CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

Analyzing Fixed Task Priority Based Memory Centric Scheduler for the 3-Phase Task Model

Jatin Arora
Syed Aftab Rashid
Cláudio Maia
Eduardo Tovar

CISTER-TR-220608

2022/08/23

Analyzing Fixed Task Priority Based Memory Centric Scheduler for the 3-Phase Task Model

Jatin Arora, Syed Aftab Rashid, Cláudio Maia, Eduardo Tovar

CISTER Research Centre

Polytechnic Institute of Porto (ISEP P.Porto)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: jatin@isep.ipp.pt, syara@isep.ipp.pt, clrrm@isep.ipp.pt, emt@isep.ipp.pt

https://www.cister-labs.pt

Abstract

The sharing of main memory among concurrently executing tasks on a multicore platform results in increasing the execution times of those tasks in a non-deterministic manner. The use of phased execution models that divide the execution of tasks into distinct memory and execution phase(s), e.g., the PRedictable Execution Model (PREM) and the 3-Phase task model, along with Memory Centric Scheduling (MCS) present a promising solution to reduce main memory interference among tasks.

Existing works in the state-of-the-art that focus on MCS have considered (i) a TDMA based memory scheduler, i.e., tasks' memory requests are served under a static TDMA schedule, and (ii) Processor-Priority (PP) based memory scheduler, i.e., tasks' memory requests are served depending on the priority of the processor/core on which the task is executing. This paper extends MCS by considering a Task-Priority (TP) based memory scheduler, i.e., tasks' memory requests are served under a global priority order depending on the priority of the task that issues the requests. We present an analysis to bound the total memory interference that can be suffered by the tasks under the TP-based MCS. In contrast to most existing works on MCS that consider non-preemptive tasks, our analysis considers limited preemptive scheduling. Additionally, we investigate the impact of different preemption points on the memory interference of tasks. Experimental results show that our proposed TP-based MCS can significantly reduce memory interference that can be suffered by the tasks in comparison to the PP-based MCS approach.

Analyzing Fixed Task Priority Based Memory Centric Scheduler for the 3-Phase Task Model

Jatin Arora[†]*, Syed Aftab Rashid[†‡], Cláudio Maia[†], Eduardo Tovar[†]
[†]CISTER Research Centre, ISEP, Porto, Portugal [‡]VORTEX CoLab, Porto, Portugal

***Abstract*—The sharing of main memory among concurrently executing tasks on a multicore platform results in increasing the execution times of those tasks in a non-deterministic manner. The use of phased execution models that divide the execution of tasks into distinct execution and memory phase(s), e.g., the PRedictable Execution Model (PREM) and the 3-Phase task model, along with Memory Centric Scheduling (MCS) present a promising solution to reduce main memory interference among tasks.**

Existing works in the state-of-the-art that focus on MCS have considered (i) a TDMA-based memory scheduler, i.e., tasks' memory requests are served under a static TDMA schedule, and (ii) Processor-Priority (PP) based memory scheduler, i.e., tasks' memory requests are served depending on the priority of the processor/core on which the task is executing. This paper extends MCS by considering a Task-Priority (TP) based memory scheduler, i.e., tasks' memory requests are served under a global priority order depending on the priority of the task that issues the requests. We present an analysis to bound the total memory interference that can be suffered by the tasks under the TP-based MCS. In contrast to the recent works on MCS that considers non-preemptive tasks, our analysis considers limited preemptive scheduling. Additionally, we investigate the impact of different preemption points on the memory interference of tasks. Experimental results show that our proposed TP-based MCS can significantly reduce the memory interference that can be suffered by the tasks in comparison to the PP-based MCS.

I. Introduction

In commercial-off-the-shelf (COTS) multicore processors, different hardware resources are shared among the processing cores such as last-level caches, interconnects/memory buses, main memory, etc. Consequently, tasks executing on a given core can suffer inter-core interference while trying to access any of these shared resources. Several works in the literature [2], [3], [6], [10], [13], [14] have shown that the *main memory interference* is one of the main sources of interference in COTS multicore platforms that can significantly impact the execution times of tasks. As a result, a plethora of works have focused on solving the memory interference problem [2], [3], [6], [10], [13]–[15], [18], [19].

It has been shown that the use of *phased execution models*, e.g., the PRedictable Execution Model (PREM) [11], or its generalization, the 3-phase task model [4], [9], with *Memory Centric Scheduling* (MCS) [15], [18]–[20] provide an efficient solution to the memory interference problem in particular. Phased execution models, such as PREM or the 3-phase task model divide the execution of tasks into execution and memory phase(s) to ensure that tasks only access the main memory during their memory phase(s), and computation is only performed during the execution phase, without the need of accessing the main memory. Then, at the system level, a memory centric scheduler can be used to serialize the accesses to the main memory to reduce inter-task memory interference.

Several implementations of MCS have been proposed in the literature. Works like [10] generate a system-level offline schedule of tasks such that no two tasks can access the main memory at the same time, thereby eliminating memory interference. However, enforcing such an offline schedule may not be possible in scenarios where tasks are event/time triggered. Other works have adopted time-division multiple-access (TDMA) to implement MCS [18], [19]. However, due to its non-work-conserving nature, a TDMA-based MCS can overestimate the memory interference of tasks. In a recent work, Schwäricke et al. [15] have presented an analysis that implements MCS using Processor-Priority (PP)-based memory scheduler. In PP-based MCS, memory requests (or phases) of tasks are served depending on the priority of the processor/core on which the tasks are executing. A two-level priority mechanism is used where at the core level, tasks are scheduled using partitioned fixed-priority non-preemptive scheduling and the memory arbiter employs a global fixed processor priority based scheduling to schedule memory phases of tasks.

While the PP-based MCS approach can outperform the TDMA-based MCS, it still has limitations. For example, under the two-level priority mechanism used by the PP-based MCS, a task τ_i with the highest local priority on a core π_l may still suffer memory interference from other tasks that are executing on processors/cores with higher global priorities than π_l. This can have a significant impact on the schedulability of τ_i and consequently on the schedulability of the system.

State-of-the-art [3], [12] has shown that fixed Task-Priority (TP)-based memory arbitration schemes provide much tighter bounds on the main memory interference for the generic task model. This provides strong motivation to use a TP-based MCS to schedule PREM/3-phase tasks. Thus, in this work, we propose a Task-Priority (TP) based implementation of MCS, i.e., tasks' memory requests (or phases) are served under a global priority order depending on the priority of the task that issues the requests. This leads to a significant reduction in the memory interference of tasks.

The main **contributions** of this work are the following:

1) We present an analysis to bound the total memory interference that can be suffered by the tasks under a TP-based MCS approach. In contrast to most existing works on MCS

*Corresponding author

that consider non-preemptive tasks, our approach considers limited preemptive scheduling at the core level;
2) Existing implementations of MCS that allow task preemptions, e.g., [19], usually assume fully preemptive computation phases of tasks. We investigate the impact of different preemption points on the memory interference suffered by tasks and show that this assumption may not always lead to a tighter bound on the memory interference of tasks; and
3) We compare the performance of our proposed TP-based MCS approach to the PP-based MCS approach [15] under different settings. Experimental results show that our proposed approach can provide significantly tighter bounds on the memory interference of tasks, which can lead to an improvement in the task set schedulability by up to 91 percentage points.

Paper Organization: The rest of the paper is organized as follows: Section II describes the system and execution models. Motivational example is presented in Section III. Section IV discusses the proposed TP-based MCS analysis. The WCRT analysis for the proposed TP-based MCS is presented in Section V. The impact of different preemption point selection on the memory interference of tasks is discussed in Section VI. Experimental results are detailed in Section VII, followed by related work in Section VIII and conclusion in Section IX.

