CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

Schedulability Analysis for Global Fixed-Priority Scheduling of the 3-Phase Task Model

Cláudio Maia*

Geoffrey Nelissen*

Luis Nogueira*

Luis Miguel Pinho*

Daniel Gracia Pérez

CISTER-TR-170603

Schedulability Analysis for Global Fixed-Priority Scheduling of the 3-Phase Task Model

Cláudio Maia*, Geoffrey Nelissen*, Luis Nogueira*, Luis Miguel Pinho*, Daniel Gracia Pérez

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

http://www.cister.isep.ipp.pt

Abstract

Scheduling real-time applications on general purpose multicore platforms is a challenging problem from atiming analysis perspective. Such platforms expose uncontrolledsources of interference whenever concurrent accesses to memoryare performed. The non-deterministic bus and memory accessbehavior complicates the estimations of applications 19 worst-caseexecution times (WCET).The 3-phase task model seems a good candidate to circumventthe uncontrolled sources of interference by isolating concurrentmemory accesses. A task is divided in three successive phases;first, the task loads its instruction and data in a local memory,then it executes non-preemptively using those pre-loaded instructions and data, and finally, the modified data are pushed back tomain memory. Following this execution model, tasks never accessthe bus during their execution phase. Instead, all the bus accessesare performed during the first and third phases.In this paper, we focus on the global fixed-priority schedulingof the 3-phase task model. A new schedulability test is derivedby modelling the interference happening on the bus rather thanthe interference on the cores as in the state-ot-the-art techniques.The effectiveness of the test is evaluated by comparing it againstthe state-of-the-art.

Schedulability Analysis for Global Fixed-Priority Scheduling of the 3-Phase Task Model

Cláudio Maia, Geoffrey Nelissen, Luis Nogueira, Luis Miguel Pinho
CISTER/INESC-TEC, ISEP. Polytechnic Institute of Porto
Email: {crrm, grrpn, lmn, lmp}@isep.ipp.pt

Daniel Gracia Pérez
Thales Research & Technology, Palaiseau, France
Email: daniel.gracia-perez@thalesgroup.com

***Abstract*—Scheduling real-time applications on general purpose multicore platforms is a challenging problem from a timing analysis perspective. Such platforms expose uncontrolled sources of interference whenever concurrent accesses to memory are performed. The non-deterministic bus and memory access behavior complicates the estimations of applications' worst-case execution times (WCET).**

The 3-phase task model seems a good candidate to circumvent the uncontrolled sources of interference by isolating concurrent memory accesses. A task is divided in three successive phases; first, the task loads its instruction and data in a local memory, then it executes non-preemptively using those pre-loaded instructions and data, and finally, the modified data are pushed back to main memory. Following this execution model, tasks never access the bus during their execution phase. Instead, all the bus accesses are performed during the first and third phases.

In this paper, we focus on the global fixed-priority scheduling of the 3-phase task model. A new schedulability test is derived by modelling the interference happening on the bus rather than the interference on the cores as in the state-ot-the-art techniques. The effectiveness of the test is evaluated by comparing it against the state-of-the-art.

I. Introduction

Current commercial-of-the-shelf (COTS) platforms integrate multicore processors in their designs to overcome the physical limitations of single core processors, both in terms of power consumption and heat dissipation. Since they are designed for the average-case performance, it is fairly common that in multicore processors, resources such as memory and buses are shared among the cores. While several applications may potentially benefit from having multiple cores available in the system, having shared resources hinders application development for COTS platforms in industry domains where predictability is one of the most important aspects of application execution (e.g., automotive and avionics domain).

Predictability is affected by concurrent accesses to shared resources, which results in new sources of interference. If there exists behavior not accounted for in the WCET analysis of the tasks composing a system, the actual worst-case timing behavior observed at run-time may drastically deviate from the predictions made at design time. Measuring the effect of interference is a complex task since an accurate modelling of all sources of interference must be available. This leads to the two following challenges: (i) details on processor specific implementations of hardware features are often loosely described by processor producers, and (ii) the difference between the average, peak and worst-case behaviors may be very large, hence resulting in inaccurate timing models.

978-1-5386-1898-1/17/$31.00 © 2017 IEEE

Solutions that decouple task execution from memory accesses are good candidates to circumvent the problem of interference on shared resources. The principle behind this type of models is that jobs of a task are divided into distinct phases: memory phases and execution phases. Tasks use their memory phases to fetch data and instructions from main memory into the core's local memory, and to push back modified data into the main memory. Tasks can then use their execution phase to execute their code without the need of accessing main memory anymore. The PRedictable Execution Model (PREM) [1] is one such solution that relies on a single memory phase and a single execution phase per task.

This paper focuses on the 3-phase task model. In the 3-phase model, tasks are divided into three successive phases. First, in their *Acquisition* phase (A-phase) tasks load their instructions and data into a core's local memory. Then, tasks execute non-preemptively using those pre-loaded instructions and data during their *Execution* phase (E-phase). Finally, the modified data are pushed back to main memory during the Restitution (R-phase). Tasks never access the bus during their Execution phase. Instead, all the bus accesses are performed altogether during the first and third phases. The time required to perform each phase can therefore be accurately computed. Another property enforced at run-time by the solution discussed in this paper, is that at most one task can perform memory accesses at a time. Hence, uncontrolled interference is avoided. These properties make the model suitable for real-time and embedded multicore systems as system predictability may be improved by having memory phases and execution phases of different tasks executing in parallel while avoiding, at the same time, memory contention issues when tasks access main memory.

Specifically, this paper presents a new schedulability test for the global fixed-priority scheduling of the 3-phase task model in a multicore setting. This work differs from current state of the art, (i.e., [2]), by analyzing the schedulability of the system from a bus perspective instead of a core's perspective and fill in some gaps left open by their analysis (for instance, their work neglects some of the interference generated by R-phases). The worst-case scenario that upper-bounds the interference on the bus is identified and from this scenario a schedulability test is derived.

Contributions. This paper improves current state of the art

schedulability test for the 3-phase task model, namely [2]. In particular we provide the following contributions: (1) we provide a new perspective on the analysis of the schedulability problem of the 3-phase task model and introduce the concept of bus holes; (2) we identify a couple of shortcomings in the state-of-the-art analysis and propose solutions; (3) we derive a schedulability test that provides an upper-bound on the interference occuring on the bus; and finally (4) we compare our approach with the current state-of-the-art approach by extensive simulations.

Paper organization. The remainder of this paper is organized as follows. Section II presents the related work. Section III describes the system model used throughout the paper and Section IV the runtime model. In Section V some background is presented and in Section VI the new analysis concepts are introduced. Section VII details the schedulability analysis. Section VIII presents the simulation results from experiments conducted on synthetic task sets. Finally, Section IX concludes the paper and presents perspectives for future works.

II. Related Work

Several models exist in the literature that focus on the analysis of bus interference either by considering time-driven approaches (i.e., [3]) or event-driven approaches (i.e., [4]). In particular, in [3] the authors show that among the several resource access models analysed in that work, the one that performs best is the 3-phase task model. The 3-phase task model is a generalization of the PRedictable Execution Model (PREM) [1]. In PREM, tasks consist of two distinct phases: a memory phase and an execution phase. In the memory phase, tasks fetch data and instructions from main memory while in the execution phase tasks execute without requiring any access to shared memory. PREM was originally designed for single core systems and later extended to multicores in [5]. The multicore version of PREM follows an offline time-driven arbitration approach to assign fixed slots of time to cores to access main memory. In addition, each core has its own scheduling policy. During runtime, the execution of memory phases is promoted by increasing their priority over the execution phases, whenever a core is allowed to access the main memory.

