Modeling multicore contention under cyclic executive scheduling

Palomo Teruel, X.

Award date:
2018

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

Modeling Multicore Contention under Cyclic Executive Scheduling

Master Thesis

Xavier Palomo Teruel

Supervisors:

Technische Universiteit Eindhoven:
Dr. Reinder J. Bril

Barcelona Supercomputing Center:
Dr. Francisco Javier Cazorla Almeida
Dr. Enrico Mezzetti

Eindhoven, The Netherlands, August 2018

Acknowledgements

This Thesis was carried out as an internship assignment at the Barcelona Supercomputing Center, from February to August 2018. Having done so would have been much more difficult without the help from many people around me.

Firstly and foremost, I owe my deepest gratitude to Dr. Enrico Mezzetti and Dr. Francisco Cazorla, my supervisors at Barcelona Supercomputing Center, for the time you invested on me since the very first day. Your patience and expert guidance helped me move forward step by step and keep me motivated. Thank you for the incredible good influence you have had on me, both at technical and human levels. Above everything else, thank you for always being so nice and ready to help.

I am grateful to my University supervisor, Dr. Reinder J. Bril. I highly appreciate your comments and improvements suggestions, which have undoubtedly increased the quality of this Thesis. You read my drafts several times with scrupulous attention to every detail, raising critical and right-to-the-point questions and comments.

I would like to express my sincere appreciation to Barcelona Supercomputing Center, and to the Computer Architecture and Operating Systems department in particular. It has been a great pleasure to work with a group of so knowledgeable and interesting people like you every day. The work environment you provide is everything one could ask for.

I would like to thank the EIT Digital Master School too, for providing a unique opportunity to explore science and entrepreneurship in conjunction, and offer the chance to meet brilliant people from all over the globe, who have made of this journey an even more amazing experience. Without any doubt, these two years at the Technische Universität Berlin, and at the Technische Universiteit Eindhoven, will remain as an indelible vital memory for me.

Thanks to my parents, Elvira and Angel, and my partner Sara, who have always supported me, encouraged me to study what passionate me the most, and helped me choose the best decisions unconditionally.

Xavier Palomo Teruel
Eindhoven, the Netherlands
August 2018

Abstract

Commercially available Off The Shelf (COTS) multicores have been assessed as the baseline computing platform even in the most conservative real-time domains. Multicore contention arising on shared hardware resources, with its circular dependence with scheduling, is among the most challenging issues that require urgent attention before multicores can be fully embraced for real-time computing. In the context of statically scheduled systems, still the most used scheduling approach in real-time industries, this Thesis proposes two different formulations for computing the worst-case contention delay potentially suffered by a task on account of the interference potentially incurred when sending requests to shared resources. These two models provide tight contention delay upper bounds under reasonable assumptions, by capturing the interdependence between timing interference of conflicting requests, issued in parallel by other cores, and the identification of the particular set of tasks co-running on those cores. This work shows, via extensive evaluation, that jointly accounting for worst-case task overlapping and request distribution scenarios provides tighter contention bounds when compared to state-of-the-art solutions. Furthermore, a fully automated framework for the implementation and execution of both methodologies has been developed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This work fits in the scope of timing analysis of critical embedded real-time systems running on Commercially available Off The Shelf (COTS) multicore platforms. In particular, we deal with the effect of inter-core (timing) interference stemming in multicore systems when applications running on distinct cores concurrently attempt to access a shared hardware resource such as a bus. We focus on statically scheduled, cyclic executive systems, for their widespread adoption in the most conservative critical embedded real-time domains such as avionics and aerospace, and propose two analytical methods to derive safe and tight bounds to the worst-case delay suffered by a given application as a result of contention on shared hardware resources. Further, to demonstrate the practical applicability of the proposed methods, we present a proof of concept evaluation of the end-to-end analysis process on top of a concrete FPGA implementation of a reference multicore design in the aerospace domain.

In this chapter, we first provide the context and background on critical embedded real-time systems, and introduce the problem we address in this work. Second we present our research objectives and approach, and anticipate what will be the main contributions of this Thesis. Finally, we provide an outline of the document organization.

1.1 Context and Background

The umbrella term of critical embedded systems is used to address a multifaceted set of embedded systems, under a variety of industrial domains, that share common requirements in terms of safety, security, finance or a combination thereof. Then, critical real-time embedded systems (CRTES) identify the subset of those systems that are also meant to meet stringent timing requirement: correctness and appropriateness of such systems is not only limited to the functional aspects of computation, but also depend on the timeliness of the computation, where soft or hard timing constraints are typically defined. Interestingly and a bit unexpectedly, critical embedded real-time systems are already a pervasive aspect in our lives [2].

