. For example, the cache controller could not support performing a write-back cache operation (e.g. clean or invalidate) while a previous cache operation has yet not finished. This would not allow a W-Phase to preempt another W-Phase. For such situations we support non preemptive memory phases, which can however co-exist with preemptive ones.

In the next subsections we explain the implementation in more detail. Section 4.2 deals with the kernel-level memory phase scheduler and system calls to start and end a memory phase, while Section 4.3 presents the user level API that uses those system calls to request prefetch and write-back phases. In Section 4.4 we will describe how to use existing techniques to analyze our task model.

4.2 Kernel-level: memory phases scheduler (MPS)

As we described in the previous subsection, to implement the memory phases scheduler (MPS) we get inspiration from how mutexes operate, aiming to implement a mechanism that acts like a "*preemptive critical region*".

The two main components of the MPS are the system calls to invoke the start and end of a memory phase, called `memphase_start` and `memphase_end`, respectively. To illustrate how they operate we present a simple example with two tasks, τ_1 and τ_2, in which the memory phases of τ_1 have a higher priority to those of τ_2. Additionally, each task is mapped to a different core, and a third core handles the scheduler and MPS (HIPPEROS *master core*). Figure 5 shows the timeline of their execution, with a focus on the memory phase invoked by

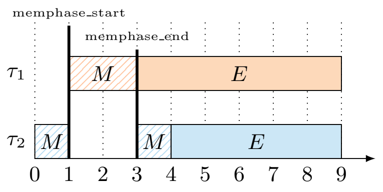

Figure 5 Simple example of scheduling of memory phases and needed system calls.

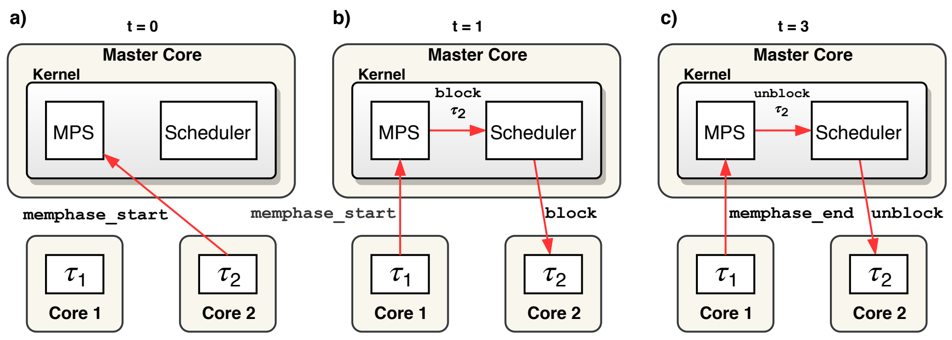

Figure 6 System calls to schedule the simple example of memory phases.

τ_1, while Figure 6 shows the different system calls involved and the interaction between the *master core* and the *slave cores*. It is worth noting that the *master core* is still available to execute user tasks.

First, at $t = 0$, τ_2 requests the start of a memory phase with `memphase_start`, which is immediately granted by the MPS as no other memory phase is executing at that time (Figure 6a). At this point the MPS stores τ_2 in its internal task queue that keeps track on the tasks that have a pending memory phase, ordered according to their priorities. Note that in this context, the MPS grants access to τ_2 by just letting it continue its execution. At $t = 1$, τ_1 requests the start of a memory phase (Figure 6b). At this time the MPS decides to grant access to τ_1 since its memory phase has a higher priority than the memory phase of τ_2. Accordingly, the MPS requests the system scheduler to block the execution of τ_2. At $t = 2$, τ_1 signals the end of its memory phase with the system call `memphase_end` (Figure 6c), which prompts the MPS to unblock τ_2 to let it finish its memory phase.

The pseudocode shown in Listing 1 describes the system call `memphase_start`. The variable "caller" is a pointer to the task requesting the memory phase, while "owner" is a pointer to the task currently executing a memory phase. Basically, the system call always add to the queue the "caller", and blocks the task with the lowest priority memory phase between "caller" and "owner", updating the "owner" when necessary. If "owner" is non preemptive, the "caller" is always blocked.

Similarly, Listing 2 shows a pseudocode describing `memphase_end`. When called, this function unblocks the next highest priority memory phase (if any), and assign it as the new "owner".

4.3 User level API

The previous sub-section presented the mechanisms to invoke and schedule memory phases. It is important to note that, in our implementation, the kernel is not concerned about the contents of the memory phases, or even if they access the main memory or not. We

Listing 1 `memphase_start` pseudocode.

```
memphase_start(nonpreemptive)
      caller = getCaller()
      owner = getFirst(queue)
      if (nonpreemptive)
            setnonpreemptive(owner)
      insert(caller, queue)
      if owner != NULL
            if nonpreemptivemp(owner)
                block(caller)
            else if memprio(caller) > memprio(owner)
                block(owner)
            else
                block(caller)
```

Listing 2 `memphase_end` pseudocode.

```
memphase_end()
      caller = getCaller()
      removeFirst(caller, queue)
      owner = getFirst(queue)
      if owner != NULL
            unblock(owner)
```

provide the semantics of a memory phase in user-space, by creating functions that employ the `memphase_start` and `memphase_end` system calls to protect prefetch and write-back instructions, defining prefetch and write-back phases respectively.

Two variants of this user-space API are implemented:

- **Synchronous memory phases (S-MP)** (Section 4.3.1).
- **Asynchronous memory phases (A-MP)** (Section 4.3.2).

4.3.1 Synchronous Memory Phases (S-MP)

Synchronous Memory Phases (S-MP) are memory phases that are actively executed by the tasks that request them. To illustrate S-MP, Figure 7 shows an example with three tasks τ_1, τ_2, τ_3 in decreasing order of fixed memory phase priority, with each task executing in a dedicated core, and τ_1 released one time instant after τ_2 and τ_3. For simplicity, the labels for the phases do not show the sub-indices. We can see that the tasks follow the sequence P-Phase → E-Phase → W-Phase, each one executed by the requesting task in its core. In the figure we can also see that the MPS decides at each time to schedule the highest priority memory phase, preempting memory phases when necessary.