The 3-phase task model has been subject to experiments carried out to evaluate the applicability of the model in different domains. In [6] the authors use the 3-phase task model to model periodic tasks in a flight management system. Moreover, in [7], the authors show that executing tasks in a multicore system leads to increases in the WCET measured in isolation of up to 3x the value in isolation, and that by using the 3-phase task model it is possible to obtain an interference-free execution in a multicore system. In [8] several global priority assignment heuristics are presented for scheduling 3-phase tasks. The authors compare each of the heuristics with a global-EDF scheduling approach that takes into account interference. The conclusion of this study is that using the task period to decide the priority of the task is a good option and that in some cases this assignment presents a higher schedulability ratio than the interference prone global-EDF version. In [9], the 3-phase task model is applied to AUTOSAR applications in order to obtain a contention free execution for this type of applications in a many-core architecture. In [10], the authors integrate the 3-phase task model with TDMA managed accesses to system bus as a way to serialize the acquisition and restitution phases in the design of a multicore operating system for embedded scratchpad-based multicore architectures.

Regarding the above mentioned works, none of them tackle the global scheduling of the 3-phase task model. The only works, to the best of our knowledge, that follow the same line of research as the work presented in this paper are the works presented by Alhammad and Pellizzoni in [2] and in Chapter 3 of [11]. Alhammad's work focuses on the schedulability analysis of the 3-phase task model by upper-bounding the interference a task suffers in a problem window from a core's perspective. In this paper we improve their analysis by applying a different analysis perspective, i.e., from a bus perspective. Moreover we fill in the gaps in [2] with respect to the restitution phases that may execute after the problem window. Section V presents an in-depth overview of [2] so that the reader is able to understand how these approaches compare with each other.

III. System Model

We consider a system composed of m identical cores where each core accesses the system's main memory using a shared bus. From a core's perspective the shared bus is a shared resource. Therefore, it is a source of interference whenever concurrent accesses are made by different cores to fetch data from main memory to the core's local memory.

We assume that Input/Output (I/O) data transfers from or to the main memory are performed using a Direct Memory Access (DMA) controller. We also assume that the local memory (*e.g.,* scratchpad, L1 cache) associated with each core is large enough to fully store any task's code and data and may fit the data and instructions of only one task at a time. If this is not the case, the task should be divided in smaller entities, each entirely fitting in the core's local memories. At any time, at most one task can be saved in each local memory.

Task Model: We consider a system composed of n independent real-time tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i in τ has a fixed priority. We denote the set of tasks with higher or equal priority than τ_i (including τ_i) by $hep(i)$, and we use $lp(i)$ to denote the set of tasks with lower priority than τ_i.

Each task comprises three distinct phases, namely, the acquisition (A), execution (E) and restitution (R) phases. Phases have a precedence constraint in the sense that a job must first execute its A-phase, then its E-phase and finally its R-phase. Each phase executes *non-preemptively*.

We let A_i, E_i and R_i denote the maximum execution time of the A, E and R-phase of task τ_i, respectively. The worst-case execution time (WCET) in isolation of τ_i (without suffering any interference) is given by the sum of the execution times of each phase, i.e., $C_i = A_i + E_i + R_i$. Each task is characterized

further by a period T_i and a constrained-deadline $D_i \leq T_i$. That is, the A-phases of every two successive jobs of τ_i are released at least T_i time units apart, and the R-phase of a job of τ_i must complete at most D_i time units after the release of the A-phase of that same job. Therefore, for a task to be schedulable, its WCET should be no greater than its relative deadline, i.e., $C_i \leq D_i$.

The *utilization* of task τ_i is given by $U_i \stackrel{\text{def}}{=} \frac{C_i}{T_i}$ while the *total utilization* of the task set τ is given by $U_\tau \stackrel{\text{def}}{=} \sum_{i=1}^{n} U_i$. Moreover, the memory utilization by a task τ_i is given by $M_i \stackrel{\text{def}}{=} \frac{A_i+R_i}{T_i}$. To ensure the feasibility of the system, the core's utilization should not exceed 100% and consequently, the total system utilization should be no greater than the number of cores in the system, i.e., $U_\tau \leq m$. Similarly, the system bus utilization should not exceed 100%, i.e., $\sum_{i=1}^{n} M_i \leq 1$.

Shared Resource Model: The shared resource covered in this paper is the system bus. Specifically, whenever a task executes a memory phase (either A or R), one of the cores locks the bus and initiates a memory request to fetch/store data from/to main memory. The core releases the bus at the end of the memory phase. Therefore, memory phases are non-preemptive and at most one task executes a memory phase at any time instant.

IV. Runtime Execution Model

A and R-phases are *memory phases* during which each application transfers data between main memory and the core's local memory (*e.g.,* scratchpad, L1 cache). When an A-phase starts, the code and data needed for the task's execution are fetched from main memory to the core's local memory. We assume that anything running on a core must have been loaded in the core's local memory first in order to avoid non-deterministic accesses to the bus during the execution of the E-phase. Consequently, one cannot have one task running on a core and another one being loaded to the core's local memory at the same time. After completing execution, the R-phase pushes back to the main memory the data modified by the task that were saved in the core's local memory. This behaviour entails that all the content of the core's local memory must have been pushed back to main memory before the A-phase of the next task assigned to that core can be started. Therefore, a core remains idle whenever an A/R-phase is executed by a task running on that core. In this execution model, a task does not require any access to the bus during its E-phase and hence does not suffer unpredictable interference due to tasks executed on other cores.

A. Scheduling Policy

Jobs released by tasks are executed on cores in a non-preemptive global fixed-priority manner. Once assigned to a core, a job starts the execution of its A-phase, followed by its E-phase and finally its R-phase in a non-preemptive manner. Thus, at any given time instant there are at most m uncompleted jobs that have started their execution.

Even though execution is non-preemptive, a job might have to wait between its E and R-phase to gain access to the bus (remember that the bus is locked by cores to ensure exclusive access to the main memory during an A or R-phase). If a job J must start a R-phase and the bus is already busy serving another memory phase of a job executing on another core, J spin-locks (non-preemptively) waiting for the bus to be freed.

We assume that A-phases have always higher priority than R-phases to access the bus. We further assume that R-phases execute in a FIFO order. FIFO ordering ensures progress. If priority ordering was also used to schedule R-phases, then a low priority job that started executing can have its R-phase unboundedly delayed if high priority jobs keep being released and executed on other cores. Conversely, FIFO ordering ensures that a low priority job cannot be blocked (spin-locking), and hence keep a core idle, indefinitely due to higher priority tasks running on other cores. However, on the down side, using FIFO means that more than one lower priority task can block higher priority ones during their R-phase.

The scheduler is event-driven. It is invoked whenever one of the following events happens: (1) a job release; (2) the completion of a A, E or R-phase.