CRTES are normally deployed in different application domains (e.g., automotive, avion-

ics, railways, medical systems, etc.). Depending on the application domain, a misbehavior of a critical embedded real-time system may lead to unacceptable consequences in terms, for example, of human life or huge economic losses. For this reason, domain-specific qualification and certification standards identify the criteria which should be met in the development of CRTES. This is the case, for example, of DO-178B [57] and ISO26262 [35], respectively for avionics and automotive applications. Despite the inherent differences among CRTES certification standards, the stringent requirements they set on verification and validation (V&V) activities end up burning off a large share of the time and resources allocated to the software development process. With respect to timing correctness, V&V standards demand for the provision of reliable guarantees on the timing behavior of software functionalities. Timing analysis techniques are typically deployed to derive Worst-Case Execution Time (WCET) bounds for each functions in a system. WCET bounds are later fed to schedulability analysis to provide guarantees that a given set of functionalities can be effectively scheduled on a given system without time overruns[1].

CRTES domains are transversely witnessing an unprecedented growth in performance needs and computational complexity of their respective embedded applications, to support advanced, next-generation functionalities such as, for example, autonomous driving systems [14, 34] in the automotive domain. In order to meet these new emerging requirements, even the most conservative CRTES are being forced to move from relatively simple (single-core) platforms to cutting-edge modern heterogeneous multicore and manycore systems [58], with scores of advanced acceleration features, as a means to provide the necessary computational power with low Size, Weight and Power (SWaP) solutions. In particular, CRTES are considering the adoption of COTS hardware platforms as the reference solution. Besides performance, COTS are indeed considered to be able to provide better cost, flexibility and time to market [8, 4, 50] than custom hardware platforms.

1.2 Problem Statement

Since CRTES demand for reliable timing bounds in order to guarantee that hard systems deadlines are met, the concept of *time predictability* becomes paramount. In reason of their real-time requirements, timing to become a critical non-functional property: being able to provide strong timing guarantee is thus a fundamental concern in critical real-time systems. Several approaches have been proposed to derive trustworthy guarantees on the timely execution of software functions [65]. Although not systematically, the same techniques have been embraced by the domain-specific certification standards. In the case of single core systems, timing analysis basically boils down to bound the execution time of a task or program executing in isolation on a given hardware platform, thus comprising some form of low-level architectural analysis and high-level program characterization. The obtained bounds are then used to properly schedule the task set so that all jobs can meet their deadlines, provided tasks' timing requirements and dependency information.

[1] Deadline misses are to some extent deemed as acceptable in *soft* and *firm* real-time systems.

However, when it comes to multicore architectures, another factor plays a fundamental role: *contention* on shared hardware resources.

Cores in a multi or manycore system necessarily share some common hardware resources (e.g., part of the memory hierarchy, the interconnect, I/O etc.). Access and usage of those resources must be properly managed. Basically, since multiple cores can potentially (and concurrently) access the same hardware resource, such as the memory controller or the bus, it may be the case that upon a request the addressed resource may not be available because it is already serving requests from other cores. The requesting core must then wait for the other core to relinquish the resource, which is typically arbitrated by a (predictable) protocol. The higher the number of cores, the higher the potential contention effects, which jeopardizes predictability. In fact, providing strong performance guarantees on top of COTS hardware is complicated by the fact that contention can inordinately affect tasks' execution time, which can cause a severe increase of the response time of a program [53, 37]. While it is recommended to counter and mitigate all inter-core *interference channels* [23], so far no COTS hardware exists that allows excluding all sources of interference [64]. Therefore, if timing interference across cores is generally inescapable, its effect on a task execution time must be safely and precisely accounted for, to guarantee that timing budgets assumed and enforced by the scheduling framework are never exceeded at run time.

Several algorithms and methodologies have been proposed to derive bounds to the worst-case inter-core interference [21, 18, 26], typically referred to as Worst-case Contention Delay (WCD), possibly incurred by a task. Factoring WCD bounds in the apportionment of the time budgets allows guaranteeing that tasks will meet their deadlines even in the presence of contention. However, contention effects are difficult to characterize precisely as they depend on when (and how often) the program and its co-runners are accessing a given shared resource. Approaches for the computation of the WCD are forced to build on conservative assumptions, which often turn out to lead to overly-pessimistic WCD bounds. Pessimism, in turn, translates into a reduction of the system's guaranteed performance, which is pretty unwanted in embedded systems.