We implement two functions to request the start of Prefetch and Write-back phases, called `h_memphase_prefetch` and `h_memphase_writeback` respectively. A simplified version of their code is presented in Listing 3 and 4, respectively. We can see that these functions use

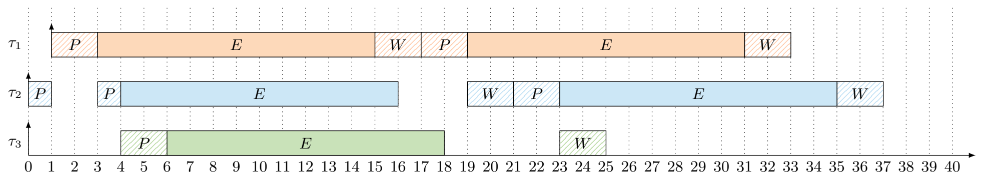

Figure 7 Taskset scheduled with synchronous memory phases (S-MP).

Listing 3 `memphase_prefetch` simplified code.

```
void h_memphase_prefetch(void *addr, size_t bytes) {
        memphase_start();
        prefetch(addr, bytes);
        memphase_end();
}
```

Listing 4 `memphase_writeback` simplified code.

```
void h_memphase_writeback(void *addr, size_t bytes) {
        memphase_start();
        writeback(addr, bytes);
        memphase_end();
}
```

the memory phase start and end system calls to protect the call to a prefetch or write-back function. The only requisite of the prefetch function is that they result in data/instructions copied to the task cache partition or private cache, while the write-back function must be able to copy the updated data into the shared memory and leave the cache partition in a clean state. These requirements represent basic functionalities that are commonly implemented in any commercial processor. Implementation details for our particular platform will be provided in Section 4.4.

In this initial version of the implementation we only target the prefetch and write-back of data (i.e. not instructions). The evaluation section will show how this limitation provides good results. Additionally, the current API is limited to contiguous memory prefetches and write-backs (i.e. defined by base address + memory size).

4.3.2 Asynchronous Memory Phases (A-MP)

An Asynchronous Memory Phase (A-MP) is a memory phase that is executed by a dedicated worker in the background, which we call the Memory Phase Worker (MPW). Crucially, the main benefit of A-MP is that the execution times of these background memory phases do not contribute to the execution times of the requesting tasks. It is worth noting that, contrary to previous proposals for background memory phases [37, 4], our implementation does not rely on any special hardware like scratchpad memories or support for DMA transfers from memory to cache, so it can be accomplished in a wider spectrum of available processors.

A-MP requires that the task cache partitions must be divided into two sub-partitions, which we call the A and B sub-partitions. Formally, given a task τ_i, we define A_i^A and A_i^B as its two sub-partitions, with $A_i = A_i^A \cup A_i^B$. The basic idea behind A-MP is that while an E-Phase is working on data from a given sub-partition, background memory phases are preparing the other sub-partition. By switching the executing sub-partition, we can effectively hide the execution times of the memory phases while continuously executing E-Phases. We extend the names of the phases to indicate in which sub-partition they are operating, e.g. P^A, E^A and W^A express a P, E and W Phases operating on sub-partition A, respectively.

The MPW is implemented as a task which just executes memory phases on behalf of other tasks, therefore it must have access to the same address space as the requesting tasks. If all the tasks share the same address space (e.g. single-page table), a single MPW is enough. Otherwise, each task must spawn its own MPW thread. Following the philosophy of HIPPEROS, by default we map the MPW(s) to the *master core*. As a result, if A-MP is used, the overheads due to the execution of the memory phases are localized in the *master core*.

Listing 5 A-MP API.

```
void h_memphase_init_a(void);
h_mp_request_t h_memphase_prefetch_a(void *addr, size_t bytes, u32 part);
h_mp_request_t h_memphase_writeback_a(void *addr, size_t bytes, u32 part);
bool h_memphase_finished_a(h_mp_request_t *request);
void h_cache_set_partition(u32 partition);
void h_cache_revert_partition(void);
```

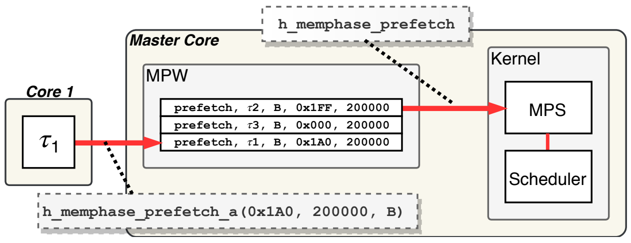
h_memphase_prefetch
Master Core
Kernel
MPW
Core 1
prefetch, τ2, B, 0x1FF, 200000
prefetch, τ3, B, 0x000, 200000
prefetch, τ1, B, 0x1A0, 200000
MPS
τ1
Scheduler
h_memphase_prefetch_a(0x1A0, 200000, B)

Figure 8 Elements involved in A-MP.

Listing 5 shows the functions implemented to use A-MP. Before it can be used, the MPW must be initialized by using the API function `h_memphase_init_a`, after which it remains waiting for memory phase requests. A memory phase request is stored in a data type called `h_mp_request_t` which has 4 elements: type (i.e. prefetch or write-back), data memory address and size, and the cache sub-partition to use (A or B). Tasks can request an A-MP by using functions `h_memphase_prefetch_a` and `h_memphase_writeback_a` for P and W Phases respectively, which return a `h_mp_request_t` data structure. The parameters of these request functions are the memory address region of data to prefetch/writeback (`addr` and `bytes`), and which task cache sub-partition to target (`part`). Tasks can determine if an A-MP has finished by using the function `h_memphase_finished_a`.