II. System Model

We assume a multicore system comprising m identical cores $(\pi_1, \pi_2, \ldots, \pi_m)$ that access the *main memory* (e.g. DRAM) through a single memory arbiter that can handle only one memory request at a time. As in [19], we also assume that the local memory (i.e., cache/scratchpad) of each core can be partitioned among all the tasks running on that core such that each task has its own non-overlapping partition which is sufficiently large to store all its code/data. If this is not possible due to the limited size of the local memory, tasks can be divided into multiple segments using existing framework [16] so that the code/data required by any segment of a task can be stored in its own partition.

A. Task Model

We consider a task set Γ comprising n sporadic tasks from which a subset Γ' is assigned to each core at the design time according to any given task-to-core mapping strategy. Each task τ_i is characterized by C_i, that is the Worst-Case Execution Time (WCET) of τ_i *measured in isolation*, T_i, that is the minimum inter-arrival time between any two consecutive jobs of τ_i, and D_i, that is the relative deadline of τ_i. We assume $D_i \leq T_i$. Tasks assigned to a core at design time are not allowed to migrate during run-time. Task priorities are assigned at design-time using a fixed-task priority algorithm such as rate/deadline monotonic [8], ensuring that the index of each task is unique, which provides a global priority order. The global priority of each task translates into a local priority order on each core which is used for scheduling purposes.

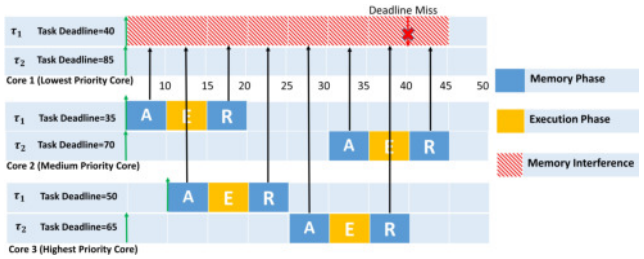
In this work, we consider a 3-phase task model which have been studied by the academia and industry [2], [4], [9], [10], [17]. In the 3-phase task model, the execution of a task is divided into *Acquisition* (A), *Execution* (E), and *Restitution* (R) phases. When a task is released, it first executes the A-phase by fetching all its data/instructions from the main memory to the core's local memory. It then executes its E-phase using the data/instructions already available in the core's local memory without requiring access to the main memory. Finally, in the R-phase, the task writes-back the modified data to the main memory and completes its execution. This execution behavior categorizes the A- and R-phase into memory phases, i.e., time intervals in which accesses to the main memory are allowed/performed, and the E-phase into a computation phase, i.e., no memory request can be issued in this phase. The WCET of A, E and R-phases of task τ_i is given by C_i^A, C_i^E, and C_i^R, respectively, which sums up to the total WCET of task τ_i, i.e., $C_i = C_i^A + C_i^E + C_i^R$.

We assume fixed-priority limited preemptive scheduling where a lower priority task can be preempted *any time during the execution of its E-phase* by a higher priority task released on the same core. This assumption is in line with existing works, e.g., [19]. Memory phases are assumed to be non-preemptive and only one phase can execute at a time on a given core. Each task releases potentially infinite number of jobs where each job instance is denoted by k. The response time of the k^{th} job of task τ_i is denoted by $R_{i,k}$. The Worst-Case Response Time (WCRT) of task τ_i, i.e., the largest response time of any job of τ_i, is denoted by R_i^{max}.

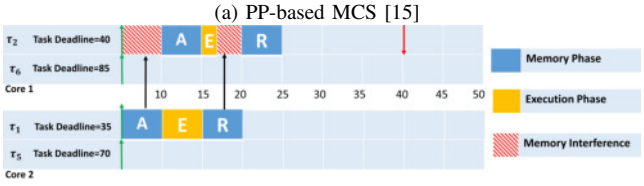
For notational convenience, we use: $hp_{i,l}$, $hep_{i,l}$ and $lp_{i,l}$ to denote the set of tasks assigned to the same core π_l as τ_i with priorities higher, higher or equal, and lower than that of τ_i, respectively. The core on which the task under analysis, i.e., τ_i, executes is termed as the *local core*. Similarly, $hp_{i,r}$ and $lp_{i,r}$ denotes the set of tasks assigned to a core π_r (i.e., $\pi_r \neq \pi_l$) with priorities higher and lower than that of τ_i, respectively. All cores in the platform other than the local core are termed as the *remote cores*.

B. Task Priority (TP) based Memory Centric Scheduler

We assume that a Task Priority (TP) based memory centric scheduler is used to control tasks' accesses to the main memory. Under the TP-based memory centric scheduler, tasks' memory requests/phases are served in a global priority order. Each core maintains a *memory buffer* which stores at most one memory phase that is ready to execute. The state of this memory buffer can be updated by the core as per the tasks released on that core. The core's memory buffer can be empty if there is no active task or the core is executing an E-phase. If the memory buffer of at least one core is non-empty, the TP-based memory scheduler schedules a memory phase of a task that has the highest global priority among all ready tasks. Once a memory phase starts executing, the TP-based memory scheduler does not schedule any other memory phase to ensure the non-preemptive execution of the memory phase. Once the ongoing memory phase completes its execution, the TP-based memory scheduler checks the memory buffers of all the cores and schedules the memory phase of a task that has the highest global priority among all ready tasks.



(a) PP-based MCS [15]



(b) TP-based MCS

Fig. 1: Inter-Core Memory Interference

III. Motivational Example

The first implementation of Memory Centric Scheduling (MCS) [19] considers TDMA-based static slots to schedule the memory accesses of tasks. The TDMA-based MCS allows the memory phases to preempt the execution phases at the core level to efficiently utilize the available TDMA slots. This can potentially improve the response time of tasks. However, the TDMA-based MCS is built on top of conventional TDMA which is a non-work-conserving arbitration policy and thus may overestimate memory interference of tasks. Schwäricke et al. [15] improved the TDMA-based MCS by considering Processor Priority (PP)-based memory scheduling. Their work considers a two-level scheduling approach: 1) fixed-priority non-preemptive scheduling to schedule tasks at the core level; 2) fixed processor priority to schedule the memory requests (phases) at the system level. Due to the two-level scheduling used by the PP-based MCS, tasks with higher local priorities that execute on lower global priority cores can suffer high memory interference, i.e., from all tasks that execute on all higher priority cores. This can potentially result in deadline misses. See Figure 1a, for an example scenario that shows 6 tasks are scheduled on 3 cores such that two tasks execute on each core. Task priorities are assigned at the core level using deadline monotonic, i.e., shorter the deadline, higher the priority, and each core has a unique global priority to access the main memory. We can see in Figure 1a, that task τ_1 that executes on core 1 is the highest priority task on that core. However, since core 1 has the lowest global priority among all the cores, τ_1 on core 1 can suffer memory interference from all tasks executing on other higher priority cores (disregard of their local priorities). Consequently, this memory interference may lead to a deadline miss for task τ_1 on core 1.

It has been proven in the literature [3], [12] that fixed task priority-based memory scheduling can perform significantly better than fixed processor priority or TDMA-based scheduling for the generic task model (see Figure 5 of [3]). This provides a strong motivation to implement memory centric scheduler using a Task Priority (TP) based scheduling approach. In TP-based MCS, task priorities are assigned in a global priority order to schedule the main memory accesses. This global priority order translates into a local priority at the core, which is used to schedule the tasks at the core level. Consequently, by doing so, TP-based MCS can improve the response time of all higher priority tasks, e.g., tasks with shorter deadlines/periods, at the system level. To illustrate, consider the same example scenario shown in Figure 1a applied to the TP-based MCS. The resulting schedule of tasks is shown in Figure 1b. Since the TP-based MCS assigns a global priority order to tasks, the task τ_1 executing on core 1 in Figure 1a, will be assigned a global priority of 2 according to the TP-based MCS. Effectively, task τ_1 executing on core 1 in Figure 1a is labeled as task τ_2 in Figure 1b. Consequently, we can see in Figure 1b that due to the global priority ordering used by TP-based MCS, task τ_2 will only suffer memory interference from one higher priority task, i.e., task τ_1 on core 2. This confirms that, under TP-based MCS, task τ_2 will suffer significantly lower memory interference in comparison to the PP-based MCS.

