CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

Improved Memory Contention Analysis for the 3-Phase Task Model

Jatin Arora*
Syed Aftab Rashid
Geoffrey Nelissen
Cláudio Maia*
Eduardo Tovar*

*CISTER Research Centre

CISTER-TR-240506

2024/08/21

Improved Memory Contention Analysis for the 3-Phase Task Model

Jatin Arora*, Syed Aftab Rashid, Geoffrey Nelissen, 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, gnn@isep.ipp.pt, clrrm@isep.ipp.pt, emt@isep.ipp.pt

https://www.cister-labs.pt

Abstract

In multiprocessor-based real-time systems, main memory is identified as a major bottleneck in the worst-case timing analysis of tasks. Phased execution models such as the 3-phase task model, i.e., that divides the execution of tasks into distinct computation and memory phases, have shown to be a good candidate to tackle the memory contention problem. The 3-phase execution model in particular has gained much attention from both academia and industry as it limits when tasks can access main memory to pre-defined phases. Information on when those phases may happen and their length can then be leveraged to build a fine-grained memory contention analysis. However, the existing work that focus on the memory contention analysis for 3-phase tasks may overestimate the memory contention caused by interfering write requests. This yields pessimistic bounds on the total memory contention suffered by tasks which in turn leads to pessimistic worst-case execution time (WCET) and worst-case response time (WCRT) bounds. In this work, we improve the state-of-the-art memory contention analysis for 3-phase tasks by (i) tightly bounding the memory contention that can be suffered due to write requests; and (ii) providing a new memory contention-aware WCET analysis.

Improved Memory Contention Analysis for the 3-Phase Task Model

Jatin Arora[*§¶] , Syed Aftab Rashid[†], Geoffrey Nelissen[‡], Cláudio Maia[*], Eduardo Tovar[*]
[*]CISTER, ISEP, Porto, Portugal [§]VORTEX CoLab, Portugal [†]Hitachi Energy Research, Baden-Dättwil, Switzerland
[‡]Eindhoven University of Technology, Eindhoven, the Netherlands

***Abstract*—In multiprocessor-based real-time systems, main memory is identified as a major bottleneck in the worst-case timing analysis of tasks. Phased execution models such as the 3-phase task model, i.e., that divides the execution of tasks into distinct computation and memory phases, have shown to be a good candidate to tackle the memory contention problem. The 3-phase execution model in particular has gained much attention from both academia and industry as it limits when tasks can access main memory to pre-defined phases. Information on when those phases may happen and their length can then be leveraged to build a fine-grained memory contention analysis. However, the existing work that focus on the memory contention analysis for 3-phase tasks may overestimate the memory contention caused by interfering write requests. This yields pessimistic bounds on the total memory contention suffered by tasks which in turn leads to pessimistic worst-case execution time (WCET) and worst-case response time (WCRT) bounds. In this work, we improve the state-of-the-art memory contention analysis for 3-phase tasks by (i) tightly bounding the memory contention that can be suffered due to write requests; and (ii) providing a new memory contention-aware WCET analysis.**

I. Introduction

The adoption of multicore platforms in *hard real-time systems*, i.e., systems that run applications with stringent timing requirements, is still under the scrutiny of academia and industry. The main challenge hindering the adoption of commercial off-the-shelf (COTS) multicore platforms in hard real-time systems is their temporal unpredictability. This unpredictability stems from the sharing of different hardware resources, such as shared caches, interconnects, and the main memory. Tasks from different cores may compete to access these resources and this competition may result in *shared resource contention*. All hardware resources that are shared among multiple cores of a multicore platform can be subjected to contention, however, it has been identified in the state-of-the-art [1], [7], [16], [18], [20], [32], [34] that the contention on *main memory* has the most significant impact on the timing behavior of applications (see surveys [22], [24]).

Among several existing solutions to the memory contention problem, the use of *phased execution models* is one of the most promising [3], [7], [9], [12], [26], [29], [33]. These phased execution models such as the *3-phase task model* [9], [26] allow to derive precise bounds on the memory and bus contention suffered by tasks [2]–[4], [7], [13], [14], [23], [30], thanks to dividing the execution of tasks into distinct *execution* and *memory phases*. Specifically, the 3-phase execution model has (1) an *Acquisition* or *A-phase*, during which the task prefetches its required data/instructions from main memory and save them in the core's local memory, (2) the *Execution* or *E-phase*, during which computation is performed on the preloaded data without generating any main memory accesses, and (3) the *Restitution* or *R-phase* that writes back the modified data of the task to the main memory. These phases are categorized into *computation* only phase, i.e., the E-phase, and *memory* only phases, i.e., the A and R-phases. This model allows one to infer the specific *time intervals* in which *memory accesses* can happen as well as the *memory access type*, i.e., read or write, happening in that interval. This information is used to derive precise estimates on the maximum main memory contention suffered by tasks. Several existing works [2]–[4], [13], [14], [23], [30] have shown that the 3-phase task model can provide significantly tighter memory contention estimates in comparison to the generic task model, i.e., memory accesses can happen anytime during the task execution. However, most of these existing works rely on a *black-box memory model*, i.e., they assume main memory and memory bus as a single resource that can handle at most one request at a time and remains busy while handling the memory request. These assumptions may lead to both pessimistic and optimistic bounds as they ignore factors such as the memory organization, number/type of competing requests, locality of the requested memory block, memory controller arbitration policies that can potentially reorder memory requests, etc.

To address this issue, some other works in the state-of-the-art [1], [7], [16], [18], [20], [32], [34] have also proposed *white-box* memory model-based approaches to bound the memory contention suffered by tasks. Specifically, these approaches consider *Dynamic Random Access Memory* (DRAM) as the main memory and take into account the organization of DRAM and the low-level arbitration mechanism employed by the DRAM memory controller. To the best of our knowledge, the only work in the literature that relies on the white-box memory modeling-based approach to bound memory contention for the 3-phase task model was proposed in [7]. The memory contention analysis in [7] assumes that the memory controller employs *write batching*[1] [8] in which the memory controller prioritizes read requests over write requests. Write batching ensures that if there are any pending read memory requests, the write requests are enqueued in

[¶] Corresponding author

[1]The information on write batching in DRAM is detailed in Section II.

the write buffer until the *watermarking threshold* is reached. Once the number of writes in the write buffer reaches the watermarking threshold, write requests are served in *batches*. This improves the turnaround time of the data bus [10], i.e., responsible for the data transfer between the memory controller and memory banks, as serving memory requests of the same type is more efficient. Even though the analysis presented in [7] is safe, it has some limitations. For example, the analysis of [7] overestimates the memory contention that can be caused by interfering write requests to the task under analysis. Specifically, the analysis in [7][2] assumes that either *one batch of write requests* will cause interference to each read request of task τ_i or the overall delay suffered by the A-phase of task τ_i is given by the length of the write-buffer plus the *R-phases of jobs of all tasks* that can be released on all other cores during the execution of the A-phase of the task under analysis. This assumption is pessimistic because, in the 3-phase task model, an R-phase can only be issued by a core after the completion of an A-phase. In such a scenario, *the actual number of R-phases that can be issued by a core depends on the number of A-phases that can be completed on that core* and not necessarily on the number of jobs released by tasks on that core during a given time window. Hence, the bound on the total memory contention is often overestimated, yielding pessimistic bounds on the WCET and WCRT of tasks.