Once a request is received by the MPW, it is stored in an internal FIFO queue. The MPW serves these requests by performing S-MP's. Figure 8 illustrates an example of how the A-MP requests operate, in which we can see that the MPW queue is filled with prefetch requests, which are served by requesting an S-MP with function `h_memphase_prefetch`.

It is important to note that the MPW is just another task executing in the system, and as such it has its own cache partition assignment. Accordingly, when it performs a memory phase on behalf of another task, it must temporarily change its own cache partition to the appropriate A or B sub-partition of the requesting task. This way the S-MP performed by the MPW will operate on the correct cache partition. Once the request has been served, the MPW cache partition returns to its default value. We implement two functions to dynamically set the task cache partition: `h_cache_set_partition` to set a specific partition, and `h_cache_revert_partition` to return to the default task partition. Details on how this dynamic cache partition mapping is implemented are detailed in Section 4.4.

It is worth mentioning that since the A-MP are just S-MP executed by the MPW, both S-MP and A-MP can coexist in the same system without further modifications. In our implementation we give the MPW memory phases the lowest priority, so they do not interfere with other tasks S-MP requests. Additionally, the MPW task itself has the highest priority, so any other task mapped in the *master core* would not preempt it and delay the execution of other tasks A-MP's.

To illustrate the benefits of A-MP, Figure 9 shows the same task-set as Figure 7, with the difference that now the tasks use A-MP. In the example, the tasks start by invoking an S-MP to prefetch the initial batch of data to sub-partition A of each task. This request is synchronous because the cache partitions start with a clean state, so there is no benefit in requesting an A-MP prefetch and wait for it to finish.

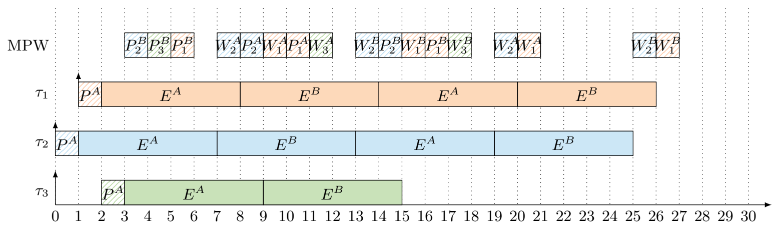
MPW
P_2^B P_3^B P_1^B W_2^A P_2^A W_1^A P_1^A W_3^A W_2^B P_2^B W_1^B P_1^B W_3^B W_2^A W_1^A W_2^B W_1^B
τ_1 P^A E^A E^B E^A E^B
τ_2 P^A E^A E^B E^A E^B
τ_3 P^A E^A E^B
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

Figure 9 Task-set scheduled with asynchronous memory phases (A-MP).

If we now focus on τ_1, we can see that it starts its first execution phase E^A at $t = 2$. At that moment, the task also requests an A-MP prefetch for its sub-partition B (P_1^B), which is served by the MPW at $t = 5$. By the time E^A finished, that prefetch request has already finished so E^B can start without delay ($t = 8$). In general, we can see that while a task is executing an E-Phase on a partition, the MPW is preparing the other sub-partition with the pertinent write-back and prefetch phases, which allows a continuous execution of E-Phases. As a result, the execution time of the tasks is just composed of the execution time of its E-Phases plus an initial synchronous P-Phase. It is worth noting that with A-MP, the phases operate on half the data size compared to S-MP, as they operate on a sub-partition.

In conclusion, A-MP has the potential to *drastically* reduce the execution times of the tasks, at the expense of adding workload in the *master core*. In practice, the *master core* only remains available for non real-time tasks, or real-time tasks with big slack times. Furthermore, it is trivial to see that the benefits of A-MP can only be realized if the E-Phases are long enough to completely cover the execution time-span of the background memory phases.

4.4 Implementation Details

We implemented the Memory Centric Scheduling elements described in Section 4 in version 18.09 of HIPPEROS. While HIPPEROS also supports ARMv8, Intel x86 and PowerPC, we focused our efforts on the widely available ARMv7 architecture. Specifically, we target the NXP i.MX6Q system-on-chip (SoC), composed of 4 Cortex A9 cores, a 16-way 1MB shared L2 cache with the L2C-310 cache controller, and 1 GB of DDR RAM.

Cache partitioning

As described in Section 3, our implementation of MCS requires that each task is guaranteed a dedicated cache partition, which is proposed to avoid intra-core and inter-core cache evictions. Any of the available cache partitioning solutions [24, 19, 6, 11] that provides that guarantee could be used with our implementation. Nonetheless, some caveats must be taken into account, which are described below.

When the cache partitioning can only be achieved at the core level, all the tasks in the same core share the same cache partition, and thus could evict lines of each other. Under this situation, our implementation of MCS is restricted to one task per core, or to multiple non preemptive tasks per core. To implement cache partitioning in this paper, and without loss of generality, we will exploit a feature in the L2C-310 cache controller called *lockdown by master*, which restricts the cache allocations of each core to a particular set of L2 cache-ways, thus effectively achieving core-level cache partitioning. As a consequence, to meet the requirements of our implementation of MCS, we will consider only one task per core. Nevertheless, this limitation does not curtail us from pursuing the objectives of this paper. For the evaluation of our implementation of MCS, we are mainly focused on studying the contention (or lack thereof) in the shared memory. Accordingly, we view the cores as mere producers of memory requests, with no regard to which particular task produced it. For this objective, a configuration of just one task per core is sufficient.

Special consideration must also be taken with systems with a high number of preemptive tasks. Here the cache partitions could get very small, and as a consequence more memory phases would be needed, increasing the overall overheads in the system. These MCS related overheads compound with other pre-existing preemption delays [5]. To mitigate this problem, our implementation of MCS supports non preemptive tasks, which allows core-level cache partitioning and therefore larger partition sizes.