This Thesis aims at contributing to the following research question:

> *Can we derive safe and tighter bounds on the worst-case contention delay on cyclic-executive, time-triggered multicore platforms with respect to the state-of-the-art works?*

1.3 Goals

Several approaches have been proposed for the characterization of inter-core interference on hardware resources. Some of them aim at exploiting hardware or software level mechanisms to control [39, 66, 50] or even to avoid altogether [53, 52, 12, 17, 20] the effects of contention. While provably effective, these approaches typically build on some assumptions on hardware, RTOS support, and/or application characteristics (e.g., phased execution [53]) that cannot always be granted to hold in practice, especially in COTS platforms. When

none of the above solutions can be applied, the only practical way to provide trustworthy performance guarantees under multicore execution, remains that of accounting for the worst-case contention delay [62, 26]. An important aspect in bounding contention effect is that the computed bounds should be as tight as possible, which in turn depends on the amount of information available to the analysis.

The more information we have access to, the better we can schedule our system. In that line, the goal of this Thesis is to define a methodology to compute reliable and tight WCD bounds for tasks executing on COTS platforms. Our method will be exploiting all the available information on the task set as well as on the platform where tasks are meant to execute, to tighten up the WCD bounds and avoid wasting computational resources.

Being able to compute tight WCD bounds is fundamental to make use of the available resources and to avoid a costly over-dimensioning of the system. The unused (wasted) computational power could be used, for example, to accommodate more functions (contributing to reduce hardware procurement costs to allocate a fixed set of software functionalities); to perform best-effort activities; to reduce the energy profile; or even to deploy an imprecise-computation model [44].

Moreover, this work focuses on time-triggered cyclic executive systems – a well-known case of statically-scheduled systems – where wasting computational resources is especially unwelcome since resources can not be reclaimed at run-time. Static scheduling is a prevalent solution in real-time software specifications (e.g., AUTOSAR RTE [15], ARINC [13], ARINC in Space [30]) in critical embedded real-time system domains, as a means to guarantee determinism, predictability and isolation, even more under multicore execution. Since these systems build on a static table-driven schedule, unused computational resources cannot be dynamically reclaimed by the scheduling algorithm. The ability to derive tight contention bounds is, therefore, of utmost importance in those systems.

Some conservative assumptions leading to pessimism are somehow unavoidable: for instance, it is virtually impossible to model precisely *how* requests from different cores align in the access to hardware shared resources, and analysis approaches are forced to assume worst-case alignments. Some other aspects, instead, as how tasks may overlap on different cores, can be modeled to a certain degree. We contend that tight contention bounds can be obtained by combining information on the activity on shared resources of both the program or task under analysis and a concrete set of potential co-runners. In this view, the goal of this Thesis is to present analytical solutions and industrially-practical approaches to derive reliable bounds to the execution time of a task on contention-prone multicore COTS platforms. As a relevant concern, the computed bounds shall be tighter when compared to other state-of-the-art solutions.

1.4 Approach

In order to derive tight WCD bounds, we will build on combining task-level information on resource accesses and system-level information on tasks execution partially or entirely overlapping (i.e., executing at the same time on different cores). In this respect, we observe

that a circular dependence exist between the overlapping among tasks and the worst-case execution time inflation caused by contention effects on those tasks. Tight WCD bounds can only be obtained by effectively capturing such circular dependence.

Figures 1.1 and 1.2 provide an illustrative example of circular dependence: the potential contention depends directly on the overlapping of tasks. Contenders of a certain task under analysis, in turn, depend on the WCD accounted for in all the predecessor tasks in the same core. Following the simple example in Figures 1.1 and 1.2, we can see that task *0_1* (representing the first task allocated to Core 0) would only run in parallel with task *1_1* if no contention exists (Figure 1.1). However, when we inflate the timing budget for those tasks to account for the potential contention they may suffer, task *0_1* could also partially overlap with task *1_2*, which means that extra contention effect might be incurred (Figure 1.2). On the other hand, still in Figure 1.2, task *1_1* would no longer run in parallel with task *0_2*, causing a reduction in the worst-case contention potentially suffered by *1_1*, when compared to the first scenario.