IV. Analyzing Fixed Task Priority Based Memory Centric Scheduler

When combining phased task models, e.g., PREM or the 3-phase task model, with a memory centric scheduler, the goal is to eliminate/minimize main memory interference suffered by the tasks. However, depending on the scheduling algorithm and the behavior of the memory scheduler, tasks may still be subjected to different types of execution delays. Under TP-based MCS, each task in the system is assigned a global priority using a fixed-priority scheduling scheme, e.g., Rate/Deadline monotonic. Effectively, any task τ_i executing on a core π_l will be served in a global priority order depending on its priority. Formally, under TP-based MCS, task τ_i can suffer four types of delays due to the tasks running on the local core and on remote cores, namely,

1) **Intra-core Interference:** The maximum interference that can be suffered by task τ_i due to all *higher priority tasks* released on the *local core* π_l.
2) **Intra-core Blocking:** The maximum blocking that can be suffered by task τ_i due to *lower priority tasks* that execute on the *local core* π_l.
3) **Inter-core Memory Interference:** The maximum memory interference that can be suffered by task τ_i due to all *higher priority tasks* executing on *all the remote cores*.
4) **Inter-core Memory Blocking**[1]**:** The maximum memory blocking that can be suffered by task τ_i due to all *lower priority tasks* executing on *all the remote cores*.

In fixed-priority limited preemptive scheduling, the WCRT of task τ_i is observed during the longest level-i busy window [1].

Definition IV.1. [Level-i busy window (from [7])] A level-i busy window is a time interval (a, b) in which the pending workload of tasks with priorities higher or equal to that of

[1]Note that PP-based MCS [15] use global memory preemptions to avoid inter-core memory blocking. However, global memory preemptions in TP-based MCS can lead to unbounded priority inversion (see Figure 3 of [15]).

task τ_i is positive for all $t \in (a, b)$ and 0 at the boundaries a and b.

Let $W_{i,l}$ denote the length of the longest level-i busy window for a task τ_i executing on the local core π_l. The value of $W_{i,l}$ can only be obtained by first bounding the following terms.

A. Bounding Intra-Core Interference

The maximum intra-core interference that can be caused by all tasks in $hp_{i,l}$ during the level-i busy window $W_{i,l}$ depend on the maximum number of jobs released by all the tasks in $hp_{i,l}$ during $W_{i,l}$. Therefore, to upper bound intra-core interference, we use the upper event arrival function $\eta_h^+(\Delta)$ that captures the maximum number of jobs released by a task τ_h in any time interval of length Δ [13]. Consequently, the maximum intra-core interference that can be caused by all tasks in $hp_{i,l}$ during $W_{i,l}$ is given by

$$I_i(W_{i,l}) = \sum_{\tau_h \in hp_{i,l}} (\eta_h^+(W_{i,l}) \times C_h) \quad (1)$$

Equation 1 considers the WCET of all jobs released by all higher priority tasks on the local core, i.e., $\forall \tau_h \in hp_{i,l}$, during any time interval of length $W_{i,l}$.

B. Bounding Intra-Core Blocking

As explained in the system model, we assume limited preemptive scheduling where tasks can be preempted during the execution of their E-phases. Considering this, a given task τ_i can only suffer intra-core blocking due to only one memory phase of a lower priority task that starts executing before the arrival of τ_i because τ_i can preempt the lower priority task once it starts executing its E-phase. Therefore, the maximum intra-core blocking that can be suffered by task τ_i is given by the length of the largest memory phase (i.e., either A- or R-phase) among all the tasks in $lp_{i,l}$. The upper bound on the intra-core blocking of τ_i is denoted B_i and can be computed as follows:

$$B_i = \max(\max_{\forall \tau_j \in lp_{i,l}} \{C_j^A\}, \max_{\forall \tau_j \in lp_{i,l}} \{C_j^R\}) \quad (2)$$

C. Bounding Inter-Core Memory Interference

Under the TP-based MCS, the memory phases of a task τ_i can only be served after the completion of all the memory phases of all tasks having higher priority than τ_i. The contribution of tasks with higher priority than τ_i, executing on the local core π_l, is already accounted for in the intra-core interference $I_i(W_{i,l})$. Therefore, the maximum inter-core memory interference caused by all higher priority tasks running on all the remote cores will be computed using the following lemma.

Lemma 1. *The maximum inter-core memory interference that can be suffered by tasks executing on the local core π_l due to higher priority tasks running on all the remote cores during $W_{i,l}$ is upper-bounded by $I_i^{Mem}(W_{i,l})$, where*

$$I_i^{Mem}(W_{i,l}) = \sum_{r=1, r \neq l}^{m} \sum_{\tau_u \in hp_{i,r}} \eta_u^+(W_{i,l}) \times (C_u^A + C_u^R) \quad (3)$$

Proof. Under the TP-based MCS, memory phases of tasks are served in a global priority order. Thus, a task τ_i executing on a core π_l can suffer inter-core memory interference from all tasks executing on all the remote cores that have a higher priority than τ_i. A task τ_u released on a remote core π_r with priority higher than that of τ_i, i.e., $\tau_u \in hp_{i,r}$, can only cause inter-core memory interference on τ_i when it executes its memory phases. So, the maximum inter-core memory interference that one job of $\tau_u \in hp_{i,r}$ can cause is given by the sum of the WCET of its A- and R-phases, i.e., $C_u^A + C_u^R$. Furthermore, from the upper event arrival function, the maximum number of jobs released by task τ_u during any time interval of length $W_{i,l}$ is upper bounded by $\eta_u^+(W_{i,l})$. Hence, the maximum memory interference that can be caused by a task $\tau_u \in hp_{i,r}$ during $W_{i,l}$ is upper bounded by $\eta_u^+(W_{i,l}) \times (C_u^A + C_u^R)$. Considering that all higher priority tasks released on core π_r during $W_{i,l}$ can contribute to the inter-core memory interference, the maximum inter-core memory interference that can be caused by all tasks executing on core π_r is given by $\sum_{\tau_u \in hp_{i,r}} \eta_u^+(W_{i,l}) \times (C_u^A + C_u^R)$. Extending this result to all remote cores, the maximum inter-core memory interference that can be suffered by tasks executing on the local core during $W_{i,l}$ is upper bounded by Equation 3. □

D. Bounding Inter-Core Memory Blocking

Due to non-preemptive memory phases, a task τ_i can suffer inter-core memory blocking if a lower priority task on a remote core starts executing its memory phase before the release of a memory phase of task τ_i. This behavior is observed for all tasks that execute on the local core π_l during $W_{i,l}$. We use the following steps to compute the inter-core memory blocking.

- Bounding the *maximum number* of inter-core memory blockings that can be *suffered*.
- Bounding the *maximum number* of inter-core memory blockings that can be *caused*.
- Upper bounding the *maximum inter-core memory blocking* during $W_{i,l}$.

Next, we explain how each of these steps will be performed.

1) ***Bounding the maximum number of inter-core memory blockings that can be suffered****:* In this step, we will explain how to upper bound the maximum number of inter-core memory blockings that can be suffered by tasks executing on the local core during $W_{i,l}$. Firstly, we present the following example to illustrate the computation of this step. We then use Lemma 2 for formal computation.