Finally, we assign the HIPPEROS *master core* an L2 cache partition that is big enough to meet the memory requirements of the kernel. This way we can assume that the kernel does not interfere with the tasks in neither the L2 cache nor in shared memory.

Memory phases

We have considered two types of memory phases: prefetch and write-back phases.

The objective of the prefetch phases is to copy lines from the shared memory into the task cache partition. As we have previously stated, in this initial implementation we only target the prefetch of data. For the prefetch we use the ARM `PLD` instruction, which has two main caveats:

1. This instruction copies data to the L1 cache only. This is not a problem because, with *lockdown by master*, any eviction in the L1 cache is allocated into the task L2 cache partition. As a result, all the data prefetched with `PLD` would end up in the L1 cache (private to the core) or in the L2 cache partition.
2. `PLD` is generally defined as a hint instruction, that is, it is not guaranteed that it would produce any effects. However, in our evaluation we have confirmed that, at least in the i.MX6Q SoC we used, this instruction always performs its operation.

As an alternative, ARM defines an optional component in the Cortex-A9 core called the Preload Engine (PLE), which can be used to program loads of selected regions of memory into L2. This component nicely fits the objective of the prefetch phases, but unfortunately is not available in our SoC. It is important to note that we have disabled the speculative hardware prefetchers, which would interfere with our own P-Phases.

Regarding the write-back phases, their objective is two-fold: (1) to copy into the shared memory any data updated during a previous execution phase, and (2) leave the L2 cache partition in a clean state. This can be achieved with common flush cache operations, targeting the L1 private cache and L2 cache partition.

Memory phases worker (MPW)

As described in Section 4.3.2, the Memory Phases Worker (MPW) is a task that performs memory phases on behalf of other tasks. Therefore, it must dynamically change its cache partition to match that of the requesting task. With the *lockdown by master* feature described before, we can perform this partition switch by dynamically changing the core cache ways assignment, adding a necessary L1 flush before the switch to avoid inter-partition pollution.

5 Schedulability Analysis

From an analytical point of view, the main consequence of employing S-MP or A-MP is that the unpredictable interferences in the shared memory and cache are now replaced by predictable scheduling delays. This characteristic can lead to a simplification in the

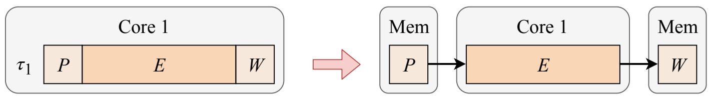

Figure 10 Transformation from PREM task to end-to-end flow (distributed task).

mathematical frameworks needed to determine execution and response time bounds, compared to the fully contended case. Essentially, with S-MP or A-MP, the response time of the tasks is composed of two main components:

1. The execution times of the phases, which are not going to suffer from contention in the shared memory or interferences due to any type of unplanned cache evictions. This removes a great source of uncertainty that usually leads to an over-inflation of the execution time estimations.
2. Scheduling delays: these can come from other tasks in the same core (intra-core), which is the classical scheduling problem [13], or from other cores (inter-core) due to the scheduling of the memory phases, which also have a predictable nature.

If we make the simplifying assumption that other sources of inter-core interferences besides the shared memory and cache are negligible, we can establish that the execution times of the task phases is not affected by the number of tasks in the system. In this situation, with S-MP or A-MP, the response times of any task would only depend on the number of higher priority tasks in its own core, and the number of higher priority memory phases system-wide. In the evaluation section (Section 6) we will see how this assumption mostly holds true in the measurements.

To define a formal response time analysis, and as already hinted by [39], we can draw similarities between PREM and the distributed task model. In distributed systems, the tasks, also sometimes called transactions or end-to-end flows, are formed by a sequence of sub-tasks, each executing in a different processing resource (e.g. processor or network). A sub-task can be a computational task in a processor, or a message scheduled and sent via a network, that could trigger a further sub-task in the recipient processor. Additionally, these sub-tasks are statically mapped to a processing resource due to the high costs that migration would induce in a distributed system. Accordingly each processing resource typically has its own scheduler for the sub-tasks it contains.

In view of this, we can model our PREM tasks as a distributed task: P-Phases and W-Phases are modelled as sub-tasks executing in a *memory* processing resource, and the E-Phases are sub-tasks executing in their original cores. Figure 10 illustrates this transformation with a simple task that uses S-MP, where "Mem" is the *memory* processing resource. For a task that uses A-MP, its equivalent distributed task is composed of just two sub-tasks: one for the initial S-MP needed, and another for the sum of all E-Phases.

The distributed task model has been extensively studied, with several analysis techniques proposed to calculate response times, a number of which are implemented in readily available open-source applications. One of these tools is MAST [25, 14], which implements the seminal holistic analysis [36], and several offset based analyses [26, 27] with different levels of precision and complexity that improves on the holistic analysis, all for sporadic tasks. Section 6 will compare these analytical techniques with actual measured response times. Additionally, compositional analysis techniques can be applied to use different scheduling policies for each processing resource [32][15]. This could enable for example the analysis of Memory Centric Scheduling with EDF memory phases and fixed task priorities execution phases.

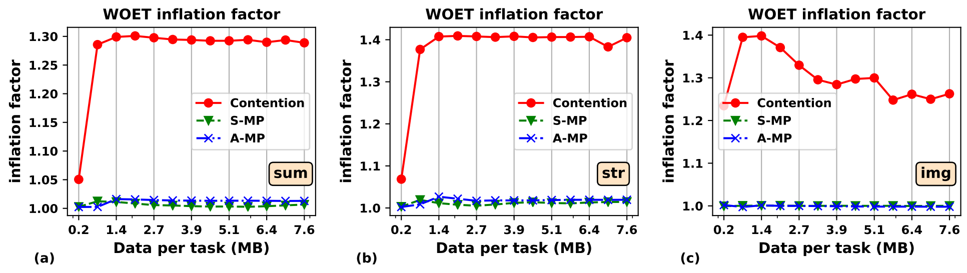

Figure 11 Inflation factors for (a) Sum tasks, (b) Str tasks, and (c) Img tasks.