The scheduler uses two different queues to keep track of ready phases (i.e., phases that are waiting to access a processor and/or the bus). The phases pushed in the first queue (henceforth called PriorityQueue) are sorted in a non-increasing priority order. The phases pushed in the second queue (referred to as FIFO-Queue) are ordered following a first-in first-out (FIFO) ordering policy. Following the idea described above, ready A-phases are always pushed into the PriorityQueue, while ready R-phases are pushed in the FIFOQueue. Hence, at a job release, the A-phase of the released job is enqueued into the PriorityQueue. Similarly, when the E-phase of a job completes, the R-phase of that job is inserted into the FIFO-Queue.

Algorithm 1 provides a pseudo-code of the scheduling algorithm executed at each scheduler invocation. It first checks if the bus is available. If it is not, then it simply exits and waits for the active memory phase to complete its execution. If the bus is free then the scheduler checks if at least one of the two queues contains ready memory phases. Since, following our assumption, A-phases have higher priority than R-phases, the scheduler checks first if there is an A-phase waiting in the PriorityQueue.

If an A-phase is ready, the scheduler then checks if there is an idle core π_k. If it is the case, the job to which the A-phase belongs to is assigned to π_k, the bus is locked and the the A-phase starts on the bus. Otherwise, if no core is available then the A-phase must wait until another job completes its execution and releases a core. If no A-phase can be started, either because the priority queue is empty or no core is free, the FIFOQueue needs to be checked for ready R-phases so that any job that still has a pending R-phase can be completed and the core be freed to execute other jobs.

When an A-phase completes its execution on a core π_k, the bus is unlocked and the E-phase of the respective job immediately starts its execution on core π_k. Hence, there is no idle time between the completion of an A-phase and the

Algorithm 1 Scheduling algorithm pseudo-code

1: **if** Bus is Free **then**
2:     **if** PriorityQueue not empty **then**
3:         **if** Free Core Available **then**
4:             Pull the task τ_i with the highest priority A-phase from the PriorityQueue;
5:             Assign τ_i to one of the free cores;
6:             Lock the bus and start the A-phase of τ_i;
7:             **return**;
8:         **end if**
9:     **end if**
10:     **if** FIFOQueue not Empty **then**
11:         Take the first R-phase from the FIFOQueue;
12:         Lock the bus and start the R-phase;
13:     **end if**
14: **end if**

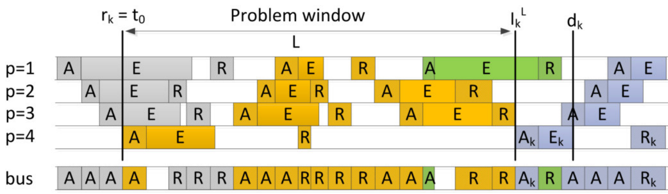


Fig. 1: Problem Window

execution of its corresponding E-phase. Further, there is no migration from one core to any other core between phases of a same job. Finally, at the completion of a job E-phase, the R-phase of this job is enqueued into the FIFOQueue. Upon completion, an R-phase releases both the bus and the core on which it executed.

V. Background

Alhammad and Pellizzoni [2] provide a schedulability test for the 3-phase task model based on the technique proposed in [12] and [13] for global non-preemptive scheduling on multiprocessor systems.

The schedulability test consists in analyzing a time interval $[t_0, d_k - (A_k + E_k)]$ of a job of τ_k which is assumed to miss its deadline at time d_k. That job is called the problem job. The time instant t_0 is the latest time instant earlier than the release r_k of τ_k's problem job at which at least one processor is idle. The interval $[t_0, d_k - (A_k + E_k)]$ is called problem window and is depicted in Fig. 1.

Intuitively, if the scheduler is work-conserving, for the problem job to miss its deadline, the amount of interference[1] occurring in the problem window must be greater than the computing supply[2]. Following this observation, computing an upper-bound on the interference suffered by each task and comparing it against a lower bound on the supply available in the problem window allows us to determine whether the system is schedulable or not. More precisely, if the maximum interference each task can suffer is less than the minimum supply in their problem window then the system is schedulable.

In [2], the authors apply the above analysis technique to the 3-phase task model and show that the problem window must be an interval during which either the bus is busy executing memory phases, or all processors are busy executing E-phases.

Similarly to [12], the worst-case workload of each higher-priority task within the problem window (and hence the interference generated by each higher priority task on the problem job) is divided into three parts:

1) Carry-in workload: The carry-in workload is composed of jobs (henceforth called carry-in jobs) released before t_0 and with their deadline after t_0. These jobs are represented in gray in Fig. 1.
2) Body jobs: Body jobs have their release and deadline entirely contained within the problem window. These jobs are represented in yellow in Fig. 1.
3) Carry-out workload: The carry-out workload is composed of jobs (henceforth called carry-out jobs) released within the problem window but with their deadline outside of the problem window. These jobs are represented in green in Fig. 1.

The contribution of each of those jobs to the interference suffered by the problem job is analyzed in detail below.

A. Carry-in Workload

Concerning the carry-in jobs, it was proven in [2] that at most m tasks can have a carry-in job among which at most $(m-1)$ are from higher or equal priority tasks. Furthermore, since we assume a constrained-deadline task model, each task can have at most one carry-in job. It was further proven that the worst-case interference happens when $(m-1)$ cores are busy executing E-phases from carry-in jobs while a lower priority task blocks the execution of τ_k at t_0. This result is formally stated in Theorem 1 below.

Theorem 1. *(from [2]) In the worst-case, the carry-in workload at time t_0 is limited by $m-1$ computation phases (E-phases) from tasks in $hep(k) \cup lp(k)$ and one full job from a task in $lp(k)$.*

B. Schedulability Analysis

To compute the interfering workload of the jobs without carry-in, the approach in [2] focuses on what happens on the cores within the problem window. Specifically, within the problem window, the interfering workload must consider the contribution of $\left\lfloor \frac{L_k}{T_i} \right\rfloor$ body jobs and at most one carry-out job of each higher priotity task $\tau_i \in hp(k)$.

Due to the restriction that no two memory phases can execute simultaneously on the bus, the schedule contains scheduling *holes* (see the grey blocks in Fig. 2). A scheduling hole is an interval of time in which a core is idle as a result of a memory phase executed by another core.

Therefore, to bound the interference suffered by a job in its problem window, one must also upper bound the cumulative length of the holes on the cores. Alhammad and Pellizzoni do it by lower bounding the time during which the execution

[1]The interference (I_k^L) over a task τ_k in an interval of length L is defined as the sum of all the time intervals in which τ_k is ready but cannot execute due to the execution of higher priority tasks.

[2]The supply in a time interval is the total amount of computation that could be performed within the interval.

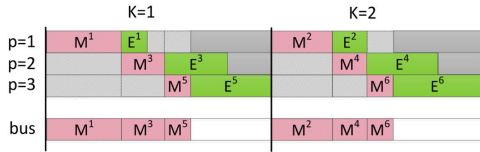


Fig. 2: $\rho = 2, m = 3$

of E-phases on cores *overlap* with memory phases executed on the bus. The holes total length is then given by ($m \times \sum_{i=1}^{\alpha} \mu_i -$ overlap) where $\sum_{i=1}^{\alpha} \mu_i$ is the sum of all memory phases executed in the problem window.

