Conference Paper

Improved Bus Contention Analysis for 3-Phase Tasks

The paper is accepted as a full paper in RTCSA 2023.

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

*CISTER Research Centre
CISTER-TR-230505

2023/08/30
Improved Bus Contention Analysis for 3-Phase Tasks

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

The 3-phase task execution model has shown to be a good candidate to tackle the memory bus contention problem. It divides the execution of tasks into computation and memory phases that enable a fine-grained memory bus contention analysis. However, existing works that focus on the bus contention analysis for 3-phase tasks, neglect the fact that memory bus contention strongly relates to the number of bus/memory requests generated by tasks, which, in turn, depends on the content of the cache memories during the execution of those tasks. These existing works assume that the worst-case number of bus/memory requests will be generated during all the memory phases of all tasks, irrespective of the already existing content in the cache memory. This overestimates the memory bus contention of tasks, leading to pessimistic worst-case response time (WCRT) bounds.

This work proposes a holistic approach towards bus contention analysis for 3-phase tasks by (1) deriving an upper bound on the actual cache misses of tasks that lead to bus/memory requests; (2) improving State-of-the-Art (SoA) bus contention analysis of two bus arbitration schemes that dominate all existing works on the bus contention analysis for 3-phase tasks; and (3) performing an extensive experimental evaluation under different settings to compare the proposed analysis against the SoA. Results show that incorporating a tighter bound on the number of cache misses of tasks into the bus contention analysis can lead to a significant improvement in task set schedulability.
Improved Bus Contention Analysis for 3-Phase Tasks

Jatin Arora†, Syed Aftab Rashid‡§, Geoffrey Nelissen‡, Cláudio Maia‡, Eduardo Tovar‡
†CISTER, ISEP, Porto, Portugal ‡VORTEX CoLab, Portugal †Eindhoven University of Technology, Eindhoven, the Netherlands

Abstract—The 3-phase task execution model has shown to be a good candidate to tackle the memory bus contention problem. It divides the execution of tasks into computation and memory phases that enable a fine-grained memory bus contention analysis. However, existing works that focus on the bus contention analysis for 3-phase tasks, neglect the fact that memory bus contention strongly relates to the number of bus/memory requests generated by tasks, which, in turn, depends on the content of the cache memories during the execution of those tasks. These existing works assume that the worst-case number of bus/memory requests will be generated during all the memory phases of all tasks, irrespective of the already existing content in the cache memory. This overestimates the memory bus contention of tasks, leading to pessimistic worst-case response time (WCRT) bounds.

This work proposes a holistic approach towards bus contention analysis for 3-phase tasks by (1) deriving an upper bound on the actual cache misses of tasks that lead to bus/memory requests; (2) improving State-of-the-Art (SOTA) bus contention analysis of two bus arbitration schemes that dominate all existing works on the bus contention analysis for 3-phase tasks; and (3) performing an extensive experimental evaluation under different settings to compare the proposed analysis against the SOTA. Results show that incorporating a tighter bound on the number of cache misses of tasks into the bus contention analysis can lead to a significant improvement in the task set schedulability.

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 that hinders the use of commercial off-the-shelf (COTS) multicore platforms in hard real-time systems is their unpredictability, which originates from the sharing of different hardware resources, e.g., Last-Level Cache (LLC), the interconnect (e.g., memory bus), and the main memory. A task executing on one core of a multicore platform has to compete with other co-running tasks (running on other cores) to access these shared resources. For instance, the shared memory bus connects all the cores to the main memory. Due to such sharing, tasks running on different cores may have to compete to access the memory bus in order to read/write data/instructions to/from the main memory, resulting in bus contention. This bus contention can significantly increase the Worst-Case Execution Time (WCET) and Worst-Case Response Time (WCRT) of tasks.

The 3-phase task model [3], [19] has been extensively used by the state-of-the-art to tackle the memory/bus contention problem [1], [5], [8], [9], [16], [18]. The 3-phase task model divides the execution of each task into distinct computation and memory phases such that a task can only access the shared memory bus/main memory during its memory phase and the core execute the task during the computation phase without accessing the shared bus/main memory. Although the 3-phase task model makes bus/memory access patterns of tasks more predictable, 3-phase tasks can still suffer bus/memory contention, e.g., when multiple concurrent tasks running on different cores try to access the bus/memory. Considering the impact bus contention can have on the WCET/WCRT of tasks, several works in the state-of-the-art [1], [8], [9], [16], [25] have focused on analyzing the maximum bus contention that can be suffered by 3-phase tasks and its impact on taskset schedulability. In a few recent works, Arora et al. [8], [9] have presented bus contention analysis for 3-phase tasks that dominate all the existing bus contention analysis for 3-phase tasks. However, when computing bus contention, these works [8], [9] assume that the number of bus/memory requests that can be generated during the memory phases of each job of a task is always equal to its worst-case memory access demand in isolation. Although this assumption is safe, it can lead to pessimistic results, especially for platforms with cache memories. Caches are smaller faster memories that store the recently referenced data/instructions of tasks and allow for data re-use, i.e., cache content fetched during the execution of one job of a task may be re-used during the execution of a subsequent job of the same task [20]. This can significantly reduce the number of bus/memory requests generated during the execution of subsequent jobs of tasks. However, this assumption is not considered by [8], [9] and hence these works result in overestimating the bus contention of tasks.

In this work, we exploit the interdependence between the cache memory, bus requests and bus contention, i.e., the bus contention suffered by tasks depends on the number of bus requests which in turn depends on the number of cache misses, to improve the analysis presented in [8], [9]. First, we analyze the cache to tightly upper bound the number of cache misses that lead to bus/memory requests during the memory phases. We then improve the existing bus contention analyses [8], [9] by incorporating a tighter bound on the number of cache misses/bus requests into the analysis. Finally, we propose a WCRT-based schedulability analysis by integrating the bounds on bus contention into the WCRT of tasks. Formally, our main contributions are:
(1) Upper bounding cache misses of 3-phase tasks to compute bus/memory requests generated during the memory phases of tasks;
(2) Using the derived bound on cache misses, we improve the existing memory bus contention analyses for 3-phase tasks considering fixed-priority partitioned scheduling [8], [9];
(3) Integrating the bounds on bus contention into a WCRT formulation to perform schedulability analysis; and
(4) Extensive empirical evaluation under different settings to compare our proposed improved bus contention analyses to the existing bus contention analyses [8], [9]. Experimental results show that bus contention analyses that consider the interdependence between the cache misses and bus requests can improve taskset schedulability by up to 55%.