6 Evaluation

In this section we present the evaluation results of the Memory Centric Scheduling implementation described in Section 5, based on actual execution time measurements of different types of tasks. The framework was implemented in HIPPEROS 18.09, and the hardware platform targeted is a Boundary Devices BD-SL-I.MX6 development board, which includes an NXP i.MX6Q SoC composed of 4 Cortex-A9 cores, 1MB of 16-Way L2 cache with the L2C-310 controller, and 1GB of DDR3 RAM. We label the cores as Core 0 to Core 3, and assign Core 3 as the HIPPEROS *master core*, which means that this core executes the system scheduler, the memory phases scheduler (MPS), and the memory phases worker (MPW) for A-MP.

The main focus is to evaluate how our implementation deals with the contention in the shared memory, but is not concerned about intra-core scheduling. Accordingly, we only map up to one task per core. The index of the task also indicates to which core it is mapped, e.g., τ_0 is mapped to Core 0. Since we only consider one task per core, the concepts of WCET and WCRT as defined in Section 3 became interchangeable during this evaluation.

We wrote three types of tasks: *sum*, which just sum batches of numbers; *str* which encrypts strings; and *img* which applies a Gaussian blur filter on images. We compare 3 scheduling configurations: **A-MP** (from Section 4.3.1), **S-MP** (from Section 4.3.2), and **Contention**. In the latter, the tasks do not invoke memory phases, and as such the hardware arbiters handle the contention in the access to the main memory. For A-MP and S-MP, the task indices also indicate the priority of its memory phases, with τ_0 having the highest priority. We vary the number of tasks in the system, and the task memory requirements from 200 KB to 7.8 MB. Each task is given a partition of 4 cache ways, which implies that for 200KB, the tasks just need 1 `Section`, while for 7.8MB the tasks need 40 `Sections`. To obtain statistically relevant results, a total of 430000 executions were performed.

The evaluation is based on measurements of the execution times of the tasks. We define $\mathrm{woet}_i^K(m)$ as the worst-observed execution time (WOET) of τ_i, for a system with m total tasks, and a K scheduling configuration (C for contention, S for S-MP, and A for A-MP).

Sequential Tasks

We first study the inflation factors of each configuration. For a system with up to m tasks, we define the inflation factor of τ_i as the ratio between its WOET with $(m-1)$ co-runners and its execution time running alone (0 co-runners), that is, $\mathrm{woet}_i(m)/\,\mathrm{woet}_i(1)$. An inflation factor above 1 indicates that the task execution time is affected by contention delays in the shared memory. Figure 11 shows the inflation factors of τ_0, for different data sizes and the three types of tasks (*sum*, *str* and *img*). We can see that as expected, with Contention scheduling, the inflation factor clearly grows above 1, with a measured maximum of up to 1.4. On the other hand, we can observe that with S-MP and A-MP, the inflation factor remains at approximately 1, with a slight advantage to S-MP. It is worth remembering that τ_0 has the highest priority memory phases, which implies that here it does not suffer from any type of scheduling delays. This result also confirms that S-MP and A-MP can indeed isolate the execution times of the tasks with respect the number of tasks accessing the shared memory.

We now study the improvement factor of using A-MP and S-MP with respect to Contention. We define the improvement factor of τ_i as the ratio between its WOET with S-MP or A-MP and its WOET with Contention. Here we consider systems with 3 tasks, that is, the improvement factor is $\text{woet}_i(3)/\,\text{woet}_i^C(3)$. Therefore, an improvement factor below 1 indicates that the task WOET improves with respect to Contention. Figure 12 shows the improvement factor of τ_0, τ_1, τ_2 for different data sizes and focusing on *sum* tasks. With a general view of the figure we can reach two main conclusions. First, with S-MP or A-MP, the WOET improvement gets more apparent the higher the size of the data is, remaining constant above about 2 MB. Second, we can confirm that A-MP yields lower WOET than S-MP. This is expected, as the majority of the memory phases with A-MP do not contribute to the execution time of the task. In more detail, we also see that for τ_0 (highest priority memory phases), its WOET is always better with S-MP or A-MP than with Contention. This result, in addition to the inflation factors shown before, indicate us that with S-MP or A-MP we can achieve at the same time task execution time isolation and a reduction in the execution times. For τ_1, τ_2 the WOET with S-MP is increased with respect Contention for low data sizes (< 0.8 MB). This indicates that in this case, the overheads of S-MP (execution of memory phases and scheduling delays) cannot be compensated by the lower execution times of the fully cached E-Phases. On the other hand, In Figure 12b and c, we can identify that for high data sizes, S-MP also grants improvements in τ_1, τ_2 over Contention.

Until now we have focused on evaluating the WOET's. Another important factor in real-time systems is the variability of the execution times, also called jitter. Figure 13 shows the average observed execution times (AOET) of τ_0, with added error bars (black vertical lines) that show the WOET and best observed execution time (BOET) of each configuration. Additionally, the overlapping patterned bars indicate the portion of the execution time that is contributed by the "operation" portion of the task, which is the whole task in the case of Contention, and the E-Phases in S-MP and A-MP. In the figure we can first confirm that the jitter with Contention is clearly higher, especially when more tasks with more data are involved. This is expected, as in these situations, with more shared memory accesses, there is a higher chance of being delayed due to contention in those accesses, which varies between different executions. On the other hand, S-MP and A-MP provide execution times with no obvious jitter. This is the desired result, and it is expected as the main source of variability (contention in shared memory) is solved by software in a predictable and constant manner. Moreover, the figure also allows us to attest that the execution times of the E-Phases