In order to compute a lower-bound on the amount of overlap (equivalently, an upper bound on the length of holes), the authors in [2] propose the following approach. First, the largest A-phases are combined with the largest R-phases into single memory phases $M^i = A^i + R^i$, with $A^i \geq A^{i+1}$ and $R^i \geq R^{i+1}$. Each element M^i is added to a sequence μ sorted in a non-increasing order. The E-phases are sorted in a sequence λ in a non-decreasing order.

Let α be the size of μ and λ, $\rho \leq \frac{\alpha}{m}$ partitions are created as depicted in Fig. 2. The ρ largest memory phases in the sequence μ and the ρ smallest computation phases in λ are assigned to the first core. The second ρ largest memory phases in the sequence μ and the ρ smallest computation phases in λ are assigned to the second core. This procedure is repeated on each core. Thus, by following this assignment the largest memory phases overlap with the smallest computation phases leading to a lower-bound on the amount of overlap between the phases. The length of the holes in each partition k is then upper-bounded by the length of the grey blocks in Fig. 2.

Eq. (1) summarizes the approach in [2]. The workload interfering with τ_k within the problem window L_k is given by the sum of the α memory phases plus the sum of all E-phases executing in the interval minus a lower bound on the overlap of the E-phases which can be computed following the procedure sketched above. For more details please check section 4.3 in their paper.

$$W_k^{NC} = m \cdot \sum_{i=1}^{\alpha} M^i + \sum_{i=1}^{\alpha} \lambda^i - \text{overlap} \tag{1}$$

C. Limitations

While the approach proposed in [2] is interesting from an analysis viewpoint, we note two main limitations.

- First, it considers that a task is schedulable if it completes its E-phase by its deadline, as depicted in Fig. 1. Therefore, it omits the time required for the task to execute its R-phase. Since the R-phase is in charge of writing the results of the task computation back to main memory, this can be problematic if tasks have precedence constraints or any form of data dependencies.
- Second, it can be very pessimistic. Specifically, by only looking at the overlap that occurs in each partition, the analysis misses the overlap that exists across partitions. Comparing Fig. 2 and Fig. 3, one can see an example of this pessimism. By allowing the memory phases of the second partition to start as soon as possible (as in Fig. 3) one can decrease the amount of interference by more than E_5 time units in comparison to Fig. 2 (which is the execution scenario assumed in [2]).

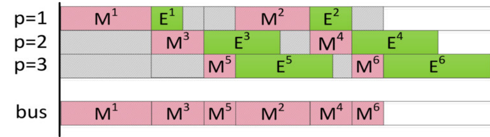


Fig. 3: Pessimism of the analysis in [2]

VI. A Different Perspective

In this paper, we look at the problem of the 3-phase task model inter-task interference from a different perspective. While [2] analyzes the schedulability of each task by modelling the scheduling behavior on the cores, we consider what occurs on the bus. Analyzing the bus instead of the cores reduces the schedulability problem to a single core problem (there is only one bus which executes at most one memory phase at a time) instead of a multicore problem.

Yet, similarly to the fact that scheduling holes can appear on the cores when a memory phase is being processed on the bus, *bus holes* can be observed on the bus whenever all the cores are busy executing E-phases (see Fig. 4). Bus holes happen because, when all the cores are busy executing E-phases, none of the local memories can accept new content nor can the computation result be written back to main memory. Hence, the bus remains idle. Formally, a bus hole is defined as follows.

Definition 1 (Bus Hole). *A bus hole is an interval of time, within the problem window, where all m cores are busy executing E-phases.*

Our analysis builds upon the observation that within the problem window, the contribution of the jobs to the response time of the task under analysis is divided into two parts: (1) the interference of the memory phases (A and R) that execute within the window, and (2) the cumulative length of time during which all cores execute E-phases (if any such interval exists). We denote this latter length by L_i^{holes}.

Upper bounding the length L_i^{holes} and adding its value to the total time required to process memory phases of body, carry-in and carry-out jobs executed in the problem window results in an upper bound on the interference that the task under analysis may suffer in the worst-case. The worst-case interference that a task τ_i can suffer in an interval of length t is therefore bounded by

$$I_i(t) = L_i^{holes}(t) + I_i^{bus}(t) \tag{2}$$

where $L_i^{holes}(t)$ is the maximum cumulative time m different E-phases are simultaneously executing on the m cores in an interval of length t, and $I_i^{bus}(t)$ is an upper bound on the interference τ_i can suffer on the bus due to the execution of memory phases of other jobs during an interval of length t.

To compute the exact length of $L_i^{holes}(t)$ one has to know how the jobs of each task are scheduled on the cores within

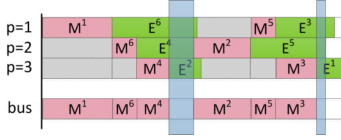


Fig. 4: Our approach

the problem window. Checking all potential jobs' schedules to find the schedule generating the longest cumulative length $L_i^{holes}(t)$ is intractable. Therefore, in this paper, we propose a pseudo-polynomial technique to compute an upper bound on $L_i^{holes}(t)$.

To summarize, the technique proposed in this paper differs from [2] in the sense that we upper-bound the interference on the bus instead of upper-bounding the interference on the cores. To achieve this, we must compute an upper-bound on the length of the so-called bus holes. By definition of bus holes, if one can maximize the length of the intervals where all cores are simultaneously busy executing E-phases, then an upper-bound to the length $L_i^{holes}(t)$ is found.

VII. Schedulability Analysis

As already explained in Section V, a task τ_i is schedulable if $I_i(t) \leq t$ where t is a lower-bound on the length of the problem window of τ_i and $I_i(t)$ is the maximum interference suffered by τ_i in that window.

An upper-bound on $I_i(t)$ can be computed using Eq. (2), where the term $L_i^{holes}(t)$ accounts for the interference suffered by τ_i due to the execution of E-phases while $I_i^{bus}(t)$ considers the interference caused by memory phases executed in an interval of length t. Yet, before computing $L_i^{holes}(t)$ and $I_i^{bus}(t)$, one should know the length t of the problem window on which $L_i^{holes}(t)$ and $I_i^{bus}(t)$ must be computed. In [2], the authors use $t = D_i - E_i - M_i$ (where $M_i = A_i + R^{\max}$ with $R^{\max}$ being the largest R-phase executed in the problem window) as that length. However, as already pointed out in Section V-B, one of the main limitations of the analysis presented in [2] is that it does not consider the time required by the R-phase of τ_i to write its data back in main memory. Therefore, in this section, we first prove an upper-bound on the time needed for τ_i to complete its R-phase. Then, we use that information to derive a bound on the length t that must be considered in the schedulability test of any task τ_i. Finally, in Sections VII-B and VII-C, we prove upper-bounds on $I_i^{bus}(t)$ and $L_i^{holes}(t)$, respectively.

A. R-phase Worst-Case Response Time and Problem Window Length

Let $\overrightarrow{A}_i$ and $\overrightarrow{R}_i$ be the set of A and R-phases of the tasks in $\tau \setminus \tau_i$ sorted in a non-increasing order (where τ_i is the task under analysis). We denote by $\overrightarrow{A}_i^{(k)}$ (resp., $\overrightarrow{R}_i^{(k)}$), the kth element in $\overrightarrow{A}_i$ (resp., $\overrightarrow{R}_i$). Therefore, $\overrightarrow{A}_i^{(k)}$ is the kth largest A-phase among those executed by tasks in τ.