II. SYSTEM MODEL

We assume a multicore system comprising \( m \) identical cores \((\pi_1, \pi_2, \ldots, \pi_m)\) that share the Last-Level Cache (LLC). The LLC is assumed to be partitioned among all the cores such that each core has an individual non-overlapping partition. Each cache partition assigned to the cores is assumed to be large enough to store all the data/code required by the task with the largest memory footprint that executes on that core. We assume that cache employs the write-back policy, is direct-mapped, and, is unified, i.e., it can store data as well as instructions. Furthermore, the write-allocate write-miss policy is assumed in the case of write-miss, which means that the memory block being written to is first loaded in the cache before performing the write operation. Note that the cache analysis presented in Section IV assumes a direct-mapped cache, however, it can be extended to set-associative caches by building on the analysis presented in [21].

We assume that a shared memory bus connects all cores to the main memory. Similarly to existing works [1], [6], [8], [9], [16], [22], [25], a single channel shared memory bus is assumed that can handle one memory request at a time and remains busy until the completion of the ongoing memory request. The maximum service time taken by the bus and the main memory to serve one memory request is given by \( t_{mem} \). Although the proposed cache-aware analysis is applicable to all the bus arbitration policies, in this work, we focus on Round Robin (RR) and First-Come-First-Serve (FCFS) based bus arbitration schemes studied in [8] and [9], respectively.

A. Task Model

This work considers the 3-phase task model (also known as the AER model) in which the execution of a task is divided into three phases namely Acquisition (A), Execution (E), and Restitution (R). The A- and R-phases are considered memory phases, i.e., the time intervals in which the task can fetch and write-back data/code from/to the main memory via the memory bus, and the E-phase is the computation phase, i.e., the time interval in which the task only performs computations using the preloaded data and does not issue any main memory request. When a task is released, it executes its A-phase to fetch the required data/code from the main memory and store it in cache memory. It then executes its E-phase by accessing the data/code that is already available in the cache, without the need to access bus/memory. Finally, the task writes the modified data back to the main memory during the R-phase.

We consider a task set \( \Gamma \) 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 \( \tau_i \), and \( D_i \) denotes its relative deadline. We assume that tasks have constrained deadline, 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 one job of task \( \tau_i \) in isolation is denoted by \( MD^A_i \) (resp. \( MD^R_i \)). Similarly, the WCET of the E-phase is denoted by \( C^E_i \). Note that the values of \( MD^A_i, MD^R_i, \) and \( C^E_i \) can be obtained by static analysis, measurement-based analysis, or by using the combination of both [27]. Finally, the WCET of task \( \tau_i \) in isolation is denoted by \( C_i \) where \( C_i = (MD^A_i + MD^R_i) \times t_{mem} + C^E_i \). We assume that tasks are scheduled using fixed-priority non-preemptive scheduling with priorities assigned using any fixed-priority algorithm such as Rate Monotonic/Deadline Monotonic [15].

Throughout the paper, we refer to the core on which task \( \tau_i \) (i.e., the task under analysis) executes as the local core, denoted by \( \pi_i \). Similarly, any core other than the local core is referred to as a remote core, usually denoted by \( \pi_r \). For notational convenience, we define \( h_{\pi_i,l} \) to denote the set of tasks with a priority higher than or equal to that of \( \tau_i \) running on a given core \( \pi_i \). Similarly, \( h_{\pi_i,l} \) (and \( l_{\pi_i,l} \)) denote the set of tasks with a priority higher (and lower) than that of \( \tau_i \), running on core \( \pi_i \).

III. BACKGROUND

This section presents the essential background on cache related concepts that we later use to build our analysis in Section IV.

When analyzing the worst-case memory access demand of tasks, many existing works [1], [2], [5]–[9], [11], [16], [24], [25] assume that each job of a task \( \tau_i \) that execute during a time window of length \( \Delta \) will always issue \( MD \) main memory accesses, i.e., the worst-case memory access demand of \( \tau_i \) in isolation. This, in other words, means that each job of task \( \tau_i \) that executes during \( \Delta \) will always load all its ECBs from the main memory to the cache.

Evicting Cache Blocks (ECBs) [26]: The set of all memory blocks that may be used by a task \( \tau_i \) during its execution.

Clearly, this assumption is pessimistic, as subsequent jobs of task \( \tau_i \) that execute during \( \Delta \) can re-use some ECBs already available in the cache due to a previous job of \( \tau_i \). These reusable ECBs are called Persistent Cache Blocks (PCBs) [20].

Persistent Cache Blocks (PCBs) [20]: All memory blocks used by a task, that once loaded in the cache, will never be evicted or invalidated by the task itself.

For a task \( \tau_i \) executing in isolation, if all its PCBs are already loaded in the cache, e.g., by a previous job of \( \tau_i \), the memory access demand for subsequent jobs of \( \tau_i \) can be much lower than the worst-case memory access demand of
In isolation. This memory access demand of the task $\tau_i$ is called residual memory access demand.

**Residual Memory Access Demand [20]:** The worst-case memory access demand of any job of task $\tau_i$ considering that all its PCBs are already loaded in the cache.

Considering the PCBs and residual memory access demand of task $\tau_i$, the total number of main memory accesses made by all the jobs of task $\tau_i$ when it executes in isolation during any time window of length $\Delta$ is given by (see Lemma 1 of [20]):

$$MD^{\text{res}}(\Delta) = \min \left( \left[ \frac{\Delta}{T_i} \right] \times MD_i, \left[ \frac{\Delta}{T_i} \right] \times MD_i + |PCB_i| \right)$$

where $\left[ \frac{\Delta}{T_i} \right]$ bounds the maximum number of jobs released by task $\tau_i$ during any time window of length $\Delta$; $MD_i$ is the worst-case residual memory access demand of one job of task $\tau_i$; and $|PCB_i|$ represents the cardinality of the set of PCBs of task $\tau_i$, i.e., the total number of PCBs of task $\tau_i$.

Equation 1 upper bounds the total memory access demand of a task in isolation. However, a task $\tau_i$ will likely have to share the core on which it executes with other tasks. So, the PCBs that were loaded by one job of $\tau_i$ can be evicted by other tasks executing on the same core. This results in generating additional main memory overhead called Cache Persistence Reload Overhead (CPRO) [20].

**Cache Persistence Reload Overhead (CPRO) [20]:** The number of main memory accesses that $\tau_i$ must make due to the evictions of its PCBs caused by the execution of tasks in $hepi,l \setminus \tau_i$.

The maximum number of tasks that can be suffered by one job of task $\tau_i$ is denoted by $\rho_i$, and is given by (from Theorem 1 of [20])

$$\rho_i = PCB_i \cap \bigcup_{\forall \tau_j \in hepi,l \setminus \tau_i} ECB_k$$

where $PCB_i$ is the set of PCBs of task $\tau_i$; and $\bigcup_{\forall \tau_j \in hepi,l \setminus \tau_i} ECB_k$ is the set union of the ECBs of all tasks in $hepi,l \setminus \tau_i$ that can potentially evict PCBs of $\tau_i$.