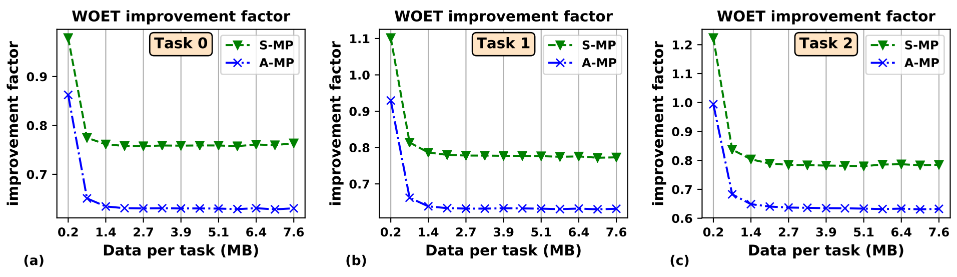

Figure 12 Improvement factors of sum tasks, for (a) Task 0, (b) Task 1 and (c) Task 2.

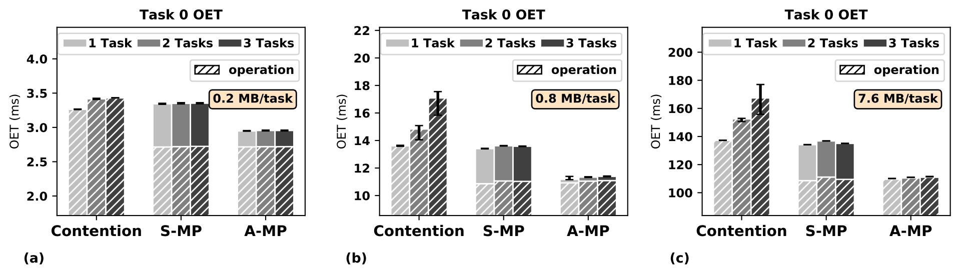

Figure 13 Average OET of Task 0, with maximum and minimum OET as error bars, for (a) 0.2 MB per task, (b) 0.8 MB per task, and (c) 7.6 MB per task.

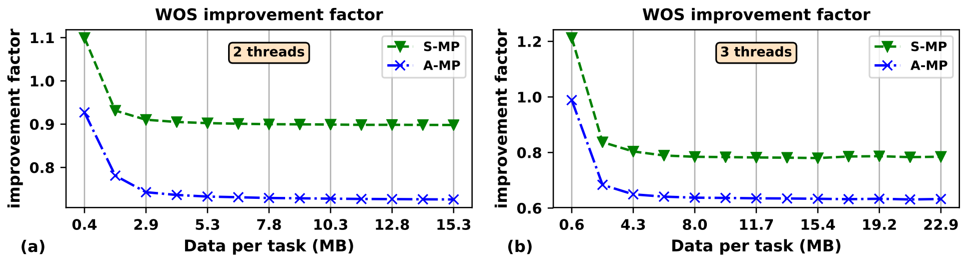

Figure 14 Improvement factors of parallel tasks with (a) 2 threads and (3) threads.

(patterned portion) is the same for S-MP and A-MP. As expected, the only difference between both originates from the higher number of memory phases that are included in the execution times with S-MP.

Multi-threaded Tasks

We now evaluate the benefits of using S-MP and A-MP with multi-threaded tasks. For this, we modify the *sum* tasks used before, so now they spawn 2 to 3 threads to process in parallel a portion of its data. Each parallel thread requests its own memory phases. We define the span of a parallel task as its execution time, which is the time interval between the first thread is spawned, until the last thread finishes. We assume that the work performed outside these parallel threads is negligible. Similarly to the WOET, we define the worst observed span (WOS) as the maximum measured span, denoted as wos^K for a K scheduling configuration. Figure 14 shows the improvement factors of the WOS for S-MP and A-MP over Contention, for parallel tasks with 2 and 3 threads, and different task data sizes. In the figure we see similar results as with sequential tasks before: (1) only S-MP for low data sizes sees and increase in the worst observed span times, and (2) A-MP gets a substantial reduction in the execution times compared to S-MP and Contention.

Schedulability Analysis

We finally compare the measured WOET's with the bounds obtained with analysis techniques originally created for distributed systems. We model the PREM tasks as showed in Section 5, and feed them as input to the MAST tool [14]. The models need worst-case execution times for each task phase. For this, we use the highest task phases measured execution times. Then, we apply three different analysis techniques: HOL, which is the original Holistic analysis by

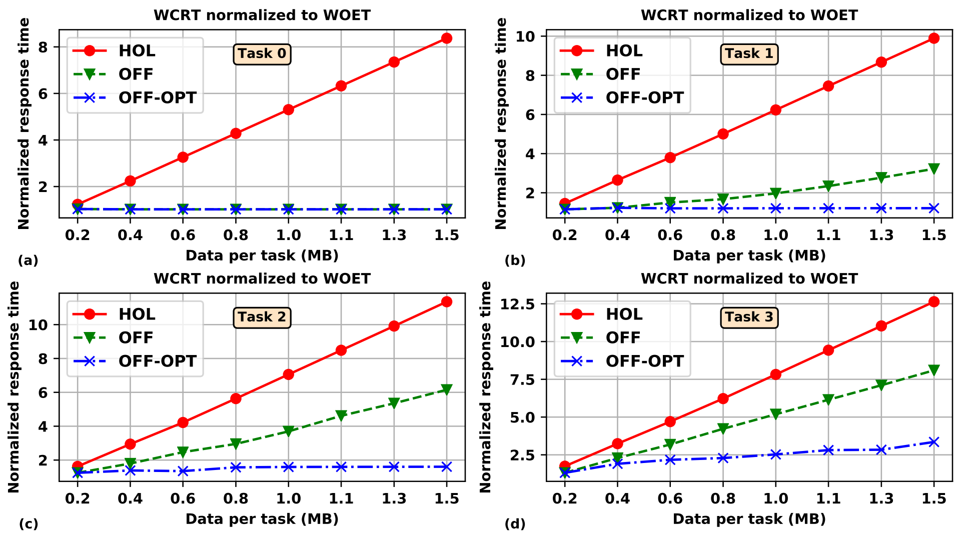

Figure 15 Analytical worst-case response times normalized to WOET, for (a) Task 0, (b) Task 1, (c) Task 2, and (d) Task 3.