Example 1: *Figure 2 shows an example schedule where 3 tasks are executing on the local core and task τ_3 is the task under analysis. Global priorities are assigned to tasks and are indexed according to their priorities, i.e., τ_1, τ_2, τ_3. We can see in Figure 2, each time a memory phase executes after an E-phase on the local core π_l, it may suffer inter-core memory blocking due to the execution of a memory phase (i.e., A or R-phase) of a lower priority task running on a remote core π_r. Furthermore, due to preemptive E-phases of tasks, each higher priority task can preempt a lower priority task during its E-phase in the worst-case scenario. So, when an E-phase*

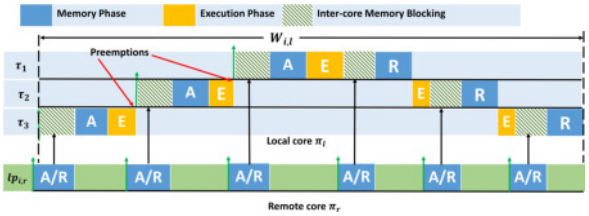


Fig. 2: Maximum number of inter-core memory blockings that can be suffered on the local core π_l during $W_{i,l}$

is executed on core π_l, a lower priority task on a remote core can start executing its A/R-phase, causing inter-core memory blocking. Therefore, we see in Figure 2 that the inter-core memory blocking is suffered by each memory phase of tasks τ_1, τ_2, and τ_3 that executes after an E-phase on the local core.

Lemma 2. *The maximum number of times that tasks executing on the local core π_l can suffer inter-core memory blocking during $W_{i,l}$ is upper-bounded by $\Phi_i(W_{i,l})$, where*

$$\Phi_i(W_{i,l}) = \sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) \times 2 \tag{4}$$

Proof. It is only during the execution of E-phases that the local core can not schedule any memory phases during the level-i busy window. Consequently, in the worst-case, the local core can suffer an inter-core memory blocking from a lower priority task executing on a remote core for every memory phase that execute on the local core after an E-phase. For example, if the local core is executing an E-phase at time instant t, the memory scheduler is allowed to schedule a memory phase of a lower priority task τ_l' executing on a remote core. Now, when the local core completes the execution of its E-phase and wants to execute a memory phase at time instant $t+\epsilon$, it may suffer inter-core memory blocking as τ_l' is already executing a memory phase. This implies that the maximum number of memory blockings that the local core can suffer depends on the number of times E-phases are executed on the local core during $W_{i,l}$. However, considering that in our model the E-phases are preemptive, a task can be preempted several times during its E-phase and each preemption may lead to an inter-core memory blocking. Consequently, the local core can suffer several memory blockings during the execution of an E-phase. Although we cannot predict how many times an E-phase is preempted during $W_{i,l}$, we know that in the worst-case each memory phase that executes during $W_{i,l}$ can suffer inter-core memory blocking. Therefore, knowing that $\eta_i^+(W_{i,l})$ upper bounds the number of jobs that can be released by task τ_i during $W_{i,l}$, $\eta_i^+(W_{i,l}) \times 2$ upper bounds the number of times τ_i can suffer inter-core memory blocking during $W_{i,l}$. Similarly, $\sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) \times 2$ upper bounds the maximum number of inter-core memory blockings that can be suffered by all tasks executing on core π_l during $W_{i,l}$. □

*2) **Bounding the maximum number of inter-core memory blockings that can be caused**:* The maximum number of inter-core memory blockings that can be caused by lower priority tasks running on all remote cores during $W_{i,l}$ are computed using the following lemma.

Lemma 3. *The maximum number of times that lower priority tasks running on all remote cores can cause inter-core memory blocking during $W_{i,l}$ is upper-bounded by $\mu_i(W_{i,l})$, where*

$$\mu_i(W_{i,l}) = \sum_{r=1, r\neq l}^{m} \sum_{\tau_q \in lp_{i,r}} \eta_q^+(W_{i,l}) \times 2 \tag{5}$$

Proof. For a task τ_q running on a remote core π_r such that $\tau_q \in lp_{i,r}$, the maximum number of jobs that can be released by τ_q during $W_{i,l}$ is upper bounded by $\eta_q^+(W_{i,l})$. As memory phases are non-preemptive, each inter-core memory blocking caused by a lower priority task can be of at most one memory phase. Consequently, the maximum number of inter-core memory blockings that can be caused by one job of task τ_q during $W_{i,l}$ is 2 (i.e., by its A- and R-phases) and the maximum number of inter-core memory blockings that can be caused by all jobs of τ_q that execute during $W_{i,l}$ is upper bounded by $\eta_q^+(W_{i,l}) \times 2$. Similarly, the maximum number of inter-core memory blockings that can be caused by all lower priority tasks released on a remote core π_r, i.e., $lp_{i,r}$, during $W_{i,l}$ is upper bounded by $\sum_{\tau_q \in lp_{i,r}} \eta_q^+(W_{i,l}) \times 2$. Extending this to all remote cores, the Lemma follows. □

*3) **Upper bounding the maximum inter-core memory blocking**:* Having bounded the values of $\Phi_i(W_{i,l})$ and $\mu_i(W_{i,l})$, we will now compute an upper bound on the maximum inter-core memory blocking that can be suffered by tasks executing on the local core during $W_{i,l}$. To do so, we consider the following cases:

Case 1: $\Phi_i(W_{i,l}) \geq \mu_i(W_{i,l})$, the maximum number of inter-core memory blockings that can be suffered by tasks executing on core π_l is *greater than or equal* to the maximum number of inter-core memory blockings that can be caused by all lower priority tasks running on all remote cores during $W_{i,l}$.

Case 2: $\Phi_i(W_{i,l}) < \mu_i(W_{i,l})$, the maximum number of inter-core memory blockings that can be suffered by tasks executing on core π_l is *less* than the maximum number of inter-core memory blockings that can be caused by all lower priority tasks running on all remote cores during $W_{i,l}$.

Maximum Inter-Core Memory Blocking for Case 1: Under Case 1, the maximum inter-core memory blocking will be computed using the following lemma.

Lemma 4. *If $\Phi_i(W_{i,l}) \geq \mu_i(W_{i,l})$, then the maximum inter-core memory blocking that can be suffered by tasks executing on the local core π_l due to lower priority tasks running on all remote cores during any time interval of length $W_{i,l}$ is upper-bounded by $B_i^{Mem}(W_{i,l})$, where*

$$B_i^{Mem}(W_{i,l}) = \sum_{r=1, r\neq l}^{m} \sum_{\tau_q \in lp_{i,r}} \eta_q^+(W_{i,l}) \times (C_q^A + C_q^R) \tag{6}$$

Proof. As proven in Lemma 2, tasks running on the local core π_l during $W_{i,l}$ can suffer at most $\Phi_i(W_{i,l})$ inter-core memory blockings. As the precise memory access time of the lower priority tasks running on a remote core is not known at design-time, if $\Phi_i(W_{i,l}) \geq \mu_i(W_{i,l})$, there can be a scenario in which all the memory phases of all lower priority tasks released on

all the remote cores during $W_{i,l}$ can cause inter-core memory blocking to tasks executing on the local core π_l during $W_{i,l}$.