Lemma 1. *The interference suffered by the R-phase of τ_i is upper-bounded by:*

$$I_i^R = \sum_{k=1}^{m-1} \left(\overrightarrow{A}_i^{(k)} + \overrightarrow{R}_i^{(k)} \right) \quad (3)$$

Proof. R-phases are inserted in a FIFO queue. Therefore, the worst-case for the task τ_i under analysis occurs when $(m-1)$ other jobs inserted their R-phases before τ_i in the queue. Hence, τ_i has to wait until all those other R-phases complete before τ_i's R-phase can start. Further, because tasks have constrained deadlines, each task has at most one active job and hence one active R-phase at any time (assuming the system is schedulable). Therefore, the (at most) $(m-1)$ R-phases interfering with τ_i's R-phase are from different tasks. The contribution of R-phases to the interference of τ_i is thus upper-bounded by the sum of the $(m-1)$ largest R-phases in the system, i.e., by $\sum_{k=1}^{m-1} \overrightarrow{R}_i^{(k)}$.

Furthermore, for each completed R-phase, a core is freed and Algorithm 1 is called. Since A-phases have higher priority than R-phases, the transmission of τ_i's R-phase can be delayed by the transmission of a ready A-phase. Note however that a maximum of $(m-1)$ cores can be freed before the transmission of τ_i's R-phase, and therefore, by Line 3 of Algorithm 1, at most $(m-1)$ A-phases can interfere with τ_i's R-phase. Similarly to the discussion for R-phases, because each task can have at most one A-phase ready at any time, the $(m-1)$ A-phases interfering with τ_i's R-phase must be from different tasks. The contribution of A-phases to the response-time of τ_i's R-phase is thus upper-bounded by the sum of the $(m-1)$ largest A-phases in the system, i.e., by $\sum_{k=1}^{m-1} \overrightarrow{A}_i^{(k)}$.

Adding both contributions, we get that the interference suffered by τ_i's R-phase is upper-bounded by $\sum_{k=1}^{m-1} \left(\overrightarrow{A}_i^{(k)} + \overrightarrow{R}_i^{(k)} \right)$. □

Corollary 1. *The response time of the R-phase of τ_i is upper bounded by $R_i + I_i^R$.*

Proof. Directly follows from Lemma 1. □

Now that we have an upper bound on the response time of τ_i's R-phase, we derive a bound on the length t of the problem window.

Lemma 2. *If the problem job of τ_i misses its deadline, then $I_i(t) \geq t$ where $t = D_i - A_i - E_i - R_i - I_i^R + \epsilon$ and ϵ is an arbitrary small number.*

Proof. Let us assume that the problem job of τ_i is released at time r_i and has its deadline at time $r_i + D_i$. We prove the claim by contradiction. Let us assume that $I_i(t) < t$. Since $I_i(t)$ sums all the instants where the bus is busy executing memory phases or all cores are busy executing E-phases (see Eq. (2)), then, by our contradictory assumption, there must exist an instant t_{idle} such that $r_i \leq t_{idle} < r_i + t$ at which both the bus is idle and at least one core is idle.

By Algorithm 1, the A-phase of τ_i's problem job can start executing on the bus at t_{idle}. Since A and E-phases execute non-preemptively, they complete their execution by $t_{idle} + A_i + E_i$. Furthermore, since the response time of τ_i's R-phase is upper-bounded by $R_i + I_i^R$ (Corollary 1), the R-phase of τ_i's problem job completes by $t_{idle} + A_i + E_i + R_i + I_i^R$. Replacing t_{idle} by its upper-bound, we get $t_{idle} + A_i + E_i + R_i + I_i^R < r_i + D_i - A_i - E_i - R_i - I_i^R + \epsilon + A_i + E_i + R_i + I_i^R = r_i + D_i + \epsilon$. Since ϵ is an arbitrarily small number, the R-phase of τ_i's problem job therefore completes at or before $r_i + D_i$. It is a

contradiction with the assumption that the problem job of τ_i misses its deadline, hence the claim. □

Theorem 2. *If for all $\tau_i \in \tau$, $I_i(t) < t$ where $t = D_i - A_i - E_i - R_i - I_i^R + \epsilon$ and ϵ is an arbitrary small number, then the system is schedulable.*

Proof. It is the contra-positive of Lemma 2. If $I_i(t) < t$ for any task τ_i, then every job of τ_i meets its deadline. It follows that if the condition is true for all tasks then all jobs meet their deadlines and the system is schedulable. □

B. Upper-bound on $I_i^{bus}(t)$

In this section, we derive an upper-bound on $I_i^{bus}(t)$.

Let $\mathcal{J}(t)$ be the largest set of jobs that can execute (completely or partially) in an interval of length t and prevent τ_i's A-phase to start executing. We divide the set $\mathcal{J}(t)$ in two different subsets composed of (i) carry-in jobs, and (ii) body and carry-out jobs, respectively.

With respect to (i), Theorem 1 tells us that the carry-in workload is upper-bounded by $(m-1)$ computation phases (E-phases) from tasks in $hep(k) \cup lp(k)$ and one full job from a task in $lp(k)$. Since every E-phase is followed by an R-phase, and because every full job has both an A and an R-phase, the contribution of the carry-in workload to $I_i^{bus}(t)$ is upper-bounded by $A_{low}^{max} + \sum_{k=1}^{m} \overrightarrow{R}^{(k)}$, where A_{low}^{max} is the largest A-phase among the tasks with lower priority than τ_i and $\overrightarrow{R}^{(k)}$ is the kth largest R-phase among all tasks in τ.

Regarding the number of body and carry-out jobs in $\mathcal{J}(t)$, two cases must be considered:

- $\tau_k \in hp(i)$. The maximum number of jobs of τ_k that can be released and have their deadline within an interval of length t is upper bounded by $\left\lfloor \frac{t}{T_k} \right\rfloor$. The contribution of body jobs of τ_k to $I_i^{bus}(t)$ is thus upper-bounded by $\left\lfloor \frac{t}{T_k} \right\rfloor (A_k + R_k)$. Further, the contribution of τ_k to the carry-out workload is limited to one job of size at most $\min\{(A_k + R_k), (t \bmod T_k)\}$ (i.e., since we have a constrained-deadline model, each task can have at most one carry-out job, that job is released no earlier than $(t \bmod T_k)$ time units before the end of the problem window, and the carry-out job cannot execute for more than $(t \bmod T_k)$ in $(t \bmod T_k)$ time units).
- $\tau_k \in lep(i)$. If τ_k's priority is lower than or equal to the priority of τ_i, then, thanks to Line 4 of Algorithm 1, no job released by τ_k after or at the same time than a job of τ_i can interfere with τ_i. Therefore, no body or carry-out job of τ_k participates to $I_i^{bus}(t)$.