To address this limitation, this paper presents a fine-grained white-box memory contention analysis for 3-phase tasks. For this, we first identify the sources of pessimism in the existing analysis [7]. We then show how such pessimism can be tackled by accurately quantifying the maximum memory contention caused by write memory requests. Finally, we integrate the bounds on the maximum memory contention into the WCET of tasks to derive tighter bounds on the WCRT of tasks.

In sum, this work has the following **contributions**.

1. We propose a memory contention analysis for 3-phase tasks that provides a tighter bound on memory contention that can be caused by write memory requests by showing that interfering write requests depend on interfering read requests.

2. We show how the derived bounds on memory contention can be incorporated in the WCET of tasks to derive a fine-grained memory contention-aware WCET analysis.

3. We perform an extensive experimental evaluation to compare the performance of the proposed memory contention analysis against the state-of-the-art [7]. Results reveal that the proposed memory contention analysis can perform significantly better under various configurations.

II. Background

This section introduces relevant background concepts and notions related to DRAM. Similarly to most of the existing approaches [7], [16], [18], [20], [34], we consider the *DDR3 DRAM*.

The DRAM is composed of 1) a *memory controller* that is responsible for determining the order in which the memory requests from all the cores will be served; 2) a *memory chip*, i.e., an array of memory cells that store the data; 3) a *command bus*, i.e., the interconnect through which the memory controller issues all commands to the memory chip, and 4) a *data bus*, i.e., the interconnect through which data is transferred from/to the memory controller to/from the memory chip.

The DRAM is organized into multiple *ranks* in which each rank is composed of multiple *banks*. Each bank is composed of rows and columns that store the data. Each memory bank has a *row buffer* that stores the last row accessed during the most recent access to that bank. The row stored in the buffer of a bank is said to be *activated*. The order in which the memory requests from all cores will be served by the DRAM is managed by the memory controller. Memory requests targeting each bank are enqueued into *per-bank queues* that are managed by a so called *intra-bank scheduler*. Each per-bank queue is then further exposed to the *inter-bank scheduler* that schedules memory requests from the per-bank queues by issuing the required commands on the command bus and transfers data on the data bus.

Commonly, COTS multicore processors are designed with the goal of achieving high throughput which is achieved by reordering memory requests such that memory resources can be efficiently utilized. To achieve this, the intrabank scheduler, i.e., each per-bank queue, uses the First-Ready First-Come-First-Served (FR-FCFS) scheduling policy [7], [18], [20] which means that 1) memory requests that result in a row-hit are prioritized over memory requests that result in a row-miss; and 2) in case of a tie, older memory requests are prioritized over newer memory requests. Furthermore, at the inter-bank level, the Round-Robin (RR) scheduling policy is used to serve the memory requests from each per-bank queue [18], [20], [34].

There are three commands that are possibly issued during a DRAM operation. These commands are: 1) the PRE (PREcharge) command which stores the current content of the row buffer to its corresponding row in the DRAM bank; 2) the ACT (ACTivate) command which activates the requested row in the row buffer; and 3) the CAS command which performs the intended read/write operation on the activated row. Based on the status of the row buffer and the requested row by the next memory request enqueued in the per-bank queue, the following are the possible sequences of commands that the memory controller can issue to serve a single memory request [18], [20], [34]:

1. If the requested row is the same as the activated row, i.e., row-hit, then only the CAS command is issued to perform the intended read/write operation on the already activated row.

2. If the bank has an activated row different from the requested row, i.e., row-miss, then the memory controller first issues the PRE command to move back the current content of the row buffer to its corresponding row in the bank. The ACT command is then issued to activate the requested row. Finally, the CAS command is issued to perform the intended read/write operation on the newly activated row.

3. If the bank has no activated row, i.e., row-miss, the ACT

[2]Note that, unlike the proposed work, the existing work [7] does not consider bank partitioning among the A-phases of tasks of different cores.

command is issued to activate the requested row followed by the CAS command.

Each of the above-mentioned commands is issued according to the JEDEC standard [19] which defines all the timing constraints that need to be satisfied while performing a memory operation on the DRAM. As shown in Table I, the JEDEC timing constraints are divided into *intra-bank* timing constraints and *inter-bank* timing constraints. The intra-bank timing constraints can be defined as timing constraints applied between the commands issued to the same bank. For example, if a memory request issued an ACT command to bank 1, it can issue the CAS command to the same bank only after elapsing tRCD cycles. On the other hand, the inter-bank timing constraints can be defined as timing constraints applied between commands of the same type (PRE, ACT, or CAS) issued to any bank. For example, if a memory request on bank 1 issues a CAS command and another memory request on bank 2 also wants to issue a CAS command at the same time, it has to wait for tCCD cycles.

Parameters	Description	Cycles
Intra-bank constraints		
tRCD	ACT to CAS delay	9
tRL	RD to Data Start	9
tRP	PRE to ACT delay	9
tWL	WR to Data Start	8
tRAS	ACT to PRE delay	24
tRC	ACT to ACT (same bank)	33
tWR	Data End of WR to PRE	10
tRTP	Read to PRE delay	5
Inter-bank constraints		
tCCD	CAS to CAS delay	4
tRTW	RD to WR delay	6
tWTR	WR to RD delay	5
tRRD	ACT to ACT (different bank in same rank)	4
tB	Data bus transfer	4
tFAW	Four bank activation window	20

TABLE I: JEDEC timing constraints for DDR3-1333H [19].

COTS multicore processors typically use *write batching* to prioritize read memory requests over write memory requests since writes do not stall the processor pipeline [7], [16], [18], [34]. The write requests are then served in batches [8] to improve the turnaround time of the data bus as serving a set of memory requests of the same type is more efficient [10]. The most common method of implementing write batching is the *watermarking technique* [8]. Specifically, when there are pending read requests, write requests can only be served by the memory controller if the number of write requests in the write buffer is greater than or equal to the *watermarking threshold*, and then at least one batch of write requests will be served by the memory controller. To avoid write buffer overflow, it is assumed that the size of the write buffer is sufficient to insert write requests from all cores, and the watermarking threshold is always kept less than the size of the write buffer. Please refer to [18], [20] for a comprehensive overview of DRAM.

III. System Model

We assume a multicore platform comprising m identical cores $(\pi_1, \pi_2, \ldots, \pi_m)$. The DRAM is shared among all the cores. Similarly to the existing work [7], we assume that the shared DRAM is accessed by cores via a set of crossbar switches that facilitates the point-to-point connection between each core and main memory. We assume that the local memory of each core (e.g., scratchpad or cache) is large enough to store all the data/instructions required by the task with the largest memory footprint that can execute on that core. If this is not possible due to the limited size of the local memory, the tasks can be divided into smaller segments [27], [28] such that the local memory is sufficient to store the data/instructions required by the task's segment with the largest memory footprint that can execute on that core. Furthermore, if the system has a shared cache, it is assumed to be partitioned between the cores such that cores have non-overlapping partitions.