### IV. Cache-aware Bus Contention Analysis for 3-Phase Tasks

For a task $\tau_i$ scheduled using fixed-priority non-preemptive scheduling, the WCRT is observed during the longest level-i busy window [12]. Formally, the level-i busy window is defined as follows.

**[Level-i busy window (from [14])]:** 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 task $\tau_i$ is positive for all $t \in (a, b)$ and 0 at the boundaries $a$ and $b$.

Due to the problem of bus contention, the length of the level-i busy window depends not only on the behavior of set of tasks running on the same core but also on the bus contention caused by tasks running on remote cores. Consequently, a plethora of works have been proposed in the literature to bound bus contention for 3-phase tasks [11], [8], [9], [16]. Among all these approaches, the works like [8], [9] have proven to be the best approaches to bound bus contention for 3-phase tasks considering partitioned fixed-priority scheduling. Specifically, the analysis presented in [8] bound the bus contention for 3-phase tasks considering the RR bus arbitration policy. Similarly, the analysis presented in [9] bound the bus contention for 3-phase tasks considering the FCFS bus arbitration policy.

Even though the solutions presented in [8], [9] are safe, the bounds on bus contention derived in [8], [9] can be pessimistic. This is mainly because [8], [9] assume that the number of bus/memory requests that can be generated during the memory phases of each job of a task is always equal to its worst-case memory access demand in isolation. As discussed in Section III, this assumption can be pessimistic, as the cache content fetched during the execution of one job of a task may be re-used during the execution of a subsequent job of the same task, resulting in a reduction in the number of bus/memory requests. In the following subsections, we show that a tighter bound on cache misses/bus requests of tasks can improve the bus contention analysis presented of [8], [9].

**A. Cache-aware Bus Contention Analysis for RR Policy**

In the RR bus arbitration policy, when multiple cores require access to the bus then each core can only access the bus during its bus slot. Considering this, the bus contention that can be suffered by tasks will also depend on the number of bus slots that the local core as well as the remote core requires during the level-i busy window. Building on this, the SOTA RR bus contention analysis presented in [8] upper bounds the bus contention by first computing the maximum number of bus slots required by the local core as well as the remote cores. The maximum number of bus slots required by tasks running on the local core $\tau_j$ during the level-i busy window of length $W_{i,l}$ is upper-bounded by $\beta_{\tau_j}(W_{i,l})$, given by the following equation (From Lemma 1 of [8])

$$\beta_{\tau_j}(W_{i,l}) = \sum_{\tau_k \in hepi,j} \left[ \frac{W_{i,l}}{T_h} \right] \times \left( \frac{MD^A \times t^\text{mem}}{SS} + \frac{MD^R \times t^\text{mem}}{SS} \right) + \max_{\forall \tau_k \in hepi,j} \left\{ \frac{MD^A \times t^\text{mem}}{SS} + \frac{MD^R \times t^\text{mem}}{SS} \right\}$$

In Equation 3, the term $\left[ \frac{W_{i,l}}{T_h} \right]$ upper bounds the maximum number of jobs that task $\tau_k$ can release during any time window of length $W_{i,l}$. $MD^A \times t^\text{mem}$ (resp. $MD^R \times t^\text{mem}$) is the maximum number of main memory accesses that can be generated by the A-phase (resp. R-phase) of one job of task $\tau_k$ and $t^\text{mem}$ is the maximum time required to serve one memory request; $SS$ is the length of the bus slot which is always greater than or equal to $t^\text{mem}$. The term $\frac{MD^A \times t^\text{mem}}{SS}$ (resp. $\frac{MD^R \times t^\text{mem}}{SS}$) represents the maximum number of bus slots required by the A-phase (resp. R-phase) of one job of task $\tau_k$. Similarly, due to fixed-priority non-preemptive scheduling, the term $\max_{\forall \tau_k \in hepi,j} \left\{ \frac{MD^A \times t^\text{mem}}{SS} + \frac{MD^R \times t^\text{mem}}{SS} \right\}$ integrates the maximum number of bus slots required by lower priority tasks.

Similarly, the maximum number of bus slots required by tasks running on a remote core $\pi_r$ during the level-i busy window. The term $\left[ \frac{W_{i,l}}{T_h} \right]$ upper bounds the maximum number of jobs that task $\tau_k$ can release during any time window of length $W_{i,l}$. $MD^A \times t^\text{mem}$ (resp. $MD^R \times t^\text{mem}$) is the maximum number of main memory accesses that can be generated by the A-phase (resp. R-phase) of one job of task $\tau_k$ and $t^\text{mem}$ is the maximum time required to serve one memory request; $SS$ is the length of the bus slot which is always greater than or equal to $t^\text{mem}$. The term $\frac{MD^A \times t^\text{mem}}{SS}$ (resp. $\frac{MD^R \times t^\text{mem}}{SS}$) represents the maximum number of bus slots required by the A-phase (resp. R-phase) of one job of task $\tau_k$. Similarly, due to fixed-priority non-preemptive scheduling, the term $\max_{\forall \tau_k \in hepi,j} \left\{ \frac{MD^A \times t^\text{mem}}{SS} + \frac{MD^R \times t^\text{mem}}{SS} \right\}$ integrates the maximum number of bus slots required by lower priority tasks.
window of length $W_{l,i}$ is upper-bounded by $\beta_{\pi_i}(W_{l,i})$, given by the following equation (From Lemma 2 of [8])

$$\beta_{\pi_i}(W_{l,i}) = \sum_{\tau \in \tau_i} \left[ \frac{W_{l,i}}{T_{\tau}} \right] \times \left( \left\lfloor \frac{MD^A_{\tau} \times T_{\tau} \times t^{mem}}{SS} \right\rfloor + \left\lceil \frac{MD^R_{\tau} \times T_{\tau} \times t^{mem}}{SS} \right\rceil \right) \quad (4)$$

We can see in Equations 3 and 4 that when bounding $\beta_{\pi_i}(W_{l,i})$ and $\beta_{\pi_i}(W_{l,i})$, it is assumed that the number of memory requests issued by each A- and R-phase of every job of each task $\tau_i$ is given $MD^A_{\tau}$ and $MD^R_{\tau}$, respectively. As discussed earlier, this can yield a pessimistic bound on bus contention as well on the length of the level-i busy window and WCRT. To address this, we will now present the cache-aware bus contention analysis to improve the bounds on bus contention considering the RR bus arbitration policy.

1) Upper Bounding Memory Access Requests of the Local Core: In this section, we will upper bound the maximum number of main memory accesses generated by all tasks that execute on the local core $\pi_i$ during the level-i busy window $W_{l,i}$. We start by bounding the maximum number of main memory accesses during the A-phases of those tasks.

Applying the notion of PCBs and residual memory access demand to the 3-phase task model, the total number of main memory accesses that can be generated during the A-phases of all the jobs of task $\tau_i$ when it executes in isolation during the level-i busy window $W_{l,i}$ is given by the following lemma.