Finally, adding the contribution of all jobs in $\mathcal{J}(t)$ together, we get that

$$I_i^{bus}(t) \leq A_{low}^{max} + \sum_{k=1}^{m} \overrightarrow{R}^{(k)} + \sum_{\tau_k \in hp(i)} \left(\left\lfloor \frac{t}{T_k} \right\rfloor (A_k + R_k) + \min\{A_k + R_k,\ t \bmod T_k\} \right) \quad (4)$$

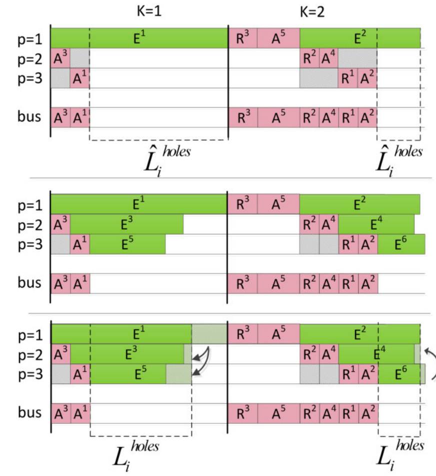


Fig. 5: Computing an upper-bound on bus holes

C. Upper-bound on $L_i^{holes}(t)$

The length $L_i^{holes}(t)$ provides an upper bound on the total time during which all cores are busy executing E-phases in a window of length t. Hence, $L_i^{holes}(t)$ depends on the jobs' schedule in that window. Finding the worst-case schedule that provides the largest length $L_i^{holes}(t)$ is intractable in the general case. Therefore, we provide an over-approximation of that length by building an artificial schedule of memory and execution phases that is at least as bad as the worst-case schedule.

Intuitively, the length $L_i^{holes}(t)$ is maximized by considering an artificial schedule as follows. Let us assume that k successive E-phases execute on the first core, and ℓ memory phases execute on all other cores, in parallel with those E-phases. Since there is only one memory bus, at most one memory phase can be processed at a time. It results that the ℓ memory phases are executed sequentially. The length of the time interval $L_i^{holes}(t)$ during which *all cores* are busy executing E-phases is therefore upper-bounded by the sum of the length of the k E-phases running on the first core, minus the lengths of the ℓ memory phases executed on all the other cores. This length is further maximized if the k E-phases executing on core 1 are the k largest, and the memory phases executed on the other cores are the ℓ shortest. This intuition can be observed in the upper part of Fig. 5. In the figure, one can observe the largest k E-phases in green running on core 1, $(m-1)$ A- and R-phases running in parallel with each execution phase (in pink), and an upper bound $\hat{L}_i^{holes}(t)$ on $L_i^{holes}(t)$ represented as the difference between the length of the execution and memory phases. Formally, we have in the general case that

$$L_i^{holes}(t) \leq \hat{L}_i^{holes}(t) = \sum_{j=1}^{k} \overrightarrow{\mathcal{E}}^{(j)} - \sum_{j=1}^{p} \overleftarrow{\mathcal{A}}^{(j)} - \sum_{j=1}^{q} \overleftarrow{\mathcal{R}}^{(j)} \quad (5)$$

where $\overrightarrow{\mathcal{E}}$, $\overleftarrow{\mathcal{A}}$ and $\overleftarrow{\mathcal{R}}$ are, respectively, the set of all E, A and R-phases interfering with τ_i's execution. $\overrightarrow{\mathcal{E}}^{(j)}$ denotes the jth element of the set sorted in a *non-increasing* order, while $\overleftarrow{\mathcal{A}}^{(j)}$ is the jth element of the set sorted in a *non-decreasing* order

(note the direction of the arrow on top of the set). Therefore, Eq. (5) accounts for the k largest E-phases interfering with τ_i and the p and q shortest A and R-phases, respectively (with $\ell = p + q$ in the explanation above).

Even though Eq. (5) provides an upper bound on $L_i^{holes}(t)$, it is extremely pessimistic due to the fact that it neglects how different E-phases execute in parallel on different cores. Only the E-phases (conservatively assumed to be the k largest ones) running on the first core are considered when computing the bound.

A tighter bound on $L_i^{holes}(t)$ can be obtained by considering the E-phases scheduled on all cores. Assume that each core (and not only the first one) executes the same number k of E-phases, then each E-phase is preceded by an A-phase, and each A-phase is preceded by the R-phase of the previous jobs that executed on the same core (see the middle part of Fig. 5). Therefore, k A-phases and at least $(k-1)$ R-phases execute on each core. [3] Now, let us build an artificial schedule as shown on the middle part of Fig. 5 where the k longest E-phases execute on the first core, the k second largest E-phases execute on the second core, and so on and so forth. Similarly, the k shortest A-phases and the $(k-1)$ shortest R-phases execute on the mth core, the k second shortest A-phases and the $(k-1)$ second shortest R-phases execute on the $(m-1)^{\text{th}}$ core, etc. Then, the amount of time the E-phases on the first core do not overlap with memory phases and hence can participate to $L_i^{holes}(t)$ is given by Eq. (5) with $p = (m-1) \times k$ and $q = (m-1) \times (k-1)$ (i.e., the number of A and R-phases, respectively, executing on the other cores). That is, it is given by

$$\sum_{j=1}^{k} \overrightarrow{\mathcal{E}}^{(j)} - \sum_{j=1}^{(m-1)\times k} \overleftarrow{\mathcal{A}}^{(j)} - \sum_{j=1}^{(m-1)\times(k-1)} \overleftarrow{\mathcal{R}}^{(j)}$$

Similarly, the amount of time the E-phases on the second core do not overlap with memory phases and hence can participate to $L_i^{holes}(t)$ is given by

$$\sum_{j=k+1}^{2\times k} \overrightarrow{\mathcal{E}}^{(j)} - \sum_{j=1}^{(m-2)\times k} \overleftarrow{\mathcal{A}}^{(j)} - \sum_{j=1}^{(m-2)\times(k-1)} \overleftarrow{\mathcal{R}}^{(j)}$$

where $(m-2) \times k$ and $(m-2) \times (k-1)$ are the number of A and R-phases executing on cores 3 to m (see middle part of Fig. 5). Doing the same for each core and summing all those contributions, we get that the total time during which E-phases do not overlap with memory phases is upper bounded by

$$\sum_{j=1}^{m\times k} \overrightarrow{\mathcal{E}}^{(j)} - \sum_{p=0}^{(m-2)} (m-1-p) \times \Big(\sum_{j=1}^{k} \overleftarrow{\mathcal{A}}^{(j+p\times k)} + \sum_{j=1}^{k-1} \overleftarrow{\mathcal{R}}^{(j+p\times(k-1))} \Big) \quad (6)$$

The maximum amount of time the m cores are all simultaneously busy executing E-phases is thus upper-bounded by

[3]Without loss of generality, if $|\overrightarrow{\mathcal{E}}| \leq (k \times m)$, then the set $\overrightarrow{\mathcal{E}}$ is appended with zero-length E-phases such that $|\overrightarrow{\mathcal{E}}| = k \times m$. Similarly, zero-length A-phases and zero-length R-phases are appended to sets $\overleftarrow{\mathcal{A}}$ and $\overleftarrow{\mathcal{R}}$, respectively, until their cardinality equals $k \times m$.

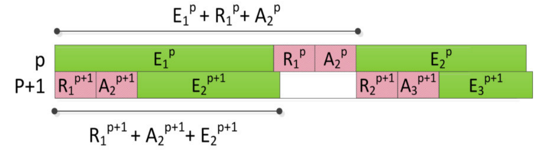


Fig. 6: Bus holes

the above equation divided by m (see lower part of Fig. 5). This gives us an upper-bound on $L_i^{holes}(t)$ as formalised in Theorem 3.