A. Task Model

We consider the 3-phase task model [9], i.e., each task is divided into A, E, and R-phases, i.e., (1) an *Acquisition* or *A-phase*, during which the task prefetches it's required data/instructions from the main memory and store them in the core's local memory, (2) the *Execution* or *E-phase*, during which computation is performed on the preloaded data without generating any accesses to main memory, and (3) the *Restitution* or *R-phase* that writes back the modified data of the task to the main memory. Each phase as well as the complete task executes non-preemptively. Code compliant with the 3-phase task model can be obtained using existing tools [17], [28].

We consider a task set Γ comprising n sporadic tasks $(\tau_1, \tau_2, \ldots \tau_n)$ partitioned among cores at design time. T_i denotes the minimum inter-arrival time between two consecutive jobs of task τ_i, and D_i denotes its relative deadline. We assume that tasks have constrained deadlines, i.e., $D_i \leq T_i$. The maximum number of memory requests that can be issued during the A-phase (resp. R-phase) of task τ_i *in isolation* is denoted by MD_i^A (resp. MD_i^R). Similarly, the WCET of the E-phase of task τ_i is denoted by C_i^E, and the WCET of the A-phase (resp. R-phase) of task τ_i is denoted by C_i^A (resp. C_i^R). Note that the values of MD_i^A, MD_i^R, C_i^E, C_i^A, and C_i^R can be obtained by static analysis, measurement-based analysis, or by using the combination of both [31]. We assume that tasks are scheduled using fixed-priority non-preemptive scheduling with priorities assigned using any fixed-priority algorithm such as, but not limited to, Rate Monotonic or Deadline Monotonic [21].

Throughout the paper, we refer to the core on which task τ_i (i.e., the task under analysis) executes as the *local core*, denoted by π_l. Similarly, any core other than the local core is referred to as a *remote core*, usually denoted by π_r. The set of all tasks mapped to a remote core π_r is denoted by Γ'_r.

B. Main Memory Model

We focus on systems with DRAM as their primary memory module. We assume a single rank composed of multiple banks. We assume the DRAM follows the organization and memory request handling protocols discussed in Section II.

We formalize the properties of the considered memory controller with the following set of *rules*.

R1: Each bank has its *per-bank queue* in which memory requests targeting respective banks are queued. Each per-bank queue is sorted using the *First-Ready First-Come-First-Served* (FR-FCFS) policy which means 1) memory requests that result in a row-hit are prioritized over memory requests that result in a row-miss; and 2) in case of a tie, older memory requests are prioritized over newer memory requests.

R2: We consider that banks are partitioned between cores such that each core has its set of banks in which data it needs to *read* are stored [15], [34]. Specifically, the A-phases of tasks mapped on a core cannot *read* from banks assigned to other cores. Note that we do not assume how memory requests are mapped within the assigned set of banks.

R3: To allow data sharing between tasks assigned on different cores, tasks can *write* data in any bank.

R4: Similarly to [7], *Round-Robin* is considered as the inter-bank scheduling policy that serves memory requests from each per-bank queue with the granularity of one memory request, i.e., one memory request per turn. Furthermore, we assume that the inter-bank scheduler cannot reorder requests [7].

R5: Similarly to [7], we consider that cores have an *in-order pipeline*, i.e., there can be at most one pending *read* memory request per core at a time. A subsequent read memory request on a given core will only be issued after the previous memory request is served.

R6: We assume that read memory requests have higher priority than write memory requests since writes do not stall the processor pipeline. Write requests are enqueued in a write buffer of size Q_{write}[3] and then served in *batches* with the *watermarking mechanism* [8]. Specifically, if there are any pending read requests, the memory controller only starts serving write requests if the number of write requests enqueued in the write buffer is greater than or equal to the *watermarking threshold* W_{th} and serves at least one *batch* of write requests where the size of a batch is denoted by N_{Wb}. Similarly to [7], [34], we assume that $Q_{write} > W_{th} > Q_{write} - N_{Wb}$.

R7: For each task τ_i, we assume that $MD_i^A \geq MD_i^R$, i.e., the maximum number of read requests of the A-phase of task τ_i is greater than or equal to the maximum number of write requests of the R-phase of task τ_i, which models the fact that most pieces of code need to load more data and instruction from main memory to execute than it writes data back in main memory.

R8: It has been shown in the state-of-the-art [7], [16], [18], [34] that DRAM refreshes have a negligible effect on memory access latency, hence this work also does not consider the impact of DRAM auto-refresh on memory contention. However, if required, it can be integrated by adding a periodic cost to WCET of tasks.

R9: We assume that DRAM has n banks $(b_1, b_2, \ldots, b_n)$ where $n \geq m$. Furthermore, we assume that *consecutive banks* are assigned to cores, e.g., if y number of banks are assigned to a core π_r then the bank assignment is given by $b_x \ldots b_{x+y} \in \pi_r$.

[3]Similarly to the state-of-the-art [7], [34], we assume that the value of Q_{write} is large enough to store the write requests from all cores and the computation of Q_{write} is not in the scope of this paper.

C. Preliminaries

In this section, we will remind the reader of the results of existing work to compute the memory contention suffered by tasks. We will later leverage them to build our proposed memory contention analysis. At the DRAM bank-level, tasks can suffer two types of contentions; 1) *intra-bank contention*, i.e., due to interfering memory requests targeting the same bank as the memory request of the task under analysis; and 2) *inter-bank contention*, i.e., due to interfering memory requests targeting a different bank than the memory request of the task under analysis.

We will now briefly discuss the bounds on the inter-bank contention derived in the state-of-the-art. It has been shown in [34] that the maximum inter-bank contention that can be suffered by a memory request is given by the maximum contention it can suffer on any of the commands it comprises. Specifically, assuming that the request under analysis and interfering requests are all row miss requests, i.e., they issue PRE, ACT, and CAS commands in sequence, the maximum inter-bank contention that can be suffered by the read request under analysis from N_{rq} interfering read requests is upper bounded by (From Theorem 1 of [34])

$$L(N_{rq}) = \max_{N_{PRE}+N_{ACT}+N_{CAS}=N_{rq}} (L^{PRE}(N_{PRE}) + L^{ACT}(N_{ACT}) + L^{CAS}(N_{CAS})) \quad (1)$$

The delay bounds on each of the PRE, ACT and CAS commands is computed as per the JEDEC standard (see Table I).

Specifically, the term $L^{PRE}(N_{PRE})$ is the delay that can be caused by N_{PRE} interfering PRE commands to the PRE command of the read request under analysis and can be computed using the following equation (From Lemma 2 of [34]).

$$L^{PRE}(N_{PRE}) = 2 \cdot N_{PRE} \quad (2)$$

The main insight behind Equation 2 is that there are no inter-bank timing constraints between the PRE to PRE command. However, a PRE command can still be delayed due to the command bus contention caused by ACT and CAS commands of interfering read request (see Lemma 2 of [34] for proof).