**Lemma 1.** The total number of main memory accesses that can be generated during the A-phases of all jobs of task $\tau_i$ when they execute in isolation within any time window of length $W_{l,i}$ is given by $MD^A_{\tau}^{tot}$, where

$$MD^A_{\tau}^{tot}(W_{l,i}) = \left| PCB_i \right| + MD^A_{\tau} + \left( \frac{W_{l,i}}{T_{\tau}} - 1 \right) \times MD^A_{\tau} \quad (5)$$

where $PCB_i$ is the set of PCBs of task $\tau_i$ and $MD^A_{\tau}$ is the residual memory access demand of the A-phase of task $\tau_i$.

**Proof.** $\tau_i$ releases at most $\left\lceil \frac{W_{l,i}}{T_{\tau}} \right\rceil$ jobs in the level-i busy window of length $W_{l,i}$. We know that the A-phase of the first job of $\tau_i$ must load all its ECBs, i.e., $\left| PCB_i \right| + MD^A_{\tau}$. Furthermore, by definition of the residual memory access demand $MD^A_{\tau}$, the A-phases of subsequent jobs of $\tau_i$ can make at most $\left( \left\lceil \frac{W_{l,i}}{T_{\tau}} \right\rceil - 1 \right) \times MD^A_{\tau}$ memory accesses. Thus, Equation 5 bounds the maximum number of main memory accesses that can be generated during all the A-phases of task $\tau_i$ when it executes in isolation during $W_{l,i}$.

As discussed earlier, other tasks that can execute on the same core as $\tau_i$ can evict the PCBs of $\tau_i$. Thus, we also need to account for the maximum CPRO that can be suffered by task $\tau_i$, when computing the memory accesses of its A-phases.

The maximum CPRO that can be suffered by an A-phase of one job of task $\tau_i$ is upper bounded by the following equation (from [20])

$$\rho_i = PCB_i \cap \bigcup_{\tau \in hep_l \setminus \tau_i} ECB_h \quad (6)$$

where $PCB_i$ is the set of PCBs of task $\tau_i$, and $\bigcup_{\tau \in hep_l \setminus \tau_i} ECB_h$ is the set union of the ECBs of all tasks in $hep_l \setminus \tau_i$ that can potentially evict the PCBs of $\tau_i$.

The key insight for Equation 6 is that a task $\tau_i \in hep_l$ can only evict the PCBs of task $\tau_i$ if the ECBs of $\tau_i$ shares the same cache lines as the PCBs of $\tau_i$. Note that a lower priority task cannot evict the PCBs of $\tau_i$ within the level-i busy window as it can only execute at the start of the level-i busy window.

Using Equations 5 and 6, the maximum number of memory accesses that can be generated during all the A-phases of $\tau_i$ within a level-i busy window is given by the following lemma.

**Lemma 2.** The maximum number of main memory accesses that can be generated during the A-phases of all the jobs of task $\tau_i$ during any time window of length $W_{l,i}$ is denoted by $MD^A_{\tau}(W_{l,i})$, where

$$MD^A_{\tau}(W_{l,i}) = \min \left( \left\lceil \frac{W_{l,i}}{T_{\tau}} \right\rceil \times MD^A_{\tau}^{tot}, |PCB_i| + MD^A_{\tau} \right) \quad (7)$$

**Proof.** By the definition of $MD^A_{\tau}$, the maximum number of memory accesses that can be generated during the A-phases of $\tau_i$ is denoted by $\left\lceil \frac{W_{l,i}}{T_{\tau}} \right\rceil$ jobs of $\tau_i$ cannot be greater than $\left\lceil \frac{W_{l,i}}{T_{\tau}} \right\rceil \times MD^A_{\tau}$.

From Equation 5, we know that $|PCB_i| + MD^A_{\tau} + \left( \left\lceil \frac{W_{l,i}}{T_{\tau}} \right\rceil - 1 \right) \times MD^A_{\tau}$ upper bounds the maximum number of main memory accesses generated during the A-phases of $\tau_i$ during $W_{l,i}$ when it executes in isolation. Furthermore, from Equation 6, we know that $\rho_i$ bounds the maximum CPRO that can be suffered by the A-phase of one job of $\tau_i$. In the worst-case, the CPRO can be suffered by the A-phases of all the jobs except the A-phase of the first job of $\tau_i$ (as it loads all its ECBs) that execute during $W_{l,i}$. Consequently, $\left( \left\lceil \frac{W_{l,i}}{T_{\tau}} \right\rceil - 1 \right) \times |\rho_i|$ bounds the maximum CPRO that can be suffered by task $\tau_i$ during $W_{l,i}$. Therefore, Equation 7 upper bounds the maximum number of main memory accesses that can be generated during all the A-phases of all jobs of task $\tau_i$ during $W_{l,i}$. The Lemma follows.

Applying Lemma 2 to each task in $hep_l$, the maximum number of main memory accesses that can be generated during the A-phases of all tasks in $hep_l$ that can execute on the local core $\pi_i$ during any time window of length $W_{l,i}$ is given by $\alpha^A_{\pi_i}(W_{l,i})$, where

$$\alpha^A_{\pi_i}(W_{l,i}) = \sum_{\tau \in hep_l \setminus \tau_i} MD^A_{\tau}(W_{l,i}) \quad (8)$$

Having bounded the number of main memory accesses that can be generated during the A-phases, we can now bound the maximum number of main memory accesses generated during the R-phases of tasks.

The R-phase is mainly responsible to write-back all the dirty cache lines (after the execution of the E-phase) to the main memory. In order to tightly bound the number of main memory accesses generated during the R-phase, a task should only
write back a subset of dirty cache blocks that can potentially be used by other tasks to load their ECBs. However, achieving this from the implementation perspective can be extremely complex because: 1) determining the address of specific cache lines that will be dirty at the end of the E-phase of the task is complex, as it depends on the run-time state; 2) enforcing a task to write-back a subset of dirty cache blocks (based on the set of ECBs of other tasks) during the R-phase can be extremely challenging, as it may require additional run-time monitoring/control mechanisms. Therefore, we assume that all cache lines that are dirty at the end of the E-phase of a task will be written-back (and invalidated) during the R-phase. By definition, all such cache lines cannot hold PCBs. Consequently, the memory access demand of an R-phase of a task will account for write-backs due to non-persistent memory blocks and is given by $MD^R$. Building on this, the maximum number of main memory accesses that can be generated during the R-phases of all tasks in $hep_{i,l}$ executing on the local core during $W_{i,l}$ is upper bounded by $\alpha^R_{\tau_i}(W_{i,l})$, where

$$\alpha^R_{\tau_i}(W_{i,l}) = \sum_{\forall \tau_h \in hep_{i,l}} \left\lfloor \frac{W_{i,l}}{T_h} \right\rfloor \times MD^R$$

(9)

Now we will integrate the proposed cache analysis to compute the maximum number of bus slots required by the local core during the level-i busy window using the following lemma.