Thus, the maximum inter-core memory blocking that can be caused by one job of a lower priority task τ_q released on a remote core π_r, i.e., $\tau_q \in lp_{i,r}$, is upper-bounded by the sum of the WCET of its A- and R-phases, i.e., $C_q^A + C_q^R$. So, the maximum inter-core memory blocking that can be caused by all the jobs of task τ_q during $W_{i,l}$ is upper-bounded by $\eta_q^+(W_{i,l}) \times (C_q^A + C_q^R)$. Similarly, the maximum inter-core memory blocking that can be caused by all lower priority tasks executing on a remote core π_r during $W_{i,l}$ is upper-bounded by $\sum_{\tau_q \in lp_{i,r}} \eta_q^+(W_{i,l}) \times (C_q^A + C_q^R)$. Finally, the maximum inter-core memory blocking that can be suffered by tasks executing on the local core due to lower priority tasks running on *all the remote cores* during $W_{i,l}$ is upper-bounded by $\sum_{r=1, r\neq l}^{m} \sum_{\tau_q \in lp_{i,r}} \eta_q^+(W_{i,l}) \times (C_q^A + C_q^R)$. □

Maximum Inter-Core Memory Blocking for Case 2: We know that all tasks that execute on core π_l during $W_{i,l}$ can suffer at most $\Phi_i(W_{i,l})$ inter-core memory blockings. If $\Phi_i(W_{i,l}) < \mu_i(W_{i,l})$, we need to extract $\Phi_i(W_{i,l})$ number of memory phases released by all the lower priority tasks running on all the remote cores during $W_{i,l}$ that can lead to the maximum inter-core memory blocking. To do this computation, we introduce the following notations.

Let M be an ordered set that contains the WCET of all the memory phases (i.e., A- and R-phases) of all the *lower priority tasks released on all the remote cores* during any time interval of length $W_{i,l}$, *sorted in a non-increasing order* as follows:

$$M = \{C_1^{A/R}, C_2^{A/R}, \ldots, C_V^{A/R} \mid C_x^{A/R} \geq C_{x+1}^{A/R}\} \quad (7)$$

where $C_x^{A/R}$ denotes the WCET of either A- or R-phase of a lower priority task released on a remote core π_r during $W_{i,l}$. In Equation 7, the index V is equal to the $\mu_i(W_{i,l})$.

The maximum inter-core memory blocking for case 2 is then computed using the following lemma.

Lemma 5. *If $\Phi_i(W_{i,l}) < \mu_i(W_{i,l})$, then the maximum inter-core memory blocking that can be suffered by tasks executing on the local core π_l due to lower priority tasks running on all remote cores during any time interval of length $W_{i,l}$ is upper-bounded by $B_i^{Mem}(W_{i,l})$, where*

$$B_i^{Mem}(W_{i,l}) = \sum_{x=1}^{\Phi_i(W_{i,l})} C_x^{A/R} \text{ where } C_x^{A/R} \in M \quad (8)$$

Proof. As proven in Lemma 2, tasks running on the local core π_l during $W_{i,l}$ can suffer at most $\Phi_i(W_{i,l})$ inter-core memory blockings. As $\Phi_i(W_{i,l}) < \mu_i(W_{i,l})$, we need to extract $\Phi_i(W_{i,l})$ number of memory phases of the lower priority tasks released on all the remote cores during $W_{i,l}$ that can lead to the maximum inter-core memory blocking. As we cannot predict the actual schedule of task executions on remote cores, we do not know the specific memory phases of lower priority tasks running on remote cores that can cause inter-core memory blocking during $W_{i,l}$. Therefore, to maximize the inter-core memory blocking, we choose $\Phi_i(W_{i,l})$ number of memory phases with the *largest execution times* among all the memory phases of lower priority tasks released on all the remote cores during $W_{i,l}$. This is achieved by summing up the first $\Phi_i(W_{i,l})$ elements of M, which contains the WCET of all memory phases of all lower priority tasks released on all remote cores during $W_{i,l}$. The Lemma follows. □

V. WCRT Analysis

In fixed-priority limited preemptive scheduling, the WCRT of task τ_i is observed during the longest level-i busy window [1]. Having bounded all the terms that can contribute to the length of level-i busy window on core π_l, i.e., $I_i(W_{i,l})$, B_i, $I_i^{Mem}(W_{i,l})$, and $B_i^{Mem}(W_{i,l})$, the length of $W_{i,l}$ is given by the first positive fixed-point solution of the following equation:

$$\begin{aligned} W_{i,l} = I_i(W_{i,l}) + B_i + \eta_i^+(W_{i,l}) \times C_i \\ + I_i^{Mem}(W_{i,l}) + B_i^{Mem}(W_{i,l}) \end{aligned} \quad (9)$$

where $\eta_i^+(W_{i,l}) \times C_i$ considers the maximum contribution of all jobs released by task τ_i during $W_{i,l}$.

Having bounded the length of the level-i busy window $W_{i,l}$, we compute the maximum number of jobs of task τ_i that can execute on core π_l during $W_{i,l}$ using the following equation.

$$K_i = \eta_i^+(W_{i,l}) \quad (10)$$

Using the values of $W_{i,l}$ and K_i, we can now compute the WCRT of task τ_i. For this, we need to analyze the response time of each job of task τ_i that execute during $W_{i,l}$. Let $\tau_{i,k}$ be the k^{th} job of task τ_i that execute during $W_{i,l}$. To compute the response time of $\tau_{i,k}$, we compute the latest start time of the R-phase of $\tau_{i,k}$ as it can be delayed by tasks running on the local core/remote cores until the start of its R-phase.

The latest start time of the R-phase of $\tau_{i,k}$ is denoted by $s_{i,k}^R$, where $s_{i,k}^R$ is given by the first positive solution to the fixed-point iteration on the following equation.

$$\begin{aligned} s_{i,k}^R = I_i(s_{i,k}^R) + B_i + ((k-1) \times C_i) + C_i^A + C_i^E \\ + I_i^{Mem}(s_{i,k}^R) + B_i^{Mem}(s_{i,k}^R) \end{aligned} \quad (11)$$

where $I_i(s_{i,k}^R)$ is the maximum intra-core interference suffered by $\tau_{i,k}$ during $s_{i,k}^R$, given by Equation 1. The term B_i is the maximum intra-core blocking, given by Equation 2. The term $(k-1) \times C_i$ considers the WCET of $k-1$ jobs of task τ_i. We consider the WCET of the A-phase and the E-phase of τ_i using $C_i^A + C_i^E$ while computing the latest start time of the R-phase of $\tau_{i,k}$. The term $I_i^{Mem}(s_{i,k}^R)$ considers the maximum inter-core memory interference suffered by $\tau_{i,k}$ during $s_{i,k}^R$, given by Lemma 1. The term $B_i^{Mem}(s_{i,k}^R)$ considers the maximum inter-core memory blocking that can be suffered by $\tau_{i,k}$ during $s_{i,k}^R$ and can be computed using Lemma 2 to Lemma 5.

As $s_{i,k}^R$ appears on both sides of Equation 11, it can be solved iteratively by initializing $s_{i,k}^R = C_i^A + C_i^E + B_i + \sum_{\tau_h \in hp_{i,l}} C_h$. The start time $s_{i,k}^R$ will then be given by the smallest positive value of $s_{i,k}^R$ for which Equation 11 converges. Having computed the value of $s_{i,k}^R$, we can compute the response time $R_{i,k}$ of $\tau_{i,k}$ using the following equation.

$$R_{i,k} = s_{i,k}^R + C_i^R \quad (12)$$

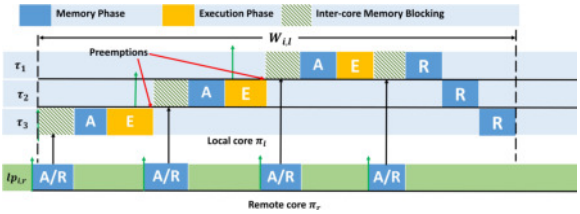


Fig. 3: Maximum number of inter-core memory blockings when E-phases are non-preemptive

Finally, we can compute the WCRT of task τ_i by analyzing the response time of each job of τ_i that executes during $W_{i,l}$ and consider the largest response time among all the jobs, i.e.,

$$R_i^{max} = \max_{k \in [1, K_i]} \{R_{i,k}\} \tag{13}$$

where the computation of K_i is obtained using Equation 10.