Similarly, $L^{ACT}(N_{ACT})$ is the delay that can be caused by N_{ACT} interfering ACT commands to the ACT command of the read request under analysis and can be computed using the following equation (From Equation 9 of [18]).

$$L^{ACT}(N_{ACT}) = 2 \cdot N_{rq} + \max\left(N_{ACT} \cdot tRRD, \left\lceil \frac{N_{ACT}+1}{4} \cdot tFAW \right\rceil\right) \quad (3)$$

The main insight behind Equation 3 is that there is a tRRD inter-bank timing constraint between ACT to ACT commands. Furthermore, tFAW inter-bank timing constraint represents that no more than 4 ACT can be issued on different banks of the same rank. The term $2 \cdot N_{rq}$ represents the command bus contention that can be suffered by the ACT command of read request under analysis from CAS and PRE commands of interfering read request.

Finally, $L^{CAS}(N_{CAS})$ is the delay that can be caused by N_{CAS} interfering CAS commands to the CAS command of the read request under analysis and can be computed using the following equation (From Equations 10 and 4 of [18]).

$$L^{CAS}(N_{CAS}) = (N_{CAS} + 1) \cdot tCCD + 2 \cdot N_{rq} \quad (4)$$

Equation 4 is also derived by considering the CAS to CAS inter-bank timing constraints and command bus contention.

Now, we will discuss the upper bound on the maximum memory contention that can be suffered by the read request under analysis due to interfering *write requests*.

(From Equation 2 of [18]) The maximum delay that can be suffered by the read request under analysis due to N *write requests* when the memory controller uses *write batching* is upper bounded by L_{WB}, where

$$L_{WB}(N) = N \cdot (\max(tRAS, tRCD + tWL + tB + tWR) + tRP) \quad (5)$$

The value of L_{WB} is derived using the JEDEC timing constraints by considering 1) N write requests will be served by the memory controller in batches; 2) the write requests can access any bank in the system; and 3) each write request is a row-miss request that issues PRE, ACT, and CAS commands sequence. The detailed derivation of all the equations discussed in this section can be found in [18], [34].

IV. Proposed Memory Contention Analysis

In this section, we present our proposed memory contention analysis. We start by computing the maximum memory contention that can be suffered by *read requests* of the A-phase of task τ_i due to *read requests* of tasks running on all *remote cores*. We now introduce important properties that will be useful in deriving the bound on the memory contention.

Property IV.1. Read requests of the task under analysis cannot suffer intra-bank contention from read requests of tasks running on remote cores.

Proof. Since banks are partitioned between cores for *read accesses* (see Rule R2), a read request of tasks running on any remote core will never access the same bank as the read request of the task under analysis executing on the local core π_l. Thus, read requests of the task under analysis cannot suffer intra-bank contention from read requests of tasks running on remote cores. □

Property IV.2. There can only be at most one pending read request in a per-bank queue at a time.

Proof. A core can only have one pending read request at a time (see Rule R5). Furthermore, banks are partitioned between cores for read accesses (see Rule R2). Therefore, there can only be at most one pending read request in a per-bank queue at a time. □

As proven in Property IV.1, read requests of the task under analysis cannot suffer intra-bank contention from read requests of tasks running on remote cores. Therefore, we will now compute the maximum *inter-bank contention* that can be suffered by *read requests* of the A-phase of task τ_i due to *read requests* of tasks running on all *remote cores*.

Lemma 1. *The maximum number of read memory requests of tasks running on all remote cores that can interfere with read memory requests of the A-phase of task τ_i is upper bounded by N_i^{read}, where*

$$N_i^{read} = MD_i^A \times (m - 1) \quad (6)$$

Proof. Due to the RR inter-bank scheduler, the read request of the A-phase of task τ_i may have to wait for the completion of one read request from each per-bank queue assigned to remote cores. However, this can be pessimistic because a consecutive set of banks is assigned to each core (see Rule 9). In such a scenario, if y banks are assigned to a remote core π_r, the bank assignment is given by $b_x, \ldots, b_{x+y} \in \pi_r$. Consequently, when the memory controller serves a read request from the per-bank queue of bank b_x, the per-bank queues of banks $b_{x+1}, \ldots, b_{x+y}$ will be empty as there can be at most one pending read request per core (see Rule 5). Thus, after serving the memory request of b_x bank, the RR algorithm of the inter-bank scheduler will directly jump to bank b_{x+y+1}. This implies that a read request of the A-phase of task τ_i can be delayed due to read requests enqueued in at most one per-bank queue assigned to each remote core and there can be at most one pending read request in a per-bank queue at a time (see Property IV.2). Thus, a read request of the A-phase of task τ_i can be delayed due to $m-1$ read requests of tasks on all remote cores. Extending this to each read request of the A-phase of task τ_i, $MD_i^A \times (m-1)$ upper bounds the maximum number of read memory requests of tasks running on all remote cores that can interfere with read memory requests of the A-phase of task τ_i. The Lemma follows. □

Having bounded the number of interfering read requests, we bound the *maximum contention* that can be caused by those interfering requests to read requests of the A-phase of task τ_i.

Lemma 2. *The maximum memory contention that can be suffered by read requests of the A-phase of task τ_i due to read requests of tasks running on all remote cores is upper bounded by MC_i^{read}, where*

$$MC_i^{read} = MD_i^A \times \max_{N_{PRE}+N_{ACT}+N_{CAS}=m-1} (L^{PRE}(N_{PRE}) + L^{ACT}(N_{ACT}) + L^{CAS}(N_{CAS})) \quad (7)$$

Proof. From Lemma 1, we know that each read request of the task under analysis can be delayed by at most $m-1$ read requests issued by all remote cores. As memory requests can potentially be mapped to any rows/columns of the set of banks assigned to that core, in the worst case, each memory request can be a row-miss. As a consequence, all memory requests issue PRE, ACT, and CAS commands in sequence. From Equation 1 proven in Theorem 1 of [34], we know that $\max_{N_{PRE}+N_{ACT}+N_{CAS}=m-1} \big(L^{PRE}(N_{PRE}) + L^{ACT}(N_{ACT}) + L^{CAS}(N_{CAS})\big)$ bounds the maximum memory contention that can be generated by $m-1$ read requests. Hence, every read request of τ_i suffers at most

$\max_{N_{PRE}+N_{ACT}+N_{CAS}=m-1}\big(L^{PRE}(N_{PRE})+L^{ACT}(N_{ACT})$ $+L^{CAS}(N_{CAS})\big)$ time units of interference. The MD_i^A read requests of τ_i's A-phase will thus suffer at most $MD_i^A \times \max_{N_{PRE}+N_{ACT}+N_{CAS}=m-1}\big(L^{PRE}(N_{PRE})+L^{ACT}(N_{ACT}) + L^{CAS}(N_{CAS})\big)$ time units of interference. □

Having bounded the contention caused by read requests, the next step is to compute the maximum contention that can be caused by *write requests* of any core to read requests of the A-phase of task τ_i. We start by briefly discussing how such a bound is derived in [7] and identify *sources of pessimism*. We then propose a new bound in Lemmas 3 and 4.