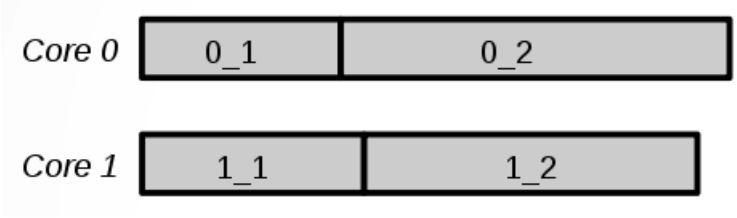


Figure 1.1: Circular dependence: without contention

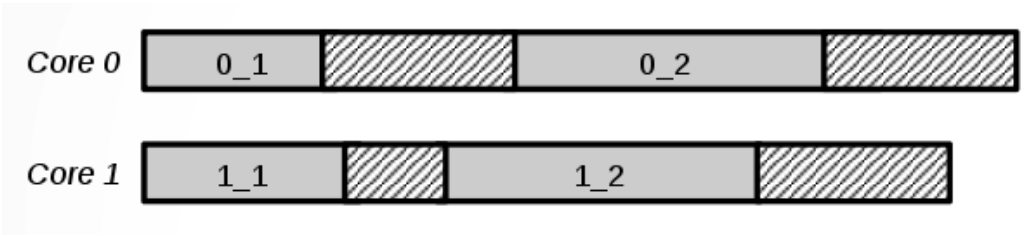


Figure 1.2: Circular dependence: with contention

From a system-level perspective, the fact that contention effects may increase or decrease makes it complex to statically analyze the WCD, which ends up being able to find the overlapping, within all the feasible scenarios, causing the worst-case contention effects across all tasks in all cores. Following a brute force approach, all possible overlapping scenarios should be considered in order to precisely know the worst-case system-level contention effects. Despite being safe, such approach will clearly suffer from scalability and even computational issues.

The guideline that this work has followed, has been to firstly derive contention information from resource access profiles of tasks in isolation, and later use the obtained information to build an analytical model of the contention effects. The computed WCD

bounds are then used to derive time budgets that safely guarantee all tasks to meet their deadline, as tightly as possible. The analytical model presented in this Thesis is tailored to static, time triggered (off-line) non-preemptive task sets, running on a multicore platform. Notably, accesses may exhibit different latencies depending on the target resource to be accessed, and the operation to be performed (access type): while the number of contention events is bounded by the number of accesses performed by the task under analysis, the contention effects (incurred delays) are determined by the access type of the co-runners.

After an accurate analysis of the benefits and limitations of different methods in the state-of-the-art, two different approaches have been derived in order to capture access-type-aware WCD bounds. Both methods will be described in-depth in the core chapter of this document. As it will be evident, the presented analysis approaches are pretty different (and experimental results are as well). The methods represent two fundamentally different approaches: they target task-level and system-level WCD computation and can be alternatively adopted, depending on the different system and application needs.

A. *Task-Level Guarantee → Iterative Script*: We propose an iterative approach to successfully identify a safe and tight schedule (i.e., set of triggering times) for each task allocated to each core. Triggering time for a given task guarantees that its predecessor tasks will have completed their execution, even under the worst possible scenario in terms of contention effects. The worst-case contention delay is computed by considering access number and types of a given task and its co-runners. The approach provides safe timing budgets at single task-level. However, this comes at the cost of incurring some sources of pessimism at *makespan* (global core-level schedule) level, which is instead avoided by the alternative approach.

B. *MIF-Level Guarantee → ILP Formulation*: We propose an Integer Linear Programming (ILP) formulation to model all possible overlapping scenarios between tasks within a time frame, and to computes the worst case contention effects by considering all possible alignments of accesses on the shared resources. The worst-case contention delay is, therefore, computed similarly to the previous approach, but is capable of matching precisely the victim and the origin of each conflicting access. This model, therefore, exploits full task-level and system-level information, resulting in accurate WCD bounds. Further precise (and reduced) WCD bounds could only be obtained under specific scenarios, by leveraging on additional details or constraints on access distribution.