Theorem 3. *An upper bound on $L_i^{holes}(t)$ is given by*

$$L_i^{holes}(t) \leq \frac{1}{m} \times \max_{k\geq 1} \Bigg\{ \sum_{j=1}^{m\times k} \overrightarrow{\mathcal{E}}^{(j)} - \sum_{p=0}^{(m-2)} (m-1-p) \times \Big(\sum_{j=1}^{k} \overleftarrow{\mathcal{A}}^{(j+p\times k)} + \sum_{j=1}^{k-1} \overleftarrow{\mathcal{R}}^{(j+p\times(k-1))} \Big) \Bigg\} \quad (7)$$

Proof. Due to space limitation we only provide a proof sketch.

As explained above Eq. (6) provides an upper-bound on $L_i^{holes}(t)$ assuming that: (i) only the longest E-phases and the shortest A and R-phases are running, (ii) core p executes E, A and R-phases that are no smaller than those executed on core $(p+1)$ for all $p \in [1, m-1]$, and (iii) at least k A-phases, k E-phases and $(k-1)$ R-phases are executed on each core. An upper-bound on $L_i^{holes}(t)$ is thus found when Eq. (6) is maximized over k, which gives us Eq. (7). However, we still have to prove that the three assumptions hold.

Assumption (i) is quite obvious. If shorter E-phases are executed then the amount of time all cores simultaneously execute E-phases cannot increase. Similarly, if longer memory phases execute, then their overlap with E-phases can only increase, hence reducing the cumulative time all cores execute E-phases simultaneously.

Regarding Assumption (ii), in the schedule seen on the midle part of Fig. 5, one can see that if memory phases were swapped between cores (e.g., swapping A^5 and A^2), then the time during which memory phases would overlap with E-phases would increase and hence the length of bus holes would decrease. The shortest memory phases must therefore execute on the cores with the largest indexes. Similarly, if E-phases are swapped between cores (e.g., E^2 and E^6 in Fig. 5), then the amount of time all cores execute E-phases in parallel can only decrease. $L_i^{holes}(t)$ is thus maximized when the largest E-phases execute on the cores of the lowest indexes.

Finally, we prove Assumption (iii). That is, if core 1 execute k A-phases and k E-phases, and if core p executes E, A and R-phases that are no smaller than those executed on core $(p+1)$ for all $p \in [1, m-1]$, then at least k A-phases, k E-phases and $k-1$ R-phases are executed on each core.

The claim obviously holds for core 1 since each E-phase is followed by an R-phase. Hence, at least $k-1$ R-phases execute along with the k A- and E-phases on core 1. The proof for the other cores is by induction. That is, we prove that if core p executes at least k A-phases, k E-phases and $k-1$ R-phases, then core $p+1$ executes at least k A-phases, k E-phases and $k-1$ R-phases.

Let phases E_1^p, R_1^p, A_2^p and E_2^p be executed in a sequence on core p and similarly phases R_1^{p+1}, A_2^{p+1} and E_2^{p+1} be

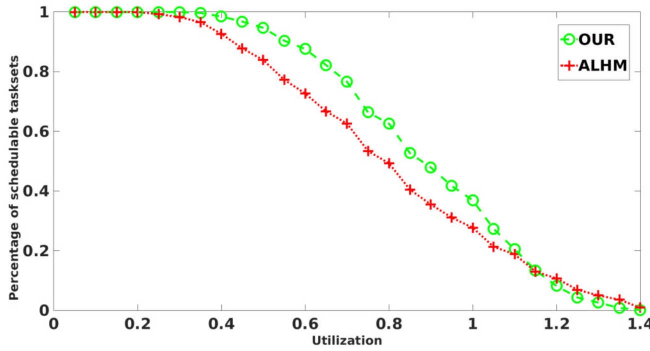


Fig. 8: % of schedulable tasksets per utilization for m=2

successively executed on core $p+1$ (see Fig. 6). Since all the E, A and R-phases executing on core $p+1$ are shorter than those executing on core p, we have that $R_1^{p+1}+A_2^{p+1}+E_2^{p+1} \leq E_1^p+R_1^p+A_2^p$. Therefore, as illustrated on Fig. 6, the E-phase E_2^{p+1} executed on core $p+1$ must complete before the second E-phase E_2^p starts executing on core p. Thus, there are at least as many E-phases executing on core $p+1$ as on core p. Since each E-phase is preceded by an A-phase and followed by an R-phase the number of A and R-phases on core $p+1$ also matches the number of phases on core p. This proves our claim. □

VIII. Results

In this section the approach presented in this paper is compared against the approach presented in [2], [11] using randomly synthetically generated task sets[4]. The generation parameters are detailed next.

The number of tasks per task set is set to $n = 5 \times m$ and the total utilization of each task set U_τ ranges from $[0.025 \times m, 0.7 \times m]$ in steps of $0.025 \times m$. UUnifast-Discard [14] is used to generate n utilization values such that $U_i \leq 1$ and $\sum_{i=1}^{n} U_i = U_\tau$.

To generate the period of each task T_i, a log-uniform distribution is used with values ranging within $[100, 1000]$. The tasks' execution times are calculated as $C_i = U_i \times T_i$. The generated tasks are assumed to have implicit deadlines and tasks' priorities are given by their periods following the Rate Monotonic approach, i.e., the lower the period the higher the priority.

Since each task is composed of memory phases ($M_i = A_i + R_i$) and execution phases (E_i), in the experiments the value for the memory phases was set to a percentage p of the execution time C_i of each task. The other $(1-p) \times C_i$ time units being assigned as the execution time of τ_i's E-phase, i.e., $E_i = (1-p) \times C_i$. The total memory phase value is equally divided between A and R-phases so that $A_i = R_i = \frac{p \times C_i}{2}$. By default, p is set to 0.1.

In all experiments, 1000 random tasks sets were generated for each plotted point. The percentage of task sets deemed schedulable by each analysis (the schedulability ratio) is used to compare the performances between approaches. In Fig. 7, the green line ('OUR') presents the results for the approach presented in this paper and the red line ('ALHM') presents the results for [2], [11].

The first set of experiments measured the percentage of task sets that are schedulable as a function of the task set total utilization. Fig. 7a shows the results for $m = 4$ cores considering the generation parameters described above. In the figure, one can observe that 'OUR' approach performs better than 'ALHM' resulting in an increase of around 10% (up to 15%) of the number of task sets deemed schedulable when the total utilization value varies between 0.7 and 1.5. An observed trend for both approaches is that task sets with half of the utilization of the platform are most likely not schedulable (for $m = 4$ task sets with $U \geq 2$ are not schedulable).