**Lemma 3.** The maximum number of bus slots required by tasks executing on the local core $\pi_i$ during any time window of length $W_{i,l}$ is upper-bounded by $\beta^R_{\tau_i}(W_{i,l})$, where

$$\beta^R_{\tau_i}(W_{i,l}) = \sum_{\forall \tau_h \in hep_{i,l}} \min \left( \left\lfloor \frac{W_{i,l}}{T_h} \right\rfloor \times \left( \frac{MD^A_{\tau_h} \times t^{mem}}{SS} + \left( \frac{W_{i,l}}{T_h} \right) - 1 \times \left( \frac{MD^R_{\tau_h} \times t^{mem}}{SS} \right) \right), \left( \frac{MD^A_{\tau_h} \times t^{mem}}{SS} \right) \right)$$

(10)

Proof. From Lemma 2, we know that the min $\left( \left\lfloor \frac{W_{i,l}}{T_h} \right\rfloor \times MD^A_{\tau_h} \{|PCB\}| + MD^R_{\tau_h} + \left( \frac{W_{i,l}}{T_h} \right) - 1 \times (MD^A_{\tau_h} + |\rho_i|) \right)$ upper bounds the number of main memory accesses of the A-phases of all jobs of task $\tau_h$ during $W_{i,l}$. Furthermore, from Equation 3, we know that it is necessary to determine the maximum number of bus slots required by the memory phases of tasks that execute on the local core $\tau_i$ during $W_{i,l}$. Consequently, using Lemma 2, the maximum number of bus slots required by A-phases of all jobs of a task $\tau_h$ that execute on the local core $\tau_i$ during $W_{i,l}$ is upper bounded by $\min \left( \left\lfloor \frac{W_{i,l}}{T_h} \right\rfloor \times \left( \frac{MD^A_{\tau_h} \times t^{mem}}{SS} \right), \frac{MD^A_{\tau_h} \times t^{mem} + (\frac{W_{i,l}}{T_h} - 1) \times (MD^R_{\tau_h} + |\rho_i|)}{SS} \right)$. This is further extended for all tasks in $hep_{i,l}$ set (including task $\tau_i$). Since we do not apply cache persistence to the R-phases, the number of bus slots required by R-phases can be computed identically to that of Equation 3, i.e., $\frac{W_{i,l}}{T_h} \times \left( \frac{MD^R_{\tau_h} \times t^{mem}}{SS} \right)$. Finally, the maximum number of bus slots required by one job of lower priority task is bounded by $\max_{\forall \tau_j \in l_{pi,l}} \left\{ \frac{MD^A_{\tau_j} \times t^{mem}}{SS} + \frac{MD^R_{\tau_j} \times t^{mem}}{SS} \right\}$. □

2) Upper Bounding Memory Access Requests of Remote Core: As discussed in Section IV-A1, the maximum number of main memory accesses of tasks depends on the PCBs, residual memory access demand, and CPRO. The notion of PCBs and residual memory access demand can be applied to the A-phases of tasks running on the remote core identically to tasks of the local core (using Equation 5). However, we cannot use Equation 6 to compute CPRO because the interfering task $\tau_u$ executing on $\pi_r$ may not be executing within an uninterrupted level-$u$ busy window during the whole interval $W_{i,l}$. Therefore, we cannot assume that only tasks of higher or equal priority execute between two jobs of $\tau_u$. We must therefore consider that any task executing on core $\pi_r$ may interfere with the PCBs of $\tau_u$ during $W_{i,l}$. Thus, the maximum CPRO that can be suffered by the A-phase of one job of task $\tau_u$ that executes on a remote core $\pi_r$ is given by $\rho_u$, where

$$\rho_u = PCB_u \cap \left( \bigcup_{\forall \tau_h \in \Gamma^r \setminus \tau_u} ECB_h \right)$$

(11)

where $\Gamma^r$ is the set of all tasks running on a remote core $\pi_r$, $PCB_u$ is the set of PCBs of task $\tau_u$, and $\bigcup_{\forall \tau_h \in \Gamma^r \setminus \tau_u} ECB_h$ is the set union of all ECBs of all tasks in $\Gamma^r$ except $\tau_u$.

**Lemma 4.** The maximum number of main memory accesses that can be generated during the A-phases of all the jobs of a task $\tau_u$ running on the remote core $\pi_r$ during any time window of length $W_{i,l}$ is given by $MD^A_{\tau_u}(W_{i,l})$, where

$$MD^A_{\tau_u}(W_{i,l}) = \min \left( \left\lfloor \frac{W_{i,l}}{T_u} \right\rfloor \times MD^A_{\tau_u} \{|PCB\}| + MD^A_u + \left( \frac{W_{i,l}}{T_u} \right) - 1 \times (MD^A_{\tau_u} + |\rho_u|) \right)$$

(12)

Proof. The proof directly follows from Lemma 2 except that the computation of $\rho_u$ is given by Equation 11. □

Applying Lemma 4 to all tasks of the remote core, the maximum number of main memory accesses that can be generated during all the A-phases of all tasks released on the remote core $\pi_r$ during any time window of length $W_{i,l}$ is upper bounded by $\sum_{\forall \tau_u \in \Gamma^r} MD^A_{\tau_u}(W_{i,l})$. As discussed earlier, we assume that a task can invalidate and write back all its non-persistent cache blocks during the R-phase. Considering this, the maximum number of memory requests that can be generated during the R-phase of a task $\tau_u$ is upper bounded by $MD^R_u$. Consequently, $\sum_{\forall \tau_u \in \Gamma^r} \left\lfloor \frac{W_{i,l}}{T_u} \right\rfloor \times MD^R_u$ bounds the maximum number of memory accesses that can be generated during the R-phases of all tasks released on a remote core $\pi_r$ during $W_{i,l}$. We can now use Equation 12 to bound the number of memory accesses of tasks executing on a remote core $\pi_r$ under the RR bus arbitration scheme.
Lemma 5. The maximum number of bus slots required by tasks running on the remote core $\pi_r$ during any time window of length $W_{i,t}$ is upper-bounded by $\beta_{\pi_r}(W_{i,t})$, where

$$\beta_{\pi_r}(W_{i,t}) = \min_{\tau_u \in \Gamma_r} \left( \frac{W_{i,t}}{T_{u}} \times \frac{MD_u^A \times t_{mem}}{SS} \right) + \left( \frac{W_{i,t}}{T_{u}} \times \frac{(MD_u^A + |\tilde{\rho}_u|) \times t_{mem}}{SS} \right)$$

(13)

Proof. The proof directly follows from Lemma 3 except that the computation of $\tilde{\rho}_u$ is given by Equation 11.