Our methodologies are able to adapt to different scenario and to compute tight WCD bounds, without requiring major modifications. The proposed contention models are extensively evaluated on randomly generated synthetic task sets, showing that the derived contention bounds support significantly higher overall utilization within a given time frame, when compared to state-of-the-art approaches [59, 26], thus allowing to deploy more functionalities while keeping strong performance guarantees. The flexibility of our approaches has been also assessed, showing that they can be easily adapted to model a wide range of

assumptions and scenarios affecting the WCD computation, such as the already mentioned distribution of accesses. Finally, as a proof-of-concept, the industrial applicability of the presented approaches is proven, by evaluating their performance when modeling real task sets running on real hardware, and using information made available by the latter. Our results proved that the proposed methods are realistically applicable and do provide safe and tight bounds.

1.5 Contributions

The contributions of this Thesis can be summarized as follows:

1. A study and review of the state-of-the-art of real-time multicore timing analysis especially with regard to inter-core contention modeling;
2. An iterative method to compute WCD bounds with task-level guarantees;
3. An alternative ILP-based approach exploiting system-level information to derive tighter WCD bounds;
4. A fully automated framework for the implementation and execution of both approaches;
5. A proof-of-concept application of the proposed methodologies on real hardware, to asses their performance and industrial applicability.

1.6 Thesis Overview

The remainder of this Thesis is structured as follows: Chapter 2 reviews and analyzes the related works on multicore contention modeling, and emphasis is made on how we differentiate with them. Chapter 3 describes our assumptions, introduces the framework in which this work builds on, and introduces the formal notation used to describe our analytical models. Chapter 4 presents and discusses both our solutions, which are later extensively evaluated and compared against other existing approaches in Chapter 5. Finally, Chapter 6 draws the conclusions and briefly discusses the next steps to potentially refine and extend the line of work defined by this Thesis.

Chapter 2

State-of-the-Art

The ever increasingly transition from single to multicore systems in critical embedded real-time domains has disruptive effects on the consolidated practice for Verification and Validation (V&V) in general, and timing analysis in particular. Several efforts have been made in the last decades to capture the effect of hardware resource sharing in multicore systems [31], which has been unequivocally identified as the main source of timing interference. In this work, we do not consider the interference stemming from software resource sharing and inter-core synchronization protocols, for which analyzable protocols have already been devised (e.g., [22]).

As part of this work, we survey the scientific literature studying the effects of inter-core timing interference caused by contention on shared hardware resources. In this chapter, we present the state-of-the-art on contention analysis in multicore platforms, mainly focusing on a shared bus as the most common and well-studied source of contention. Proposed approaches have been building on either precisely analyzing the effects of contention or attempting to control or remove them altogether. In the following we briefly review the most relevant techniques, especially in relation to our approaches. We will first define interference and the contention delay that it produces, then we will review works which have addressed approaches to remove or limit the contention effects. Approaches which do not aim at reducing contention but quantifying it will be reviewed later. After that, we will go over similar works but which build on different scheduling assumptions, others that enforce quotas for shared resources among tasks, and finally, how other works have made use of performance monitoring counters as well.

2.1 Inter-Core Interference and Contention Delay

In order to better understand the focus of this work, we classify inter-core timing interference as direct and indirect. With direct interference we address the delay a task τ_i may suffer when accessing a shared hardware resource, because some other task τ_j executing on another core is using that resource (or in any case, will be granted access to it before τ_i).

Direct inter-core interference is therefore capturing the effects of contention when accessing a resource. When otherwise not specified, we will refer to this type of interference, and the contention delay accounted will only be produced by direct interference.

However, inter-core interference is not limited to contention effects but may include any other type of *indirect* interference. For example, a co-runner task may cause the eviction of useful cache content from a shared cache, which in turn will cause the analyzed task to incur a number of additional cache misses (with respect to execution in isolation). Indirect interference is indeed a relevant problem in some types of real-time systems [24]. Precisely characterizing the effects of indirect interference, however, is a harsh challenge. Interference cannot be normally bounded without resorting to strong conservative assumptions, leading to a large amount of pessimism in the results [11]. In order to avoid overly-pessimistic bounds, and to better meet the isolation requirements arising in modern, complex mixed-criticality systems, some core-local hardware or software resource partitioning mechanism is typically deployed.

Most of the works on bounding the effect of contention are building on the existence of a set of core-local resources (e.g., L1 cache, scratchpads, etc.) or on having some segregation mechanism in place (e.g., L2 partitioning). Under these assumptions it is possible to exclude the sources on indirect interference and to restrict or control the effects of contention.

2.2 Controlling or Removing Contention Effects

In the present literature we can find a strong trend in trying to reduce or even to entirely avoid inter-core interference.