A taskset Γ is said to be schedulable only if the WCRT of each task in the taskset is less than or equal to its relative deadline, the utilization of each core is less than or equal to the core's capacity, i.e., 1, and the total memory utilization of the taskset is less than or equal to 1, i.e., $\sum_{\tau_i \in \Gamma} \frac{C_i^A + C_i^R}{T_i} \leq 1$.

VI. Analyzing the Impact of Preemption Point Selection

Most existing works in the state-of-the-art that focus on MCS of PREM/3-phase tasks assume non-preemptive scheduling at the core level [15], [18]. Considering that in general, limited preemptive based approaches tend to perform better than non-preemptive approaches in terms of schedulability, a few existing works have also considered limited preemptive scheduling-based MCS approaches [19]. The TP-based MCS approach presented in Section IV also assumes limited preemptive scheduling where tasks executing on the same core can be preempted anytime during their E-phases. However, in this section, we will explore how preemption point selection can impact the TP-based MCS, by considering an alternate task scheduling approach where E-phases of tasks are also assumed to be non-preemptive, i.e., task preemptions are only allowed at the boundary of task phases. First, we will present an example that shows how this alternate preemption point selection can reduce the inter-core memory blocking of tasks. We will then discuss how the analysis presented in Section IV needs to be adapted when considering this preemption scheme.

Example 2: *For the same example depicted in Figure 2, if the E-phases are non-preemptive, the resulting schedule is shown in Figure 3. Due to non-preemptive E-phases, each E-phase executes without being preempted and the local core can suffer at most one memory blocking from remote cores for each E-phase that executes on the local core. For instance, we can see in Figure 3 that the A-phase of τ_1, τ_2 only starts after the completion of an E-phase. Since a memory blocking can be suffered when the local core executes an E-phase, τ_1, τ_2 suffer memory blocking before their A-phases. However, this in turn leads to a scenario in which the R-phases of τ_2, τ_3 do not suffer inter-core memory blocking. This happens because the local core does not execute any E-phase after the R-phase completion of τ_1 and there is always a ready memory phase on the local core, thus, memory scheduler will not schedule a memory phase of any lower priority task of a remote core. Consequently, for the same example scenario, at most 4 memory blockings can be suffered by tasks executing on the local core when the E-phase are non-preemptive in comparison to the 6 memory blockings suffered by the local core when E-phases are preemptive (see Figure 2).*

When analyzing the impact of non-preemptive E-phases of tasks on the TP-based MCS approach, the computation of intra-core interference and inter-core memory interference remains exactly the same as presented in Section IV, i.e., the intra-core interference can still be computed using Equation 1 and the inter-core memory interference will be upper bounded using Equation 3 (Lemma 1). However, the computation of intra-core blocking and inter-core memory blocking needs to be adapted, which is explained as follows.

A. Bounding Intra-Core Blocking

When each phase of a 3-phase task executes non-preemptively, task preemptions can only happen at the start/end of E-phases. So, if task τ_i is released when a lower priority task is already executing, τ_i suffers intra-core blocking from at most one phase (i.e., A, E, or R-phase) executing on the same core as τ_i. Therefore, the maximum intra-core blocking B_i suffered by task τ_i is given by the WCET of the largest A, E, or R-phase among all the tasks in $lp_{i,l}$, i.e.,

$$B_i = \max\Big(\max_{\forall \tau_j \in lp_{i,l}} \{C_j^A\}, \max_{\forall \tau_j \in lp_{i,l}} \{C_j^E\}, \max_{\forall \tau_j \in lp_{i,l}} \{C_j^R\}\Big) \tag{14}$$

B. Bounding Inter-core Memory Blocking

As discussed in Section IV-D, the inter-core memory blocking of tasks depends on the value of $\Phi_i(W_{i,l})$, i.e., the maximum number of inter-core memory blockings that can be suffered by all tasks executing on the local core during $W_{i,l}$, and $\mu_i(W_{i,l})$, i.e., the maximum number of inter-core memory blockings that can be caused by all lower priority tasks executing on all the remote cores during $W_{i,l}$. When considering non-preemptive execution of E-phases of tasks, the computation of $\mu_i(W_{i,l})$ remains unchanged and it can be computed using Equation 5 (Lemma 3) as detailed in Section IV-D2. However, the computation of $\Phi_i(W_{i,l})$ needs to be adapted, which is done using the following lemma.

Lemma 6. *If preemptions are allowed only at the start/end of E-phases of tasks, then the maximum number of times that tasks executing on core π_l can suffer inter-core memory blocking during $W_{i,l}$ is upper-bounded by $\Phi_i(W_{i,l})$, where*

$$\Phi_i(W_{i,l}) = \sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l}) + 1 \tag{15}$$

Proof. When considering non-preemptive E-phases of tasks, each E-phase that executes on the local core during $W_{i,l}$ will run until its completion. This implies that at most one inter-core memory blocking can be caused by a lower priority task

of a remote core at the completion of each E-phase that execute on the local core during $W_{i,l}$. Consequently, the maximum number of inter-core memory blockings that can be suffered by all tasks that execute on the local core π_l during $W_{i,l}$ is equal to the maximum number of E-phases that execute on the local core π_l during $W_{i,l}$. As each job releases one E-phase, the maximum number of E-phases that can be released by a task τ_h can execute during $W_{i,l}$ is upper-bounded by $\eta_h^+(W_{i,l})$. Similarly, the maximum number of E-phases that can be released by all tasks in $hep_{i,l}$ during $W_{i,l}$ is upper-bounded by $\sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l})$.

Additionally, we need to consider one inter-core memory blocking that can be suffered on the local core π_l at the start of the level-i busy window $W_{i,l}$. Therefore, the maximum number of inter-core memory blockings that can be suffered by tasks that can execute on the local core π_l during $W_{i,l}$ is upper bounded by $\sum_{\tau_h \in hep_{i,l}} \eta_h^+(W_{i,l})+1$. The Lemma follows. □

Having computed the value of $\Phi_i(W_{i,l})$ using Lemma 6, the maximum inter-core memory blocking can be computed using the exact same steps as detailed in Section IV-D. However, knowing that the maximum inter-core memory blocking that can be suffered by the tasks during $W_{i,l}$ depends both on the value of $\Phi_i(W_{i,l})$ and $\mu_i(W_{i,l})$, and the value of $\Phi_i(W_{i,l})$ computed using Lemma 6 can be different from the value of $\Phi_i(W_{i,l})$ when computed using Lemma 2. Therefore, the resulting values of the maximum inter-core memory blocking, i.e., $B_i^{Mem}(W_{i,l})$, computed considering non-preemptive E-phases can be different from the values obtained considering preemptive E-phases (i.e., analysis detailed in Section IV-D).

Finally, the WCRT for non-preemptive E-phase based scheduling can be computed using the exact same procedure detailed in Section V, with B_i computed using Equation 14 and $B_i^{Mem}(W_{i,l})$ computed using Lemma 3 to Lemma 6.

VII. Experimental Evaluation

In this section, we discuss the experiments that were performed to evaluate the effectiveness of the proposed TP-based MCS in comparison to the existing PP-based MCS [15]. For the default configuration, we consider a multicore system composed of 4 cores and a taskset size of 32 tasks in which 8 tasks are assigned to each core. Tasks utilization U_i is randomly generated using the UUnifast-discard algorithm [5]. Task periods T_i are randomly generated in the range of [100-1000] using log-uniform distribution. The WCET C_i is then assigned by applying the relation $C_i = U_i \times T_i$. The Memory Demand (MD) is assigned a random value in the range [10%-50%] of the WCET, i.e., $MD = rand(10\%, 50\%) \times C_i$. The WCET of the A- and R-phases[2] is given by $C_i^A = C_i^R = MD/2$ and the WCET of the E-phase is given by $C_i^E = C_i - (C_i^A + C_i^R)$. Task priorities are assigned globally using rate monotonic algorithm [8]. Task deadlines are implicit (i.e., $D_i = T_i$).