Tindell [36]; OFF, which is a subsequent off-based analysis [26]; and OFF-OPT which is an optimization of the original offset-based analysis [27]. We only consider S-MP, as A-MP is modelled as a pure S-MP task with only one memory phase at the beginning (Section 5).

We consider systems with 4 tasks. In Figure 15 we show, for each task in the system, the worst-case response times obtained by these analysis techniques, normalized to the measured WOET's. First, we can confirm that the normalized response times are never below 1. This indicates that the analyses never underestimated the observed response times. Furthermore, we can attest that HOL provides the highest overestimation, followed by OFF and then OFF-OPT. For τ_0 (Figure 15a), which has the highest priority memory phases, both OFF and OFF-OPT recognized that its memory phases always execute without delay, and thus provide worst-case response times estimations very close to the WOET's. For the rest of the tasks, in which memory phases scheduling delays must be accounted for, OFF-OPT provides clearly the best results, with normalized response times clearly below 2 for τ_1, τ_2, and below 4 for τ_3. It is important to note that a higher task data size implies more memory phases. Additionally, a task with low priority memory phases (e.g. τ_3) is impacted by more higher priority memory phases, so the scheduling scenario to analyze increases in complexity.

We think that one of the main sources of overestimation of these analysis techniques may be due to task release phase considerations. The analytical concept of building a worst-case situation (i.e. critical instant) by releasing all the tasks at the same time does not always hold for distributed tasks [27]. Accordingly, part of the challenge of distributed analysis techniques is in finding the tasks release time phases that lead to the worst case, which may not have been arisen during the actual measured executions.

7 Conclusions and Future Work

In this paper we have tackled the problem of contention and interferences in the shared memory of multi-core processors, which is a great impediment for the adoption of this type of processors in real-time applications. Precisely, we have presented and tested an implementation of Memory Centric Scheduling (MCS). The main idea of MCS is to solve the access to the shared memory via a software-based dynamic scheduler, thus avoiding low level and non real-time hardware arbiters.

While previous papers have proposed MCS from a theoretical standpoint, this paper, to the best of our knowledge, is the first time it has been implemented in an actual RTOS supporting dynamic scheduling. The implementation was carried out in an asymmetric multi-core RTOS called HIPPEROS, which locates the scheduler in a dedicated core, called the *master core*.

Two variants of MCS were implemented that can be used in commercial processors: Synchronous Memory Phases (S-MP), in which the tasks execute the memory phases in the foreground, and Asynchronous Memory Phases (A-MP), where the memory phases are executed in the background by a dedicated task. We evaluated the implementation in a quad core commercial processor, and confirmed via measurements the theoretical benefits of Memory Centric Scheduling: the tasks effectively execute free of interferences in the shared memory sub-system.

Specifically, by scheduling the memory phases with fixed priorities, we showed that the tasks execution times were shielded from interferences from lower priority tasks. By extension, the highest priority task has execution times that do not depend on the number of tasks in the multi-core system. The main consequence of these isolation effects is that the execution times can be more easily bounded, compared to the fully contended case. Furthermore, we viewed that with MCS the worst-case observed execution times can be lower compared to the fully contended case, specially for tasks with high memory requirements.

We also applied existing and proven analysis techniques for distributed systems, by exploiting the similarities between the task model used in MCS and the distributed task model. We showed how these techniques can indeed provide safe bounds of the execution times of MCS systems, that are also in many cases very close to the observed values.

For future work we are planning: (1) to extend the evaluation to more than one task per core, analyzing also the benefits of non-preemptibility; (2) to compare with other approaches such as memory bandwidth regulators (e.g. MEMGUARD [42]); (3) to evaluate new scheduling schemes for the memory phases (e.g. LLF) and the MPW.

References

1 Ankit Agrawal, Renato Mancuso, Rodolfo Pellizzoni, and Gerhard Fohler. Analysis of Dynamic Memory Bandwidth Regulation in Multi-core Real-Time Systems. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 230–241. IEEE, December 2018.

2 Ahmed Alhammad and Rodolfo Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *Proceedings of the 14th International Conference on Embedded Software - EMSOFT '14*, pages 1–10, 2014.

3 Ahmed Alhammad and Rodolfo Pellizzoni. Time-predictable execution of multithreaded applications on multicore systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*, pages 1–6, New Jersey, 2014. IEEE Conference Publications.

4 Ahmed Alhammad, Saud Wasly, and Rodolfo Pellizzoni. Memory efficient global scheduling of real-time tasks. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 285–296, 2015.

5 Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, September 2012.

6 Sebastian Altmeyer, Roeland Douma, Will Lunniss, and Robert I. Davis. On the effectiveness of cache partitioning in hard real-time systems. *Real-Time Systems*, 52(5):598–643, September 2016.

7 ARM. Arm® Architecture Reference Manual Supplement Memory System Resource Partitioning and Monitoring (MPAM). URL: `https://developer.arm.com/docs/ddi0598/latest`.

8 Muhammad Ali Awan, Pedro F. Souto, Benny Akesson, Konstantinos Bletsas, and Eduardo Tovar. Uneven memory regulation for scheduling IMA applications on multi-core platforms. *Real-Time Systems*, 55(2):248–292, April 2019.

9 Stanley Bak, Gang Yao, Rodolfo Pellizzoni, and Marco 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. IEEE, August 2012.

10 Moris Behnam, Rafia Inam, Thomas Nolte, and Mikael Sjödin. Multi-core composability in the face of memory-bus contention. *ACM SIGBED Review*, 10(3):35–42, October 2013.