Several works build on the PRedictable Execution Model (PREM) [52] model (read, execute and write phases) assumption as a basis to avert (or at least shorten) bus accesses of different tasks happening at the same instant. The underlying assumption in this model is that the instants at which tasks access memory can be clearly predicted and isolated into phases. Furthermore, it assumes that these accesses are not evenly distributed along the task execution, but instead they happen in bursts. It assumes a first stage in which tasks read data from memory, followed by an execution phase and ending up again with writes to the memory. This is generally made possible by loading the task working set into local scratchpad memories, and assuming that the platform supports some degree of resource partitioning.

If all the above assumptions are met, the contention effect can be reduced by aligning the tasks phases in such a way that read and write phases are not executed at the same time on different cores, i.e. a task performs its bus accesses during the computation time of its contenders. This model usually works in conjunction with task-to-core mapping objectives.

In the same way authors in [20] try to avoid the overlapping of task phases with high memory exchange rates. Blagodurov *et al.* [21] suggest a task-to-core mapping algorithm based on several heuristics and classifying tasks according to their miss rates, avoiding

tasks with high cumulative miss rates to run together. In the same line, [40] propose an algorithm based on bank partitioning and mapping of memory-intensive tasks to the same core. By partitioning the banks, the contention can potentially be reduced, since each core accesses different memory regions without interfering others. We mainly differentiate from these works in that we do not necessarily assume that isolation of phases will be possible, and moreover we do not address the tasks-to-core mapping, although our models can be adjusted to support these assumptions.

A scheduling framework on avionics multicore platforms is presented in [48]. Authors characterize tasks, identifying and isolating mandatory-to-execute parts from optional or less critical ones (which can be delayed). A limited preemptive model is then suggested, which adopts a Single Core Equivalence (SCE) technique, mitigating overhead. The SCE can be achieved thanks to a scheduler supporting temporal and spatial partitioning, enabling job skipping which allows the less critical identified partitions to have less stringent worst-case performance requirements. Our approaches do not build on a preemptive model, hence an identification of critical parts within tasks is not addressed.

The work in [47] addresses parallel applications (still assuming read-execute-write semantics) where task-to-core mapping and schedule are statically defined, and aims at introducing additional slack time in the schedule to reduce resource contention. While sharing some similarities with this work, more concretely with the iterative approach, this method builds on pessimistic task-level bounds, especially when compared against the ILP formulation for which no pessimism on task-level is incurred. Further, it does not support pairwise mapping of different request types.

Authors in [39] focus on DRAM and memory controller operations to bound the delay potentially suffered on memory accesses, and propose bank partitioning as a way to reduce the amount of inter-core interference. Beside the differences coming from the memory-controller perspective, we share some similarities with this work, mainly with regard to pairing accesses, as will be discussed in depth in upcoming chapters: authors take the minimum between the overhead based on a task under analysis' own accesses and the overhead from other tasks interfering with it.

Although aiming at reducing the possible contention is clearly a very reasonable call, it can lead to a reduction in flexibility as it relies on strong assumptions on applications semantics. Typically, tasks accesses' collision avoidance builds on a phased task model, which is not always possible to sustain. For that reason, our work focuses on those cases in which this is not possible (for instance, in legacy systems), or, even if it is, when we need to quantify the maximum contention delay that the system can suffer.

2.3 Accounting for the Worst-Case Contention Delay

Several approaches have been attempting to analyze and derive tight bounds on the WCD under several different assumptions. In this section, we will review some of these works, with special emphasis (for its commonly used, and similarity to one of our approaches) to

two trends for which a separate subsection will be dedicated.

The work in [61] operates exclusively at task level, meaning that the authors consider the worst alignment for each task in order to compute the WCD, but not taking into account the mapping of tasks to cores, hence not setting the scope to the overall system. They, therefore, do not consider the actual real (or potentially possible) alignment of co-runners. Moreover, the authors are forced to assume the largest latency request of contenders at each point, since every running task of the system (on other cores) is susceptible to be a contender. This, in turn, is equivalent to consider the type of bus access incurring the worst latency and hence adding considerably extra pessimism. These assumptions result in a loose WCD calculation. We will use this solution as a baseline for comparison in our experiments. Further details will be provided in Chapter 5, prior to the comparison.