We evaluate the performance of the proposed TP-based MCS in comparison to the existing PP-based MCS [15] by varying: 1) the core utilization (i.e., utilization of each core); 2) the number of cores; 3) task memory demands; and 4) the task period range. We use taskset schedulability, i.e., the percentage of schedulable tasksets, as a metric to evaluate the performance of each approach. For each point depicted in each plot, 1000 tasksets were randomly generated. In all the experiments, the proposed TP-based MCS that considers preemptive E-phases is marked as "TP-MCS-PE" whereas the proposed TP-based MCS that considers non-preemptive E-phases is marked as "TP-MCS-NPE". Similarly, the existing PP-based MCS [15] is marked as "PP-MCS". In all plots, the x-axis represents the core utilization and the y-axis represents the percentage of schedulable tasksets for all the analyzed approaches.

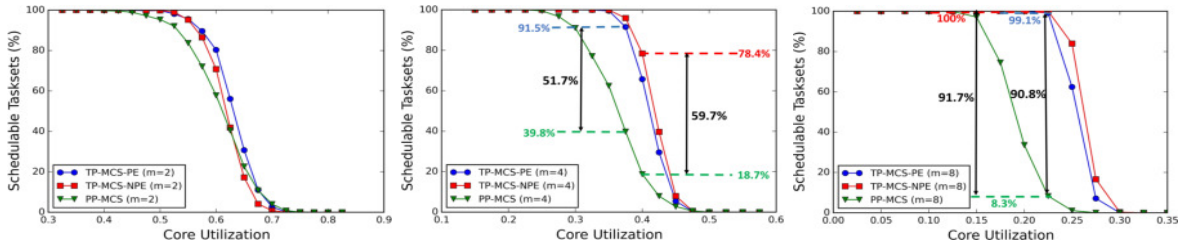
1) Varying Core Utilization: In this experiment, we varied the core utilization of each core under the default configuration from 0.025 to 1 in steps of 0.025 and plotted the resulting number of schedulable tasksets in Figure 4b. Figure 4b shows that the taskset schedulability of all the approaches decreases by increasing the core utilization. The is mainly because increasing the core utilization increases the task utilizations, which, in turn, increases the WCET of tasks. This increase in C_i results in an increase in the values of C_i^A, C_i^E, and C_i^R. Consequently, the intra-core interference/blocking and inter-core memory interference/blocking increases, resulting in decreasing taskset schedulability. Nevertheless, the proposed TP-MCS-PE and TP-MCS-NPE approaches outperform the PP-based MCS. In particular, TP-MCS-PE analysis was able to schedule around 51% of more tasksets as compared to PP-based MCS at the core utilization value of 0.375. Similarly, the TP-MCS-NPE analysis was able to schedule around 59% of more tasksets as compared to PP-based MCS at the core utilization value of 0.40. This happens due to fixed task priority based memory scheduling used by proposed TP-based MCS that reduces the inter-core memory interference/blocking suffered by tasks in comparison to the PP-based MCS. Also, due to the use of limited preemptive scheduling, the proposed TP-based MCS approach reduces the intra-core blocking in comparison to the PP-based MCS that assume non-preemptive task executions. Figure 4b also confirms that TP-MCS-NPE outperforms the TP-MCS-PE due to a tighter estimation of inter-core memory blocking.

2) Varying Number of Cores: In this experiment, we varied the number of cores, which in turn, also varies the number of tasks in the taskset[3]. We varied the number of cores m between 2 to 8 along with the core utilization. As shown in Figure 4, increasing the value of m results in a decrease in taskset schedulability for all the considered approaches. This is because increasing the number of cores also increases the number of remote cores and thus the number of tasks running on remote cores. This increases the inter-core memory interference and memory blocking, eventually resulting in decreasing taskset schedulability.

We can see in Figure 4c that increasing the number of

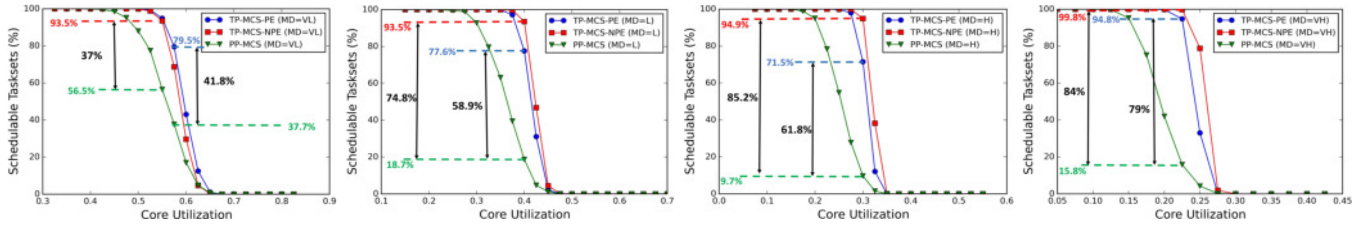
[2]For PP-based MCS [15], we consider one memory phase of length MD

[3]Per core tasks remains the same but increasing/decreasing number of cores results in increasing/decreasing the total tasks in the taskset



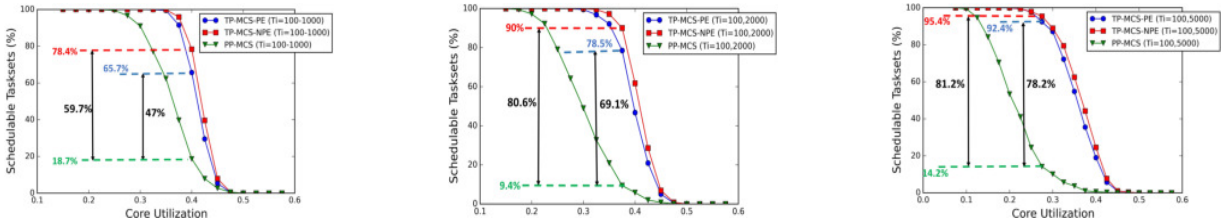
(a) Varying Core Utilization for m=2 (b) Varying Core Utilization for m=4 (c) Varying Core Utilization for m=8

Fig. 4: Varying Core Utilization and Number of Cores



(a) VL Configuration (MD=5%-20%) (b) L Configuration (MD=20%-40%) (c) H Configuration (MD=40%-60%) (d) VH Configuration (MD=60%-80%)

Fig. 5: Varying Core Utilization for Different MD Configurations



(a) Task Period Range of 100-1000 (b) Task Period Range of 100-2000 (c) Task Period Range of 100-5000

Fig. 6: Varying Core Utilization for Different Task Period Ranges

cores tends to increase the difference between our proposed approaches and the PP-based MCS. In particular, the TP-MCS-PE was able to schedule around 90% more tasksets than PP-based MCS at the core utilization value of 0.225. Similarly, the TP-MCS-NPE was able to schedule around 91% of more tasksets than PP-based MCS at the core utilization value of 0.225. On the contrary, the gain of TP-MCS-PE and TP-MCS-NPE over PP-based MCS was negligible for $m = 2$. In fact, the PP-based MCS was able to perform slightly better than TP-MCS-NPE for some values of core utilization as shown in Figure 4a. We explain these variations as follows: when the number of cores are smaller then the impact of inter-core memory interference and memory blocking is not that significant due to fewer tasks on remote cores. This results in producing similar performance of all the approaches. Similarly, for $m = 2$, tasks scheduled using PP-based MCS suffer inter-core memory interference from only one remote core, thereby, resulting in a slightly better performance than the TP-MCS-NPE approach. Note that for $m = 2$, TP-MCS-PE also performs slightly better than TP-MCS-NPE. This is mainly due to the fact that TP-MCS-PE provides a slightly tighter bound on the intra-core blocking than TP-MCS-NPE whose impact is maximized when the taskset size is smaller.