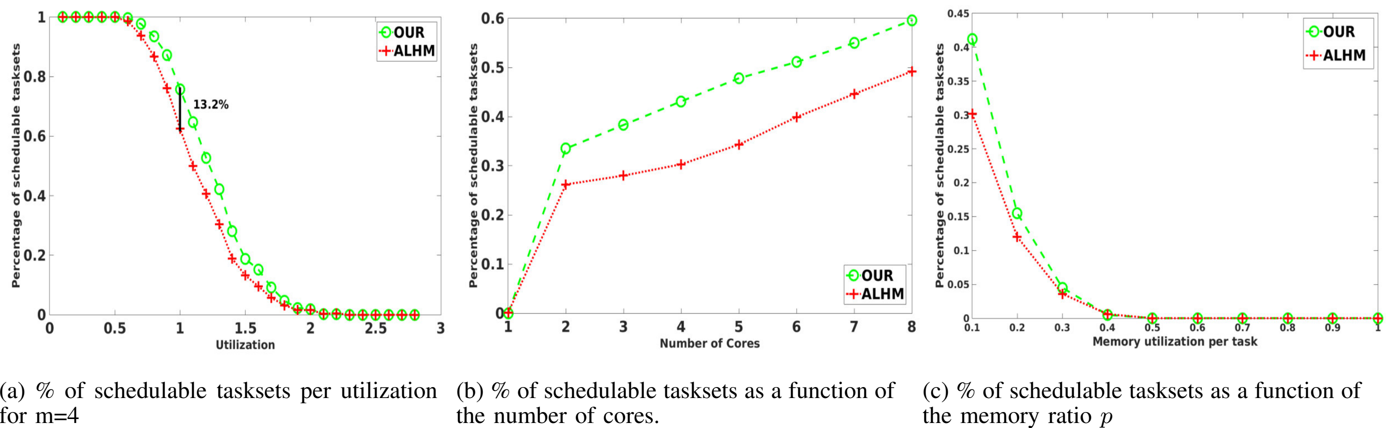
In the second set of experiments, depicted in Fig. 7b, the schedulability ratio is measured as a function of the number of cores, up to $m = 8$. In this experiment, the number of tasks per task set was set to $n = 10$ and the taskset utilization was fixed at $U = 1$. As a side note, if the number of tasks per task set varies as a function of the number of cores, a higher percentage of task sets would be schedulable in systems with a large number of cores. This behaviour occurs due to the decrease in the utilization per task, and consequently a decrease in the utilization of memory phases. Therefore, keeping a fixed number of tasks allows one to better observe the influence of the increase in the number of cores.

As it can be seen in Fig. 7b, both approaches cannot schedule any task set in a system with a single core (which is expected when the total utilization is 100% and fixed priority scheduling is used). But as the number of cores start to increase, the number of schedulable tasksets also increases. For the 3-phase model that means that more tasks can execute their E-phases in parallel thus decreasing their response-time when compared to a system with a lower number of cores. Note that the difference between 'ALHM' and 'OUR' remains more or less constant and around 10%.

In the third set of experiments, shown in Fig. 7c, the schedulability ratio is measured as a function of the ratio $p = \frac{M_i}{C_i}$. This experiment allows us to observe the influence of the bus on the schedulability of the system. In this experiment, the number of cores was set to $m = 4$, the task number $n = 10$ and the total utilization $U_\tau = 1.0$. The value of the memory ratio p varies in the interval $[0.1, 1]$ in increments of 0.1. As expected, increasing the memory utilization, decreases the percentage of schedulable task sets since the bus becomes the more and more loaded, hence increasing the access time to the memory. Another interesting aspect that can be observed is that after 40% of memory utilization both approaches perform almost exactly the same. These results are explained by the restrictions imposed by the bus on the execution of memory phases in order to avoid interference. In particular, after around 50% of memory utilization per task, bus interference dominates both approaches avoiding any of them to take advantage of parallel execution of E-phases.

Finally, one should note that the method presented in this paper does not dominate the analysis in [2], [11]. There exist task sets that are deemed unschedulable by our method but which are deemed schedulable by 'ALHM', as depicted in Fig. 8. This is usually the case when the interference on

[4]The authors in [2] compare their approach against a global non-preemptive approach in which tasks interfere with each other when accessing main memory. Since this global non-preemptive approach is outperformed by Alhammad's approach, we do not perform this comparison in this paper.



(a) % of schedulable tasksets per utilization for m=4

(b) % of schedulable tasksets as a function of the number of cores.

(c) % of schedulable tasksets as a function of the memory ratio p

Fig. 7: Simulation results

the cores is much more constraining than the interference on the bus. It is somewhat understandable since 'ALHM' tackles the problem from a core perspective while we tackle it from a bus perspective. Since their modelling of the interference on the cores is more accurate than ours, they may perform better in such situations. However, we believe that the bus will usually be the limiting factor in multicore systems. Therefore, improving the modelling of the interference on the bus should provide better results in most cases.

IX. Conclusion

This paper presents a schedulability test for the global fixed-priority scheduling of the 3-phase task model. Instead of analysing the schedulability of the system from a core's perspective as state-of-the-art approaches do, we analyse it from the perspective of the bus. In particular, the proposed approach computes an upper-bound on the length of intervals when all cores are busy with execution phases (the bus holes) and adds this length to the workload of a task due to memory phases. By looking at a problem window and analyzing the worst-case interfering workload on a task under analysis the schedulabily test is derived.

The proposed approach is evaluated by comparing it against a state-of-art schedulability test. The results show an increase on the schedulability ratio over the state-of-the-art of around 10% in average and up to 15% in some cases.

Future work includes the development of methods that compute tighter upper-bounds on the length of the bus holes so as to improve the accuracy of the test. A variation of this test to compute worst-case response times may also be interesting for the development and analysis of real-time systems running on multicore platforms.

Acknowledgments

This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the Portugal2020 Program, within the CISTER Research Unit (CEC/04234); also by the European Union under the Seventh Framework Programme (FP7/2007-2013), grant agreement nº 611016 (P-SOCRATES); also by FCT/MEC and the ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH / BD / 88834 / 2012.

References

[1] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS '11, 2011, pp. 269–279.

[2] A. Alhammad and R. Pellizzoni, "Schedulability analysis of global memory-predictable scheduling," in *Proceedings of the 14th International Conference on Embedded Software*, ser. EMSOFT '14, 2014, pp. 20:1–20:10.

[3] A. Schranzhofer, J. J. Chen, and L. Thiele, "Timing analysis for tdma arbitration in resource sharing systems," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010, pp. 215–224.

[4] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10, 2010, pp. 741–746.

[5] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, "Memory-centric scheduling for multicore hard real-time systems," *Real-Time Systems*, vol. 48, no. 6, pp. 681–715, 2012.

[6] G. Durrieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch, "Predictable flight management system implementation on a multicore processor," in *Embedded Real Time Software and Systems (ERTS'14)*, 2014.

[7] S. Girbal, D. Gracia Pérez, J. Le Rhun, M. Faugère, C. Pagetti, and G. Durrieu, "A complete tool-chain for an interference-free deployment of avionic applications on multi-core systems," in *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, 2015, pp. 1–13.

[8] C. Maia, L. Nogueira, L. M. Pinho, and D. G. Pérez, "A closer look into the aer model," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–8.

[9] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, and T. Nolte, "Contention-free execution of automotive applications on a clustered many-core platform," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, 2016, pp. 14–24.

[10] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo, "A real-time scratchpad-centric os for multi-core embedded systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–11.

[11] A. Alhammad, "Memory effcient scheduling for multicore real-time systems," Ph.D. dissertation, University of Waterloo, 2016.

[12] T. P. Baker, "Multiprocessor edf and deadline monotonic schedulability analysis," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, ser. RTSS '03, 2003, pp. 120–129.

[13] N. Guan, W. Yi, Z. Gu, Q. Deng, and G. Yu, "New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms," in *2008 Real-Time Systems Symposium*, 2008, pp. 137–146.

[14] R. I. Davis and A. Burns, "Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," *Real-Time Systems*, vol. 47, no. 1, pp. 1–40, 2011.