In a similar manner to that of Equation 13, we can improve Equations 9 and 16 of [8]. Since a system is likely to have multiple remote cores, we need to bound the maximum number of bus slots that can be utilized by tasks running on each of the remote core $\pi_r \in m$ such that $\pi_r \neq \pi_l$ using Lemma 5. Based on the values of $\tilde{\beta}_{\pi_l}(W_{i,t})$ and $\beta_{\pi_r}(W_{i,t})$, we can compute the maximum bus contention $Bus_{i,t}(W_{i,t})$ that the local core $\pi_l$ can suffer from a remote core $\pi_r$ during $W_{i,t}$ by improving equations 9 and 16 of [8]. Having bounded the maximum bus contention $Bus_{i,t}(W_{i,t})$ w.r.t each remote core $\pi_r \in m$ such that $\pi_r \neq \pi_l$, we can compute the total bus contention $Bus_{max}^{i,t}(W_{i,t})$ that tasks of all remote cores during $W_{i,t}$ using the following equation (from Equation 17 of [8]).

$$Bus_{max}^{i,t}(W_{i,t}) = \sum_{r=1,r \neq i}^{m} Bus_{i,t}(W_{i,t})$$

(14)

B. Cache-aware Bus Contention Analysis for FCFS Policy

The bus contention analysis presented in [9] considers the FCFS bus arbitration policy. The first step of their analysis is to compute the maximum number of times the bus contention that can be suffered by tasks running on the local core $\pi_l$ during $W_{i,t}$ using $N_{\pi_l}(W_{i,t})$ where $N_{\pi_l}(W_{i,t})$ is upper bounded by the following equation (from Equation 9 of [9]).

$$N_{\pi_l}(W_{i,t}) = \sum_{\tau_u \in \Gamma_{\pi_l}} \left( \frac{W_{i,t}}{T_{u}} \right) + 1$$

(15)

The main insight behind Equation 15 is that the maximum number of times bus contention suffered by tasks of the local core depends on the number of jobs/memory phases that tasks release on the local core $\pi_l$ during $W_{i,t}$.

Similarly, the next step is to compute the maximum number of times the bus contention that can be caused by tasks running on a remote core $\pi_r$ during $W_{i,t}$ using $N_{\pi_r}(W_{i,t})$ where $N_{\pi_r}(W_{i,t})$ is upper bounded by the following equation (from Equation 10 of [9]).

$$N_{\pi_r}(W_{i,t}) = \sum_{\tau_u \in \Gamma_{\pi_r}} \left( \frac{W_{i,t}}{T_{u}} \right)$$

(16)

Equation 16 is also derived on the basis of the number of jobs/memory phases that tasks running on a remote core $\pi_r$ can issue during any time window of length $W_{i,t}$.

Finally, the maximum bus contention $Bus_{i,t}(W_{i,t})$ can be computed using different cases based on the values of $N_{\pi_l}(W_{i,t})$ and $N_{\pi_r}(W_{i,t})$. For example, Section 5.3 of [9] computes the bus contention using three cases:

1. $N_{\pi_l}(W_{i,t}) > N_{\pi_r}(W_{i,t})$; 2. $N_{\pi_l}(W_{i,t}) = N_{\pi_r}(W_{i,t})$; and 3. $N_{\pi_l}(W_{i,t}) < N_{\pi_r}(W_{i,t})$.

We will now briefly discuss how the analysis in [9] can be improved for case 1 and the same approach can be used for all the cases considered in [9].

If $N_{\pi_l}(W_{i,t}) > N_{\pi_r}(W_{i,t})$, the maximum bus contention that can be suffered by tasks executing on the local core due to tasks running on a remote core $\pi_r$ during any time window of length $W_{i,t}$ is upper bounded by $Bus_{i,t}(W_{i,t})$, where (from Equation 11 of [9])

$$Bus_{i,t}(W_{i,t}) = \sum_{\tau_u \in \Gamma_{\pi_r}} \left( \frac{W_{i,t}}{T_{u}} \right) \times (MD_u^A \times t_{mem} + MD_u^R \times t_{mem})$$

(17)

We can see that the bound on bus contention given by Equation 17 is pessimistic since it considers the maximum number of memory requests issued during each memory phase in isolation. To address this pessimism, we can apply the exact cache analysis presented in Section IV-A to Equation 17. Using Lemma 4 proposed in Section IV-A2, we can improve and reformulate Equation 17 as follows.

$$Bus_{i,t}(W_{i,t}) = \min_{\tau_u \in \Gamma_{\pi_r}} \left( \frac{W_{i,t}}{T_{u}} \right) \times (MD_u^A \times t_{mem} + MD_u^R \times t_{mem})$$

(18)

In a similar manner to that of Equation 18, we can improve all cases considered in [9]. For example, the bus contention for cases 2 and 3 in [9] is derived by forming sets $M^A$, $M^R$ that contains the length of A- and R-phases in isolation. Consequently, applying the proposed persistence-aware cache analysis allows tightly bounding the length of each of the memory phases, thus, the maximum bus contention.

After bounding the maximum bus contention $Bus_{i,t}(W_{i,t})$ that can be caused by each remote core $\pi_r \in m$ such that $\pi_r \neq \pi_l$, we can compute the total bus contention $Bus_{max}^{i,t}(W_{i,t})$ that tasks of local core $\pi_l$ can suffer due to tasks of all remote cores during $W_{i,t}$ using Equation 14.

V. WORST CASE RESPONSE TIME ANALYSIS

Having bounded the maximum number of memory requests using the analysis presented in Section IV, the bound on the maximum bus contention can be computed by improving the bus contention analysis presented in [8] and [9]. Now we can incorporate the resulting bound on the bus contention into the WCRT analysis of tasks. For this, we propose improved WCRT formulation that accounts for cache reuse when computing bus contention as well as the maximum interference from higher priority tasks. By applying cache persistence to higher priority tasks, we can tightly bound the number of memory accesses issued during their memory phases which in turn reduces the
length of their memory phases. Consequently, it can reduce the overall interference that can be caused by the memory phases of higher priority tasks that execute on the local core during the level-i busy window. Building on this, the length of the level-i busy window is given by the following lemma.