Some early works focused on bounding the number of requests to a shared hardware resource for each single task in isolation: assuming uniform distribution of accesses, with a minimum distance between consecutive accesses [59], or assuming dedicated task phases [62]. An upper bound to the interference suffered by a task is determined by summing up the number of accesses from all potential co-runners. The interference bound can be tightened by limiting the analysis to co-runners that may actually overlap with the task under analysis. This observation is exploited in [25, 26], where an upper bound to the effect of contention suffered by a task is derived from an analysis of the maximum number of bus requests, issued from the other cores in a given interval. Later on, this information is used to compute a bound to the effects of contention on a task scheduled on another core. The computation of the contention effect is finally embedded in the standard response time analysis. As a common trait in these approaches, contention analysis is primarily focused on the activity of co-runners, rather than on the computation of co-runners of the task under analysis.

On the other hand, [27] does consider information on both the contender and the contending tasks, but does not assume multiple bus request types, resulting in less tight WCD bounds. Each access has to be assumed to entail the longest latency possible in the system. Besides, the way the authors compute the upper-bound on the increase in execution time of a task under analysis (by considering the maximum number of requests it can encounter), holds some similarities with the hereby suggested iterative approach. However, we also exploit the fact that a tighter bound to the number of resource requests that may potentially incur contention can be obtained by considering both, task under analysis' and co-runners' requests. Both the iterative and the ILP approaches' bounds will be compared to this work, to show how considering tasks' bus requests can tighten the results.

In our evaluation we will also compare with [62] and [52] as examples of approaches computing the WCD under the assumptions of task phases. In [52], authors propose enforcing a coscheduling mechanism to resolve the contention for accessing shared resources at high-level under the PREM assumption. On the other hand, in [62] authors consider tasks that are constituted by superblocks, which are assigned to processing elements and

run either time triggered or sequentially. Superblocks may end up being an example of dedicated phases, which will be the case for our comparison.

2.3.1 Modeling the distributions of accesses to the shared resource

Apart from the number, kind, and latency of the interfering and interfered accesses, it is also critical to determine *when* these accesses are occurring. If we have information regarding how or when accesses are happening, we can try to come up with a favorable alignment and mapping of tasks to cores. What is more, if we have specific information, we are not forced to assume the worst alignment scenario possible, hence leading to less pessimistic assumptions on conflicts.

Unfortunately, it is practically unfeasible to precisely obtain this information, being a generally unknown factor and resulting into an important source of pessimism. Still, different assumptions have been made with the objective of minimizing this pessimism.

As we already discussed in Section 2.2, some approaches rely on an execution model where tasks execution is split into memory and computation phases, where accesses to shared resources (i.e., memory system) exclusively happen during memory phases. This is the assumption made by many works [53, 52, 12, 17, 20] that addressed the task-to-core mapping in a more optimal manner when compared to full task considerations. In the same line goes the work of Rosen et al. [55] who measured the cache misses effects on the bus, assuming memory accesses occurring only at the beginning and end of tasks. Tasks are, therefore, scheduled in a way to avoid overlapping of memory phases.

Other approaches assume that accesses that tasks perform to shared resources are evenly distributed over task execution [39, 61]. In general this is not the case, and is hard to validate.

Although phased execution is a reasonable assumption in parallel applications, it lacks of generality, and clashes with the use of legacy code. While no assumption on access distribution is made, both of the presented approaches are flexible to model different application characteristics and to apply contention analysis at different granularity levels (i.e., full tasks, task sections or phases, etc.) as will be briefly explored as part of the proposed evaluation. In other words, resource requests can happen at any time throughout tasks' execution: either evenly distributed, sparsely or in bursts.

2.3.2 Using ILP to bound contention effects

The use of Integer Linear Programming (ILP) has been extensively explored by the research community in timing analysis. The generic goal which ILP seeks is to calculate an optimal solution given a set of problem constraints. In this subsection we are going to explore some of the relevant purposes with regard to WCD computation and task-to-

core mapping (still focused on contention) for which an ILP formulation has been proposed.

A first set of works try to compute the best task-to-core mapping in order to minimize contention. They do so by avoiding memory-intensive (phases of) tasks to run at the same time. An example of such case is the work in [17], which addresses non-independent runnables that can share data, and makes use of an ILP formulation to calculate the most optimal instants for tasks to access memory, building on phased execution (as in [12]). The pursuit of a contention-free environment relies on a time-triggered schedule with bank privatization, and performing an exploration of the most optimal time-triggered schedule and tasks' accesses to the shared memory.