From Lemma 3 of [7] *The overall interference suffered by read requests of the A-phase of task τ_i due to write requests in any time interval of length t is bounded by*

$$MC_i^{wr}(t) = L_{WB}(\min(NR(t) \times N_{wb}, NW(t) + Q_{write})) \quad (8)$$

where $L_{WB}(N)$ is the maximum delay that can be caused by N write requests (see Equation 5); $NR(t)$ is the total number of read requests that can be issued by the A-phase of task τ_i plus all interfering read requests from all remote cores during a time interval of length t; and $NW(t)$ is the maximum number of write requests that can be issued by all jobs of all tasks running on all remote cores during a time interval of length t.

In Equation 8, $NR(t) \times N_{wb}$ captures that each read request of the task τ_i as well as each interfering read request can be delayed by one batch of write requests. This is a pessimistic bound since it assumes that every read request will suffer from one batch of write requests without analyzing the maximum number of batches that can be triggered during the execution of the A-phase of τ_i. Similarly, the $NW(t)$ captures write requests of all jobs of all tasks released on all remote cores during a time interval of length t. This bound is also pessimistic as it assumes all writes that can be issued by all jobs released on all remote cores during a time interval of length t can interfere. However, in the 3-phase model, a core issues an R-phase only after the completion of the E-phase which further depends on the completion of the A-phase. This implies that even when several jobs are released on a core, the actual number of write requests (R-phases) issued by a core depends on the number of read requests (A-phases) served on that core. Building on this, we will now accurately bound the maximum number of *write batches* that can be served by the memory controller during the execution of the A-phase of τ_i.

Lemma 3. *The maximum number of batches of write memory requests that can be served by the memory controller during the execution of the A-phase of τ_i is upper bounded by N_i^{wb}, where N_i^{wb} is given by*

$$1 + \left\lceil \frac{\sum_{r=1,r\neq l}^{m} \max_{\tau_u \in \Gamma'_r} \{MD_u^R\} + N_i^{read} - (W_{th} - (Q_{write} - N_{wb}))}{N_{wb}} \right\rceil \quad (9)$$

Proof. When the first read request of the A-phase of task τ_i arrives at the memory controller, in the worst case, the number of write requests inserted in the write buffer is equal to the size of the write buffer Q_{write}. This will trigger one batch of write requests as accounted by the "+1" in Equation 9. After the first write batch, the maximum number of write requests still left in the write buffer is equal to $Q_{write} - N_{wb}$. Now, there can be a scenario in which a remote core just completed an E-phase and starts executing an R-phase. Thus, we need to account for write requests that can be issued by at least one R-phase on each remote core. In the worst case, each remote core executes the R-phase that issues the largest number of write requests among the R-phases of all tasks running on that remote core, i.e., $\max_{\tau_u \in \Gamma'_r}\{MD_u^R\}$. Extending this to all remote cores, $\sum_{r=1,r\neq l}^{m} \max_{\tau_u \in \Gamma'_r}\{MD_u^R\}$ upper bounds the number of write requests that can be issued by the R-phases of tasks that already completed their A-phases prior to the arrival of the A-phase of task τ_i. Note that to produce another R-phase on the same remote core, the core first needs to execute an A-phase.

From Lemma 1, we know that N_i^{read} bounds the maximum number of interfering read requests. Since the length of the R-phases of tasks is assumed to be less than or equal to their A-phases (see Rule R7), in the worst case, there can be at most N_i^{read} number of write requests that can be issued by all remote cores in the same interval the A-phase of task τ_i executed. We do not need to account for write requests issued on the local core because 1) task τ_i will only issue its R-phase after the completion of its A-phase; and 2) the R-phase of any other previously executed task on the local core must have already inserted all its write requests in the write buffer before the start of τ_i. Therefore, we have at most $Q_{write}-N_{wb}+\sum_{r=1,r\neq l}^{m} \max_{\tau_u \in \Gamma'_r}\{MD_u^R\}+N_i^{read}$ write requests enqueued in the write buffer after the first write batch. Now, there must be at least W_{th} requests in the write buffer to trigger a write batch and each batch has a size of N_{wb} requests, so the total number of write batches triggered by $\sum_{r=1,r\neq l}^{m} \max_{\tau_u \in \Gamma'_r}\{MD_u^R\}+N_i^{read}-(W_{th}-(Q_{write}-N_{wb}))$ write requests is upper bounded by $\left\lceil \frac{\sum_{r=1,r\neq l}^{m} \max_{\tau_u \in \Gamma'_r}\{MD_u^R\}+N_i^{read}-(W_{th}-(Q_{write}-N_{wb}))}{N_{wb}} \right\rceil$. □

From Lemma 3, we can simply compute the maximum *number* of write requests that can interfere with the A-phase of task τ_i using N_i^{write}, where

$$N_i^{write} = N_i^{wb} \times N_{wb} \quad (10)$$

Lemma 4. *The maximum memory contention that can be suffered by the A-phase of task τ_i due to write requests is upper bounded by MC_i^{write}, where*

$$MC_i^{write} = L_{WB}(N_i^{write}) \quad (11)$$

Proof. From Equation 10, N_i^{write} upper bounds the maximum number of interfering write requests; and from Equation 5 that is proven in Equation 2 of [18] (also reused in Lemma 3 of [7]), we know that $L_{WB}(N_i^{write})$ upper bounds the memory contention caused by N_i^{write} requests. The Lemma follows. □

Finally, using Equations 7 and 11, the *total memory contention* that can be suffered by the A-phase of task τ_i is upper bounded by MC_i^{total}, where

$$MC_i^{total} = MC_i^{read} + MC_i^{write} \tag{12}$$

As proven in [7], we do not need to account for memory contention that can be suffered by the R-phase of task τ_i. This is mainly because 1) writes do not stall the processor pipeline; 2) the write buffer of size Q_{write} is assumed to be large enough to insert write requests from all cores; and 3) the write batching mechanism ensures that a batch of writes is served as soon as the write batching threshold W_{th} is reached, where $Q_{write} > W_{th} > Q_{write} - N_{Wb}$. Furthermore, Lemma 3 also takes into account the impact of the R-phases of the previously executed tasks on the A-phase of the task under analysis on the same core. Hence, we only need to account for the WCET in isolation of the R-phase.

Having bounded the maximum memory contention that can be suffered by task τ_i, we can integrate the maximum memory contention into the WCET of task τ_i. Similar to [7], integrating memory contention into the WCET of tasks allows computing an *inflated WCET*.

Let $\hat{C}_i$ denote the inflated WCET of task τ_i which also takes into account the memory contention that can be suffered by task τ_i, where $\hat{C}_i$ is given by:

$$\hat{C}_i = MC_i^{total} + C_i \tag{13}$$

where MC_i^{total} is given by Equation 12; and C_i is the WCET of task τ_i in isolation and is computed by $C_i = C_i^A + C_i^E + C_i^R$.