**Lemma 6.** The length of the level-i busy window for a given task $\tau_i$ executing on core $\pi_i$ is denoted by $W_{i,t}$, where $W_{i,t}$ is given by the first positive solution to the fixed-point iteration of the following equation:

$$W_{i,t} = \left(\alpha_{i,t}^{A}(W_{i,t}) + \alpha_{i,t}^{R}(W_{i,t})\right) \times t_{mem} + \max_{\tau_j \in \text{h unpaid}} \{C_j\} + \sum_{\tau_h \in \text{h unpaid}_t} \left[\frac{W_{i,t}}{T_h}\right] \times C^E_h + Bus_{i,t}^{max}(W_{i,t})$$

**Proof.** From Equations 8 and 9, we know that $\alpha_{i,t}^{A}(W_{i,t})$ and $\alpha_{i,t}^{R}(W_{i,t})$ upper bounds the maximum number of memory requests that can be generated during the A- and R-phases of all tasks in $\text{h unpaid}_t$ (including task $\tau_i$) that execute on the local core $\pi_i$ during any time window of length $W_{i,t}$. Assuming that each memory request will take $t_{mem}$ time units, $\left(\alpha_{i,t}^{A}(W_{i,t}) + \alpha_{i,t}^{R}(W_{i,t})\right) \times t_{mem}$ upper bounds the contribution of the memory phases of all tasks in $\text{h unpaid}_t$ that execute on the local core $\pi_i$ during $W_{i,t}$. Due to the fixed priority non-preemptive scheduling, at most one job of a task in $\text{h unpaid}_t$ can cause blocking to $\tau_i$. This blocking is maximized by considering a task with the largest WCET among all tasks in $\text{h unpaid}_t$, given by $\max_{\tau_j \in \text{h unpaid}_t} \{C_j\}$. Similarly, the term $\sum_{\tau_h \in \text{h unpaid}_t} \left[\frac{W_{i,t}}{T_h}\right] \times C^E_h$ upper bounds the contribution of the E-phases of all tasks in $\text{h unpaid}_t$ (including task $\tau_i$) that execute on the local core $\pi_i$ during $W_{i,t}$. Finally, $Bus_{i,t}^{max}(W_{i,t})$ is the total bus contention that can be suffered by tasks that execute on the local core $\pi_i$ from all remote cores during $W_{i,t}$ and can be computed using Equation 14 for the RR and FCFS bus arbitration policies. The Lemma follows.

Note that $W_{i,t}$ appears on both sides of Equation 19 so it needs to be solved iteratively using $W_{i,t} = \sum_{\tau_h \in \text{h unpaid}_t} \left[\frac{W_{i,t}}{T_h}\right] \times C^E_h$ as the starting point.

Having bounded the length of the level-i busy window, we can compute the response time of the $k^{th}$ job of $\tau_i$ on core $\pi_i$, i.e., $\tau_{i,k}$, using the following lemma.

**Lemma 7.** The response time of $\tau_{i,k}$ is denoted by $R_{i,k}$, where $R_{i,k}$ is given by the first positive solution to the fixed-point iteration on the following equation:

$$R_{i,k} = \left(\alpha_{i,t}^{A}(R_{i,k}) + \alpha_{i,t}^{R}(R_{i,k})\right) \times t_{mem} + \max_{\tau_j \in \text{h unpaid}} \{C_j\} + \sum_{\tau_h \in \text{h unpaid}_t} \left[\frac{R_{i,k}}{T_h}\right] \times C^E_h + Bus_{i,t}^{max}(R_{i,k})$$

**Proof.** The proof directly follows from Lemma 6 except considering any time window of length $R_{i,k}$.

Finally, the WCRT of task $\tau_i$ is denoted by $R_{i}^{max}$ and can be computed by maximizing Equation 20 over all jobs of $\tau_i$ that execute during the level-i busy window as follows.

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

where $K_i = \left\lceil \frac{W_{i,t}}{T_i} \right\rceil$.

A taskset is only said to be schedulable if $R_{i}^{max} \leq D_i$ for each task $\tau_i \in \Gamma$ and the total bus utilization of the taskset is less than or equal to 1, i.e., $\sum_{\tau_i \in \Gamma} \frac{(MD_i^A + MD_i^R) \times t_{mem}}{T_i} \leq 1$.

**VI. EXPERIMENTAL EVALUATION**

In this section, we evaluate how much cache-aware bus contention analyses can improve the performance of the SOTA RR-based [8] and FCFS-based [9] bus contention analysis. For the RR analysis, we assume that the bus slot size is equal to $t_{mem}$. This is chosen due to the observation in [8] that the bus contention is least when the bus slot size is equal to $t_{mem}$. Similarly, for the FCFS bus policy, we only consider Fair Memory Access Model (FMAM) based bus contention analysis of [9] since it is the best performing FCFS-based analysis presented in [9] (see Section 8 of [9]).

For the default configuration, we model a quad-core platform with a direct-mapped unified LLC of 32KB (1024 cache sets, 32-byte block) evenly partitioned to the cores. By default, we assume that there were 32 tasks in each taskset with 8 tasks randomly assigned to each core. Tasks utilization $U_i$ was generated using the UUnifast-discard algorithm [4]. Task periods $T_i$ were randomly generated in the range [1000-10000] using log-uniform distribution. The WCET in isolation $C_i$ was then assigned by applying the relation $C_i = U_i \times T_i$. The total memory access demand (MD) of tasks was derived using $C_i$ such that, $MD_i = rand(10\%, 40\%) \times C_i$. The length of the A-phase was chosen randomly in the range [60\%-90\%] of $MD_i$, i.e., $MD_i^A \times t_{mem} = rand(60\%, 90\%) \times MD_i$. The length of the R-phase was then given by $MD_i^R \times t_{mem} = MD_i - (MD_i^A \times t_{mem})$. Finally, the length of the E-phase was given by $C_i^E = C_i - (MD_i^A + MD_i^R) \times t_{mem}$. We assume that tasks are mapped to the cache partition sequentially and in priority order. The number of ECBs of tasks was generated using the length of A-phase, i.e., $|ECB_i| = \frac{MD_i^A}{t_{mem}}$. Similarly, the number of PCBs for each task was generated randomly in the range [20\%-80\%] of its ECBs. Task priorities were assigned using rate monotonic algorithm [15]. Task deadlines were equal to task periods, i.e., $D_i = T_i$.

We compare the performance of the improved bus contention analysis with the existing RR [8] and FCFS policy-based bus contention analysis [9] by varying: 1) the core utilization; 2) the number of cores; 3) the memory access demand; and 4) the number of cache sets. We use taskset schedulability, i.e., the percentage of schedulable tasksets, as a metric to evaluate the performance of each approach.

In all figures, the improved bus contention analyses for the RR computed using Section IV-A and FCFS bus policy computed using IV-B are marked as "Improved RR" and "Improved FCFS", respectively. Similarly, the State-of-the-Art
Fig. 1: Varying Core Utilization and Number of Cores for the RR bus arbitration policy

(a) RR bus policy, m=2
(b) RR bus policy, m=4
(c) RR bus policy, m=8

Fig. 2: Varying Core Utilization and Number of Cores for the FCFS bus arbitration policy

(a) FCFS bus policy, m=2
(b) FCFS bus policy, m=4
(c) FCFS bus policy, m=8

(SOTA) analysis for RR bus [8] is marked as "SOTA RR" and for FCFS bus [9] is marked as "SOTA FCFS". For all experiments, we randomly generated 1000 tasks per point. In all figures, the x-axis represents the core utilization and the y-axis represents the percentage of schedulable tasksets.