11 Bach D. Bui, Marco Caccamo, Lui Sha, and Joseph Martinez. Impact of Cache Partitioning on Multi-tasking Real Time Embedded Systems. In *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 101–110. IEEE, August 2008.

12 Felipe Cerqueira, Manohar Vanga, and Björn B. Brandenburg. Scaling global scheduling with message passing. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 263–274, April 2014.

13 Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):1–44, October 2011.

14 Michael González Harbour, Jose Javier Gutiérrez, José M. Drake, Patricia López Martínez, and Jose Carlos Palencia. Modeling distributed real-time systems with MAST 2. *Journal of Systems Architecture*, 59(6):331–340, June 2013.

15 Arne Hamann, Marek Jersak, Kai Richter, and Rolf Ernst. Design space exploration and system optimization with symTA/S - Symbolic timing analysis for systems. *Proceedings - Real-Time Systems Symposium*, pages 469–478, 2004.

16 Tobias Kalb, Lester Kalms, Diana Göhringer, Carlota Pons, Ananya Muddukrishna, Magnus Jahre, Boitumelo Ruf, Tobias Schuchert, Igor Tchouchenkov, Carl Ehrenstråhle, Magnus Peterson, Flemming Christensen, Antonio Paolillo, Ben Rodriguez, and Philippe Millet. *Developing Low-Power Image Processing Applications with the TULIPP Reference Platform Instance*, pages 181–197. Springer International Publishing, Cham, 2019.

17 Claire Maiza, Hamza Rihani, Juan M Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I Davis. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Computing Surveys*, 52(3), July 2019.

18 R. Mancuso, R. Dudko, and M. Caccamo. Light-PREM: Automated software refactoring for predictable execution on COTS embedded systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10, August 2014.

19 Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo 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. IEEE, April 2013.

20 Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, and Heechul Yun. WCET (m) estimation in multi-core systems using single core equivalence. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pages 174–183. IEEE, 2015.

21 Renato Mancuso, Rodolfo Pellizzoni, Neriman Tokcan, and Marco Caccamo. WCET Derivation Under Single Core Equivalence With Explicit Memory Budget Assignment. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

22 Joel Matějka, Björn Forsberg, Michal Sojka, Zdeněk Hanzálek, Luca Benini, and Andrea Marongiu. Combining PREM Compilation and ILP Scheduling for High-performance and Predictable MPSoC Execution. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'18, pages 11–20, New York, NY, USA, 2018. ACM.

23 Sparsh Mittal. A Survey of Techniques for Cache Partitioning in Multicore Processors. *ACM Computing Surveys*, 50(2):1–39, May 2017.

24 Frank Mueller. Compiler Support for Software-based Cache Partitioning. In *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, &Amp; Tools for Real-time Systems*, LCTES '95, pages 125–133, New York, NY, USA, 1995. ACM.

25 University of Cantabria. MAST. URL: `https://mast.unican.es/`.

26 Jose Carlos Palencia and Michael Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 26–37. IEEE Comput. Soc, 1998.

27 Jose.C. Palencia and Michael Gonzalez Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054)*, pages 328–339. IEEE Comput. Soc, 1999.

28 Antonio Paolillo, Olivier Desenfans, Vladimir Svoboda, Joël Goossens, and Ben Rodriguez. A New Configurable and Parallel Embedded Real-time Micro-Kernel for Multi-core platforms. *OSPERT 2015*, pages 25–27, 2015.

29 Antonio Paolillo, Paul Rodriguez, Vladimir Svoboda, Olivier Desenfans, Joël Goossens, Ben Rodriguez, Sylvain Girbal, Madeleine Faugère, and Philippe Bonnot. Porting a safety-critical industrial application on a mixed-criticality enabled real-time operating system. In *Proceedings of the 5th Workshop on Mixed-Criticality Systems*, December 2017.

30 Antonio Paolillo, Paul Rodriguez, Nikita Veshchikov, Joël Goossens, and Ben Rodriguez. Quantifying Energy Consumption for Practical Fork-Join Parallelism on an Embedded Real-Time Operating System. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, pages 329–338. ACM, 2016.

31 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279. IEEE, April 2011.

32 Juan M. Rivas, Jose Javier Gutiérrez, Jose Carlos Palencia, and Michael González Harbour. Schedulability analysis and optimization of heterogeneous EDF and FP distributed real-time systems. *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 195–204, 2011.

33 HIPPEROS SA. The real-time OS for high performance embedded systems. `https://www.hipperos.com/maestro/`. 2019-02-04.

34 Ahmad Sadek, Ananya Muddukrishna, Lester Kalms, Asbjørn Djupdal, Ariel Podlubne, Antonio Paolillo, Diana Goehringer, and Magnus Jahre. Supporting Utilities for Heterogeneous Embedded Image Processing Platforms (STHEM): An Overview. In Nikolaos Voros, Michael Huebner, Georgios Keramidas, Diana Goehringer, Christos Antonopoulos, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, pages 737–749, Cham, 2018. Springer International Publishing.

35 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. `doi:10.1109/RTAS.2016.7461321`.

36 Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2-3):117–134, April 1994.

37 Saud Wasly and Rodolfo Pellizzoni. Hiding Memory Latency Using Fixed Priority Scheduling. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 75–86, 2014.

38 Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48(6):681–715, November 2012.

39 Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Heechul Yun, and Marco Caccamo. Global Real-Time Memory-Centric Scheduling for Multicore Systems. *IEEE Transactions on Computers*, 65(9):2739–2751, 2016.

40 Gang Yao, Heechul Yun, Zheng Pei Wu, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Schedulability Analysis for Memory Bandwidth Regulated Multicore Real-Time Systems. *IEEE Transactions on Computers*, 65(2):601–614, February 2016.

41 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166. IEEE, April 2014.

42 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.