After inflating the WCET of each task in the taskset using Equation 13, the WCRT of each task can be computed using the well-known WCRT analysis for single-core processors [6] as we have already accounted for the maximum memory contention that can be suffered by each task into its inflated WCET.[4]

V. Experimental Evaluation

In this section, we will evaluate the performance of the proposed memory contention analysis in comparison to the state-of-the-art. As we improve upon the work of [7], we need to directly compare the proposed analysis with [7]. However, comparing the proposed analysis with the exact analysis of [7] can be biased. This is mainly because the analysis of [7] does not consider bank partitioning. This implies that tasks suffer both intra-bank and inter-bank contention. As a consequence, the analysis of [7] may tend to perform poorly in comparison to the proposed analysis not necessarily due to the pessimism in the existing analysis but due to fundamental differences in the assumptions. Therefore, we also apply bank partitioning using Rule R2 when computing memory contention bounds for the analysis in [7]. It implies that for the existing work [7], we modeled the contention suffered by *read requests* of the task under analysis from *interfering read requests* identically to that of the proposed work, i.e., using Lemmas 1 and 2. Furthermore, for the existing work [7], we compute the memory contention suffered by *read requests* of the task under analysis from *write requests* using Equation 8. On the other hand, for the proposed work, we compute the maximum number of interfering write requests using Equation 10 and the maximum contention caused by all those interfering write requests using Equation 11. Note that Equations 8 and 11 leverages Equation 5 (proven in Equation 2 of [18]). Using the maximum contention from read and write requests, we compute the *total memory contention* suffered by tasks and then inflate their respective WCETs by integrating the total memory contention as shown in Equation 13.

For the experimental evaluation, we perform a set of experiments using synthetic tasksets to compare the performance of the proposed analysis with the existing analysis [7]. For the default configuration, we model a quad-core platform with the taskset size of 32 tasks in which 8 tasks were randomly mapped to each core. Tasks utilization U_i was generated using the UUnifast-discard algorithm [11]. Task periods T_i were randomly generated in the range of $[10^6, 10^7]$ using log-uniform distribution. The WCET in isolation C_i was then assigned by $C_i = U_i \times T_i$. The total memory access demand (MD) of tasks was derived using C_i such that, $MD_i = rand(10\%, 20\%) \times C_i$. The memory demand of the A-phase A_i was chosen randomly in the range [50%, 90%] of MD_i, i.e., $A_i = rand(50\%, 90\%) \times MD_i$. Similarly, the memory demand of the R-phase was then chosen by $MD_i - A_i$. We use a term t^{miss} to represent the maximum time taken by a memory request, where the value of t^{miss} can be computed by considering the intra-bank timing constraints given in Table I. Then we generated the maximum number of memory requests that can be issued during the A-phase (resp. R-phase) using $MD_i^A = \lceil \frac{A_i}{t^{miss}} \rceil$ (resp. $MD_i^R = \lceil \frac{MD_i - A_i}{t^{miss}} \rceil$). Task deadlines were equal to task periods, i.e., $D_i = T_i$.

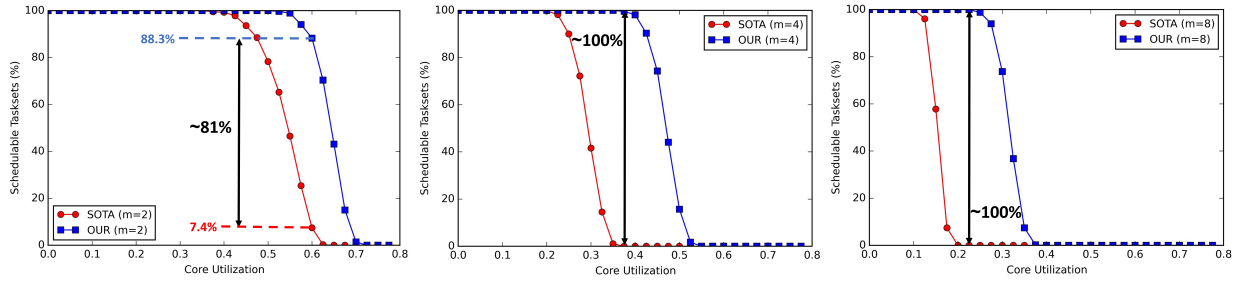
In all the experiments, for the memory controller, we consider the following parameters.

$Q_{write} = 64$ (identically to [7], [34]); $W_{th} = 54$ (identically to [34]); and $N_{wb} = 18$ (identically to [7], [34]).

We perform various experiments to compare the performance of the proposed memory contention analysis with the existing memory contention analysis [7] by varying: 1) the core utilization; 2) number of cores; 3) memory access demand; and 4) task periods. We use taskset schedulability, i.e., the percentage of schedulable tasksets, as a metric to evaluate the performance of each approach. To plot each point in every experiment, 1000 task sets were randomly generated. In all the experiments, the proposed memory contention analysis is marked as "**OUR**" and the state-of-the-art (SOTA) analysis [7] is marked as "**SOTA**". In all the plots, the x-axis represents the core utilization and the y-axis represents the percentage of schedulable tasksets, i.e., the percentage of tasksets that were deemed schedulable.

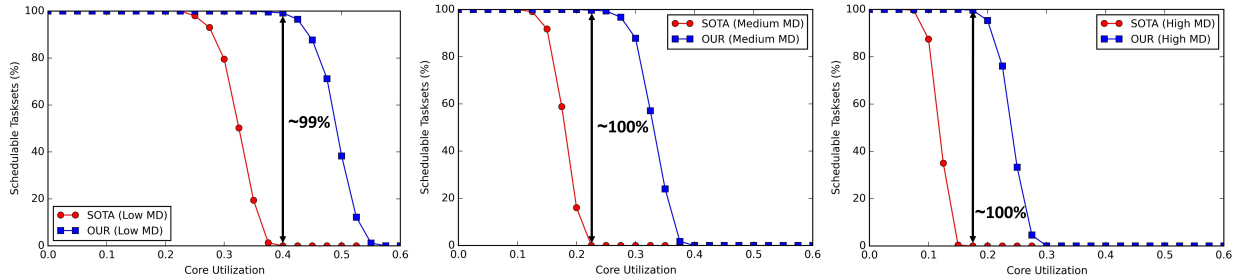
1. Core Utilization: In this experiment, we varied each core utilization between 0.05 and 1 in steps of 0.025 for the default configuration, i.e., $m = 4$. We then plotted the percentage

[4]Note that any contention on the memory controller results in busy waiting on the core. Therefore, there are no anomalies caused by any kind of self-suspension.



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

Fig. 1: Varying the Core Utilization and Number of Cores



(a) Low MD (b) Medium MD (c) High MD

Fig. 2: Varying the Memory Demand (MD)

of tasksets that were deemed schedulable by the proposed analysis and SOTA analysis for each core utilization value in Figure 1b. We can see in Figure 1b that increasing the core utilization decreases taskset schedulability for both approaches. This happens because the WCET C_i was generated using U_i and T_i, i.e., $C_i = U_i \times T_i$. This implies that an increase in core utilization increases tasks utilization which further increases the WCET in isolation and memory demand of tasks. This further results in an increase in the interference/blocking caused by tasks on the local core and the memory contention from remote cores. Consequently, the taskset schedulability decreases with the increase in core utilization. As shown in Figure 1b, the taskset schedulability is low for both approaches, e.g., no tasksets were deemed schedulable after 55% core utilization. This happens because the default configuration considers $m = 4$ and 32 tasks in the taskset so tasks can suffer significant memory contention which increases tasks' WCET/WCRT and decreases taskset schedulability. Nonetheless, the proposed approach outperformed the SOTA analysis by improving the taskset schedulability up to 100% points. This gain is mainly observed as tasks suffer a significant amount of memory contention in the default configuration, i.e., $m = 4$, and since the proposed approach tightly bounds the total memory contention, i.e., by accurately quantifying the maximum number of write requests and the contention caused by those write requests, the proposed approach significantly outperformed the SOTA analysis.