1) Varying Core Utilization: In this experiment, we varied the core utilization of each core under the default configuration, i.e., m=4, from 0.05 to 1 in steps of 0.025 and plotted the percentage of tasksets deemed schedulable by all the approaches. Figure 1b (resp. Figure 2b) shows the percentage of tasksets deemed schedulable using the improved analysis and SOTA analysis for the RR bus arbitration policy (resp. FCFS bus arbitration policy). For all approaches, we observe that increasing the core utilization decreases the taskset schedulability. This is because an increase in the core utilization also increases tasks utilization which in turn increases the WCET of tasks as $C_i = U_i \times T_i$. This increase in WCET also increases the number of memory requests, which in turn increases the bus contention suffered by tasks, resulting in a decrease in taskset schedulability. However, we note that the improved analyses for FCFS and RR bus policies outperform the SOTA analysis. For example, we can see in Figure 1b that at a core utilization of 0.475, the improved RR analysis was able to schedule up to 34% more tasksets than the SOTA RR analysis [8]. Similarly, we can see in Figure 2b that at a core utilization of 0.425, the improved FCFS analysis was able to schedule up to 18% more tasksets than the SOTA FCFS analysis [9]. These gains were observed because the improved bus contention analyses use a tighter bound on the number of LLC misses (computed using the analysis in Section IV) when computing bus requests and bus contention. On the contrary, the SOTA bus contention analyses [8], [9] are cache-oblivious policies as they always assume the worst-case number of LLC misses.

Interestingly, we observe that the gain of the improved analysis over the existing analysis was significant for the RR policy (see Figure 1b) but was relatively smaller for the FCFS bus arbitration policy (see Figure 2b). This is mainly because the FCFS bus contention analysis [9] mainly relies on the number of jobs/memory phases that can suffer/cause bus contention during a given time window. In such a case, applying the improved analysis to the FCFS bus may reduce the number of memory accesses per memory phase but it may not reduce the number of memory phases. On the other hand, the bus contention analysis for the RR policy [8] is fine-grained because it depends on the number of bus slots required by the local/remote core which further depends on the number of memory phases as well as the number of memory requests issued during each memory phase. Consequently, the improved analysis is more effective for the RR bus arbitration policy.

2) Varying Number of Cores: In this experiment, we varied the number of cores $m$ from 2 to 8 and plotted the results for the RR and FCFS bus policies in Figures 1 and 2, respectively. As shown in Figures 1 and 2, for all the approaches, increasing the number of cores decreases the taskset schedulability. For instance, when $m = 8$, the number of remote cores as well as the total number of tasks in the taskset increases, i.e., 64 tasks. This results in increasing bus contention, which decreases taskset schedulability. Due to the same reason, decreasing the number of cores improves the performance of all the approaches. Nonetheless, the improved analyses outperform the SOTA bus contention analyses for the FCFS and RR bus policies for all the values of $m$.

3) Varying Memory Access Demand (MD): In this experiment, we varied the Memory Access Demand (MD) of all tasks in the taskset. For this, we consider four configurations:

(a) Very Low (VL) MD, i.e., $MD_i = rand(5\%, 20\%) \times C_i$;
The resulting percentage of schedulable taskset for the RR and FCFS bus policies are plotted in Figures 3 and 4, respectively. From Figures 3 and 4, we can observe that for all approaches, the MD of tasks can significantly impact the taskset schedulability. Specifically, all approaches perform the best under the VL MD configuration and the worst under the VH MD configuration. This happens because increasing the MD value can increase the number of memory requests which in turn increases the bus contention and decreases the taskset schedulability. However, for all the MD configurations, the improved bus contention analyses dominate the existing bus contention analyses. In fact, the improved RR analysis was able to schedule up to 55% more tasksets than the SOTA RR analysis at the core utilization value of 0.275 under VH configuration as shown in Figure 3d.

4) Varying Cache Size: In the default configuration, we assume that the total number of sets in the cache are 1024 and 256 cache sets are allocated per-core. In this experiment, we consider different per-core cache set sizes, i.e., 64, 128, 256, 512, and plot the resulting taskset schedulability in Figure 5a for the RR bus policy and in Figure 5b for the FCFS bus policy. Note that increasing the number of cache sets allocated per-core will effectively increase the total size of the cache. For this experiment, we do not show the taskset schedulability for the existing bus contention analyses as their schedulability does not depend on the cache size.

We can see in Figures 5a and 5b that decreasing the per-core cache sets to 64 or 128 also decreases the schedulability for both the bus arbitration policies. Intuitively, this happens because for a smaller number of cache sets per-core, i.e., 64, 128, the overlap between the PCBs of a task with ECBs of other tasks increases. This leads to a higher CPRO which results in increasing the number of memory requests as well as bus contention. On the contrary, increasing the per-core cache sets to 256 or 512 also increases the schedulability for all approaches. This happens because increasing the number of cache sets allocated per-core reduces the overlap between the PCBs of tasks with ECBs of other tasks, thereby reducing CPRO. This results in improving taskset schedulability.

Interestingly, we observe that the difference between the taskset schedulability for cache set sizes of 256 and 512 is negligible. This happens because, in the default configuration, the value of MD is low, i.e., $MD_i = \text{rand}(10\%, 40\%) \times C_i$, which is further divided among A- and R-phases, with the length of A-phases used to generate ECBs of tasks. This implies that a per-core cache set size of 256 is sufficient enough to ensure that the PCBs of tasks do not overlap with the ECBs of other tasks, i.e., tasks do not suffer CPRO. Consequently, a further increase in the per-core cache set sizes
VIII. Conclusion

In this work, we presented improved bus contention analysis for 3-phase tasks. First, we present an analysis to upper bound the number of cache misses of tasks that lead to bus/memory requests. We then use that bound to improve bus contention that can be suffered by tasks under different bus policies and integrate it into a WCRT-based schedulability analysis. Experimental evaluation performed using synthetic tasksets under different settings show that improved bus contention analyses can improve taskset schedulability by up to 55%. In future works, we plan to extend our approach to heterogeneous platforms.

Acknowledgments. This work was supported by the CISTER Research Unit (UIDP/50032/2020), financed by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), by project ADACORSA (ECSEL/00010/2019 - JU grant nr. 876019) financed through National Funds from FCT and European funds through the EU ECSEL JU. The JU receives support from the European Union’s Horizon 2020 research and innovation programme and Austria, Sweden, Spain, Italy, France, Portugal, Ireland, Finland, Slovenia, Poland, Netherlands, Turkey. - Disclaimer: This document reflects only the author’s view and the Commission is not responsible for any use that may be made of the information it contains. This work is also a result of the work developed under project AeroNext Portugal (nº CA572862-00000066) and FLY-PT (grant nº 464079, POCTI-01-0247-FEDER-046079), also funded by FCT under PhD grant 2020.09532.2B.

References