3) Varying Memory Demand (MD): In this experiment, we vary the Memory Demand (MD) of all tasks in the taskset along with the core utilization. For this, we consider 4 different configurations based on the value of MD, that are, Very Light (VL) MD, i.e., MD=$(5\%, 20\%) \times C_i$, Light (L) MD, i.e., MD=$(20\%, 40\%) \times C_i$, Heavy (H) MD, i.e., MD=$(40\%, 60\%) \times C_i$, Very Heavy (VH) MD, i.e., MD=$(60\%, 80\%) \times C_i$. The value of MD is assigned to each task in the taskset randomly as per the chosen configuration.

As shown in Figure 5, all the approaches perform the best in the VL configuration and the worst in the VH configuration. This is intuitive as an increase in the value of MD also increases the WCET of memory phases that results in increasing the inter-core memory interference and memory blocking. However, we can see that for all configurations, proposed TP-based MCS approaches outperform the PP-based MCS. In fact, for H and VH configurations, the difference between our proposed approaches and the PP-based MCS becomes more prominent. This is due to increasing the length of memory phases, that directly impacts the memory interference of tasks.

4) Varying Task Periods: In this experiment, we vary the core utilization for different task period ranges. For this, we consider three task period ranges that are [100-1000], [100-2000], [100-5000]. As shown in Figure 6, number of tasks deemed schedulable by all the approaches is reduced by increasing the task period ranges. This is mainly because, by increasing the task period, the WCET of execution and

memory phases of tasks also increases. This has a direct impact on the inter-core memory interference/blocking of tasks. However, we can see in Figure 6 that the proposed TP-based MCS approaches outperforms the PP-based MCS for the all task period ranges. As proposed TP-based MCS provides a tighter bound on the memory interference, the gain of the proposed analyses over PP-based MCS increases with the increase in task period range due to the higher impact of inter-core memory interference/blocking.

VIII. Related Work

Sharing of main memory is a major source of contention in COTS multicore platforms, and several existing works have focused on the problem of inter-core memory interference in multicore platforms [2], [3], [6], [10], [13]–[15], [18]–[20].

The phased execution models such as the PREM [11] and the 3-phase task models [4], [9] along with the Memory Centric Scheduling (MCS) [10], [15], [18]–[20] have been studied in several works to solve the problem of inter-core memory interference. Pagetti et al. [10] proposed a framework to generate an offline task schedule at the system level that minimizes inter-core memory interference. However, enforcing such an offline schedule may not be possible in all the scenarios. Yao et al. [19] proposed TDMA-based MCS that uses static TDMA slots to schedule the memory accesses under partitioned scheduling. Unlike conventional TDMA scheduling [14], TDMA-based MCS [19] allows preemptions during the E-phases so that the cores can efficiently utilize the available TDMA slot by prioritizing memory phases. The concept of MCS was then extended to the global scheduling in [20]. Schwäricke et al. [15] have presented fixed Processor Priority (PP) based MCS that considers two-level scheduling approach: 1) fixed-priority non-preemptive scheduling at the core level; and 2) fixed processor priority to schedule the memory phases at the system level. Due to the two-level scheduling used by the PP-based MCS, tasks with higher local priorities, i.e., tasks with shorter deadlines/periods, that execute on lower global priority cores can suffer high memory interference, i.e., from all tasks that execute on all higher priority cores, and can potentially result in deadline misses.

Works like [3], [12] have shown that task priority based memory scheduling can significantly reduce memory interference of tasks. However, none of the existing works on MCS have considered task priority-based memory scheduling. Therefore, in this work, we investigated TP-based MCS and showed how it can improve the state-of-the-art.

IX. Conclusion

This work extends the notion of memory centric scheduling to consider task priority based memory scheduler. We showed how the memory interference of 3-phase tasks executing on a multicore platform can be bounded, assuming memory requests are served based on the priority of the generating task. Contrary to most works in the state-of-the-art, our analysis supports limited preemptive scheduling and also investigates the impact of preemption point selection on the inter-core memory interference suffered by tasks. Experimental results reveal that the proposed TP-based MCS can schedule up to 91% more tasksets than the state-of-the-art PP-based MCS [15]. As future work, we plan to extend our analysis by using the Direct-memory-access (DMA) engines.

Acknowledgments. This work was partially supported by European Union's Horizon 2020 -The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 732505. Project "TEC4Growth - Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01-0145-FEDER000020" financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement; also by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (UIDP/UIDB/04234/2020); by FCT and the Portuguese National Innovation Agency (ANI), under the CMU Portugal partnership, through the European Regional Development Fund (ERDF) of the Operational Competitiveness Programme and Internationalization (COMPETE 2020), under the PT2020 Partnership Agreement, within project FLOYD (POCI-01-0247-FEDER-045912), also by FCT under PhD grant 2020.09532.BD.

References

[1] R. J. Bril, J. J. Lukkien, and W. F. J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *ECRTS'07*, pages 269–279, 2007.

[2] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo. A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling. In *IEEE RTAS 2020*, pages 239–252, 2020.

[3] Robert I. Davis, Sebastian Altmeyer, Leandro S. Indrusiak, Claire Maiza and·Vincent Nelis, and Jan Reineke. An extensible framework for multicore response time analysis. *Real-Time Systems*, July 2017.

[4] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and W. Puffitsch. Predictable Flight Management System Implementation on a Multicore Processor. In *ERTS'14*, February 2014.

[5] P. Emberson, R. Stafford, and R.I. Davis. Techniques for the synthesis of multiprocessor tasksets. *WATERS'10*, 01 2010.

[6] Mohamed Hassan and Rodolfo Pellizzoni. Analysis of Memory-Contention in Heterogeneous COTS MPSoCs. In *ECRTS 2020*, volume 165 of *LIPIcs*, pages 23:1–23:24, Dagstuhl, Germany, 2020.

[7] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. *[1990] 11th RTSS*, pages 201–209, 1990.

[8] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[9] Claudio Maia, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Perez. A closer look into the AER Model. In *ETFA 2016*. IEEE, 2016.

[10] Claire Pagetti, Julien Forget, Heiko Falk, Dominic Oehlert, and Arno Luppold. Automated generation of time-predictable executables on multi-core. In *RTNS 2018*, POITIERS, France, October 2018.

[11] 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 *IEEE RTAS*, 2011.

[12] Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar. Cache persistence-aware memory bus contention analysis for multicore systems. In *DATE 2020*, pages 442–447, 2020.

[13] Simon Schliecker and Rolf Ernst. Real-time performance analysis of multiprocessor systems with shared memory. *ACM Transactions on Embedded Computing Systems*, 10(2):1–27, December 2010.

[14] Andreas Schranzhofer et al. Timing analysis for tdma arbitration in resource sharing systems. In *IEEE RTAS 2010*, pages 215–224, 2010.

[15] Gero Schwäricke, Tomasz Kloda, Giovani Gracioli, Marko Bertogna, and Marco Caccamo. Fixed-Priority Memory-Centric Scheduler for COTS-Based Multiprocessors. In *ECRTS 2020*, LIPIcs, 2020.

[16] M. R. Soliman and R. Pellizzoni. Prem-based optimal task segmentation under fixed priority scheduling. In *ECRTS*, 2019.

[17] Muhammad R. Soliman, Giovani Gracioli, Rohan Tabish, Rodolfo Pellizzoni, and Marco Caccamo. Segment streaming for the three-phase execution model: Design and implementation. In *RTSS 2019*, 2019.

[18] R. Tabish, R. Mancuso, S. Wasly, R. Pellizzoni, and M. Caccamo. A real-time scratchpad-centric os with predictable inter/intra-core communication for multi-core embedded systems. *Real-Time Systems*, 55, 2019.

[19] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48, 11 2012.

[20] 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.