2. Number of Cores: In this experiment, we vary the number of cores along with the core utilization. The number of cores (m) were varied from 2 to 8 along with core utilization. The taskset schedulability using both approaches on different values of m is plotted in Figure 1. We observed that increasing (resp. decreasing) the number of cores negatively (resp. positively) impacted the taskset schedulability. This is mainly because increasing the number of cores also increases the number of interfering tasks which results in an increase in the number of interfering memory requests. This in turn increases the total memory contention suffered by tasks and decreases taskset schedulability. Nonetheless, the proposed analyses outperformed the SOTA analysis for all the considered values of m. We can also see in Figure 1 that the gain of the proposed approach over SOTA decreases with the decrease in the value of m. For example, as shown in Figure 1a, the proposed approach was able to schedule 81% of more tasksets in comparison to the SOTA at the core utilization of 0.60. This happens because when $m = 2$, the impact of memory contention is lesser as there is only one remote core with 8 tasks. Since the proposed approach mainly improves the bound on memory contention, the gain of the proposed approach over the SOTA analysis was reduced for $m = 2$.

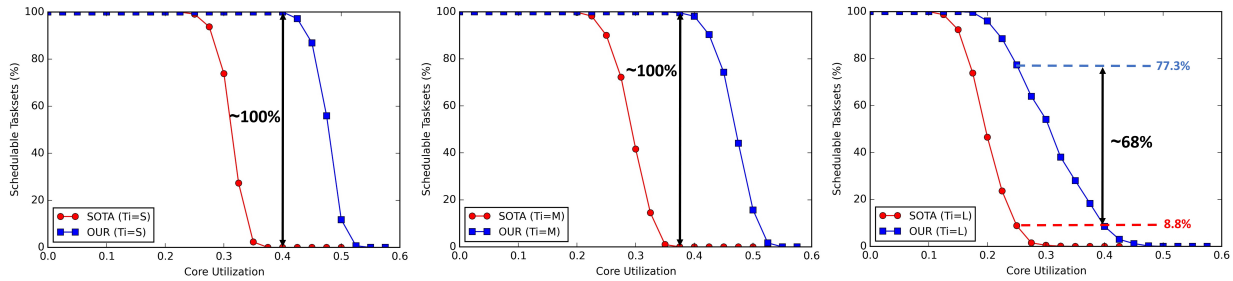
3. Memory Access Demands: In this experiment, we vary the Memory Access Demand (MD) of tasks by considering three different configurations as follows:

(a) Low MD, i.e., MD=$(5\%, 20\%) \times C_i$,

(b) Medium MD, i.e., MD=$(20\%, 40\%) \times C_i$,

(c) High MD, i.e., MD= $(40\%, 60\%) \times C_i$.

The value of MD was assigned to each task in the taskset randomly as per the chosen configuration. The percentage of tasksets that were deemed schedulable using both approaches for each MD configuration is plotted in Figure 2. We observe that an increase (resp. decrease) in the MD value negatively (resp. positively) impacts the taskset schedulability.



(a) Short (S) Task Period Range (b) Medium (M) Task Period Range (c) Long (L) Task Period Range

Fig. 3: Varying the Task Period Range

This happens because an increase in the MD value translates to an increase in the number of memory requests which also increases the number of interfering requests and total memory contention that tasks can suffer. As a consequence, both approaches performed the best under the Low MD configuration and the worst under the High MD configuration. However, for all the MD configurations, the proposed analysis performed significantly better than the SOTA analysis.

4. Task Periods: In this experiment, we varied the period range of tasks and analyzed its impact on taskset schedulability. As the WCET C_i of tasks were generated using the task periods, i.e., $C_i = U_i \times T_i$, the task periods can impact the WCET C_i of tasks. This can further impact the length of the memory phases as the memory demand of tasks is generated using the value of C_i. For this experiment, we consider three different task periods range configurations as follows:
(a) S (Short) range, i.e., $T_i = [10^6, 5 \times 10^6]$,
(b) M (Medium) range, i.e., $T_i = [10^6, 10^7]$,
(c) L (Long) range, i.e., $T_i = [10^6, 5 \times 10^7]$.

For each task period range, the task periods were generated using log-uniform distribution. The percentage of schedulable tasksets using both approaches for each task period range is plotted in Figure 3. In Figure 3, we observe that an increase in the period range has a negative impact on taskset schedulability. This mainly happens because increasing the task period increases the WCET of tasks due to the relation between C_i and T_i, i.e., $C_i = U_i \times T_i$. This in turn increases the blocking from a lower priority task on the same core, i.e., a larger period leads to a larger WCET which can cause a larger blocking from lower priority tasks. This increase in lower priority blocking also increases the WCRT of tasks. This causes a degradation in taskset schedulability when the period ranges increase. This is also the reason that for the L range configuration (see Figure 3c), the gain of the proposed approach over SOTA was reduced to around 68% since the lower priority blocking significantly impacted taskset schedulability and its computation is identical for both approaches. Nonetheless, for all the task period range configurations, the proposed analysis outperformed the SOTA analysis.

VI. Related Work

The main memory contention in multicore systems is a well-known problem identified by the real-time systems research community. A plethora of works in the literature built different solutions for the memory contention problem (see surveys [22], [24]). Among other solutions, the concept of the phased execution model, e.g., the 3-phase task model [9], [26], has been identified as suitable for the memory contention problem. The main idea of these models is to divide the execution of each task into distinct *execution phase*, i.e., computation only without any main memory accesses, and *memory phases*, i.e., all the main memory accesses. Initial works leverage the 3-phase task model to build a contention-free time-triggered schedule [5], [25] that ensures that at most one memory phase accesses the main memory at a time. Even though these solutions are important, they cannot be applicable in scenarios in which tasks are of event triggered/sporadic nature. To address this issue, several existing works analyze the maximum memory/bus contention that can be suffered by 3-phase tasks considering partitioned fixed-priority scheduling [2]–[4], [13], [14] and global fixed-priority scheduling [23], [30]. The main goal of these approaches is to quantify the maximum memory/bus contention that can be suffered by 3-phase tasks and integrate the respective bounds into their WCET and WCRT. However, these approaches model the main memory as a *black box*, i.e., they assume main memory and memory bus as a single resource without taking into account the memory organization, type of memory requests, locality of memory requests, and memory controller arbitration protocols. Consequently, the bounds derived on the memory contention using these approaches can be pessimistic/optimistic as modern memory controllers can potentially reorder memory requests depending on the locality of the memory request, type of memory requests, memory controller arbitration policies, etc.

To fill this gap, several works build memory contention analysis using *white-box memory modeling* [1], [7], [10], [16], [18], [20], [34]. The main idea of these approaches is to take into account the memory organization, low-level arbitration mechanism of the memory controller, type of memory requests, locality of memory requests, precise bound on memory access time of each memory request, etc., while deriving the upper bound on memory contention. Among all these approaches, the only work that focused on the memory contention analysis for the 3-phase task model is proposed in [7]. The work in [7] built the memory contention analysis for 3-phase tasks considering partitioned fixed-priority non-preemptive scheduling. Their work proposed an ILP formu-

lation to bound the maximum memory contention that can be caused by read requests. However, as discussed earlier, their analysis derives a pessimistic bound on memory contention caused by write requests. Therefore, the proposed work improves the bound on memory contention by tightly bounding memory contention caused by write requests, i.e., by closely analyzing the relationship between interfering A-phases (reads) and interfering R-phases (writes).

VII. Conclusion and Future Work

In this work, we proposed an improved memory contention analysis for the 3-phase task model considering partitioned fixed-priority scheduling. First, we identify the sources of pessimism in the existing memory contention analysis [7] that focuses on the 3-phase task model. We then show how this pessimism can be tackled by providing a more precise estimate of memory contention that can be caused by write requests by considering the relationship between interfering read requests and interfering write requests. Experimental evaluations show that our proposed memory contention analysis was able to derive tighter bounds on the memory contention of tasks which can improve task set schedulability by (a) up to 100% percentage points in scenarios in which memory contention is significant, and (b) around 68% to 81% percentage points in scenarios in which the memory contention is lower. In future works, we aim to improve the proposed analysis by taking into account different memory mapping schemes, i.e., how memory requests are mapped to DRAM banks. This can potentially allow us to provide a tighter bound of memory access times of each request, e.g., there can be some row hit requests if memory requests are mapped to the same row of the bank.

Acknowledgments

This work is supported by the European Union/Next Generation EU, through the Recovery and Resilience Plan (PRR) [Project Route 25 with Nr. C645463824-00000063]. This work was also supported by the CISTER Research Unit (UIDP/UIDB/04234/2020), financed by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology).

References

[1] S. Abdelhalim et al. A Tight Holistic Memory Latency Bound Through Coordinated Management of Memory Resources. In *35th ECRTS*, volume 262 of *LIPIcs*, pages 17:1–17:25, 2023.

[2] J. Arora et al. Bus-contention aware schedulability analysis for the 3-phase task model with partitioned scheduling. In *29th RTNS*, RTNS'2021, page 123–133. ACM, 2021.

[3] J. Arora et al. Analyzing fixed task priority based memory centric scheduler for the 3-phase task model. In *28th IEEE RTCSA*, pages 51–60, 2022.

[4] J. Arora et al. Improved bus contention analysis for 3-phase tasks. In *29th IEEE RTCSA*, pages 243–252, 2023.

[5] M. Becker et al. Contention-free execution of automotive applications on a clustered many-core platform. In *ECRTS 2016*, pages 14–24, 2016.

[6] R. J. Bril et al. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *ECRTS'07*, pages 269–279, 2007.

[7] D. Casini et al. A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling. In *2020 IEEE RTAS*, pages 239–252, 2020.

[8] N. Chatterjee et al. Staged reads: Mitigating the impact of dram writes on dram reads. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, 2012.

[9] G. Durrieu et al. Predictable Flight Management System Implementation on a Multicore Processor. In *ERTS'14*, TOULOUSE, France, 2014.

[10] L. Ecco and R. Ernst. Tackling the bus turnaround overhead in real-time sdram controllers. *IEEE Transactions on Computers*, 66(11):1961–1974, 2017.

[11] P. Emberson et al. Techniques for the synthesis of multiprocessor tasksets. *WATERS'10*, 01 2010.

[12] G. Schwäricke et al. Fixed-Priority Memory-Centric Scheduler for COTS-Based Multiprocessors. In *32nd ECRTS*, volume 165 of *LIPIcs*, pages 1:1–1:24, 2020.

[13] J. Arora et al. Bus-contention aware wcrt analysis for the 3-phase task model considering a work-conserving bus arbitration scheme. *Journal of Systems Architecture*, 122:102345, 2022.

[14] J. Arora et al. Schedulability analysis for 3-phase tasks with partitioned fixed-priority scheduling. *Journal of Systems Architecture*, 131:102706, 2022.

[15] L. Liu et al. A software memory partition approach for eliminating bank-level interference in multicore systems. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 367–375, 2012.

[16] M. Hassan et al. Analysis of Memory-Contention in Heterogeneous COTS MPSoCs. In *ECRTS 2020*, volume 165 of *LIPIcs*, pages 23:1–23:24, Dagstuhl, Germany, 2020.

[17] F. Fort and J. Forget. Code generation for multi-phase tasks on a multi-core distributed memory platform. In *25th IEEE*, pages 1–6, 2019.

[18] M. Hassan et al. Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2323–2336, 2018.

[19] JEDEC. Ddr3 sdram standard jesd79-3b, 2008, 2008. Accessed: 2023-03-03.

[20] H. Kim et al. Bounding memory interference delay in cots-based multi-core systems. In *19th IEEE RTAS*, pages 145–154, 2014.

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

[22] T. Lugo et al. A survey of techniques for reducing interference in real-time applications on multicore platforms. *IEEE Access*, 10:21853–21882, 2022.

[23] C. Maia et al. Schedulability analysis for global fixed-priority scheduling of the 3-phase task model. In *23rd IEEE RTCSA*, pages 1–10, Hsinchu, Taiwan, August 2017. IEEE.

[24] C. Maiza et al. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Computing Surveys*, 52(3):1–38, June 2019.

[25] C. Pagetti et al. Automated generation of time-predictable executables on multicore. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, RTNS '18, page 104–113. ACM, 2018.

[26] R. Pellizzoni et al. A Predictable Execution Model for COTS-Based Embedded Systems. In *17th IEEE RTAS*, pages 269–279, Chicago, IL, USA, April 2011. IEEE.

[27] M. R. Soliman and R. Pellizzoni. PREM-Based Optimal Task Segmentation Under Fixed Priority Scheduling. In *31st ECRTS*, volume 133 of *LIPIcs*, pages 4:1–4:23, 2019.

[28] M. R. Soliman et al. Segment streaming for the three-phase execution model: Design and implementation. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 260–273, 2019.

[29] R. Tabish et al. A real-time scratchpad-centric os with predictable inter/intra-core communication for multi-core embedded systems. *Real-Time Systems*, 55, 10 2019.

[30] T. Thilakasiri and M. Becker. An exact schedulability analysis for global fixed-priority scheduling of the aer task model. In *Proceedings of the 28th ASPDAC*, page 326–332, New York, NY, USA, 2023. ACM.

[31] R Wilhelm et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, April 2008.

[32] Z. Wu et al. A composable worst case latency analysis for multi-rank dram devices under open row policy. *Real-Time Systems*, 52, 11 2016.

[33] G. Yao et al. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48, 11 2012.

[34] H. Yun et al. Parallelism-aware memory interference delay analysis for cots multicore systems. In *2015 27th ECRTS*, pages 184–195, 2015.