Authors in [47] focus on parallel computing applications, with programs modeled as Directed Acyclic Graphs (DAGs), also within a time triggered model and with a round-robin bus. They use ILP to optimally insert slack time between tasks, claiming to offer an improvement of up to 20% in the most memory intensive scenarios. Our ILP formulation differs from these works in that we do not alter the execution by introducing slack time between tasks (result of pre-defining the triggering time), nor do we influence or force the instants in which tasks must perform their accesses to the shared resource.

Similarly, [56] proposes contention-aware time-triggered scheduling strategies using some knowledge on the application's structure to find a schedule that minimizes contention and so the overall makespan of the schedule under the read-execute-write semantics assumption as well. They use ILP to calculate the most optimal schedule, avoiding bad task alignments. This is enforced by setting statically the triggering instants. Although using ILP and forcing triggering times is similar to our iterative approach, we differentiate from this approach in that we do not seek for optimal schedules and, instead, we assume an already given task-to-core mapping.

Alternatively, other works have also relied on the use of ILP for the characterization of contention effects, especially tailored to domain-specific platforms in the automotive domain [28].

The ILP model presented in [28] focuses on the contention effect between pairs of tasks but does not consider task overlapping, i.e. exploring the circular dependence between contention and tasks alignment as we do. In [38], the authors also use ILP to derive a WCD bound in static time triggered systems, although they only model instruction caches assuming that data accesses occur via a different bus, thus not interfering with the instruction accesses and hence not considering their interference.

2.4 Contention Analysis and Scheduling Assumptions

We can further classify contention analysis approaches with respect to the underlying scheduling model. Dynamically scheduled systems, are those that rely on some kind of run-time decision making, while statically scheduled (also known as off-line) systems do not base their schedule on any dynamic factor, and, instead, rely on a pre-established

schedule, defined before the beginning of their execution.

Some works [52, 53, 25, 26] build on dynamic (run-time) scheduling, checking that indeed critical tasks will have enough availability to complete their execution. Inam *et al.* implemented Behnam's proposal [33, 18] consisting of a Multi-Resource Server (MRS) for statically partitioned multi-core real-time systems. This work considers other sources of contention such as tasks competing for CPU-bandwidth, and employs a Fixed Priority Pre-emptive Scheduling (FPPS) policy for both node scheduling and server scheduling. In [50], instead, run-time monitoring is used to ensure that co-runners' requests stay under a given utilization threshold determined at analysis time.

Even though the timing analysis scope in these works is different from the one we address in this thesis. The increase in execution time addressed in these works is mainly produced by run-time monitoring, the overhead during the tasks' preemption, and the context switching effort.

On the other hand, static (off-line) scheduling aims at providing a schedule before starting the execution of the system [47, 56, 62]. The work in [47] assumes preemptive scheduling and makes no work-conserving assumption to preserve analysis results at run-time, whilst [62] considers super-blocks of tasks that are executed sequentially or according to a time-triggered schedule under different paradigms: generic access model and dedicated phase (e.g., read-execute-write). They evaluate them and provide empirical evidence that dedicated phases is leading to better response times than addressing tasks as a whole.

Furthermore, we can distinguish tasks' triggering nature between event-triggered and time-triggered tasks. Event-triggered tasks, are tasks which trigger sporadically due to some significant event (or sequence of them) [42]. It is practically unfeasible to statically predict when these tasks are going to be executed. As a result, event-triggered tasks are dealt with dynamic scheduling [49, 60, 25, 26], and in most (but not all) cases, event-triggered task scheduling is associated with task pre-emption.

Alternatively, time-triggered tasks are those which activation happens at pre-defined lapses in time [43]. In this work, we address the case in which this happens cyclically.

2.5 Enforcing Resource Usage Quotas

Controlling how hardware resources are made available to cores, has been studied as a solution to enforce conflict avoidance or reduction. Some works rely on specific assumptions on application semantics and hardware mechanisms, or by exploiting RTOS-level mechanisms to enforce predetermined resource usage quotas.

A memory bandwidth management system to enforce memory usage quotas has been proposed in [66]. Authors distinguish memory bandwidth in two components: guaranteed and best-effort. The basis of this work relies on assigning priority to the task phases

reclaiming guaranteed bandwidth, making these tasks run in temporal isolation, while only best-effort activities are run together under contention.

To derive the response time in multicore architectures, in [60] the authors propose an extension of the method described by Tindell *et al.* [63] which relies on the usage of busy windows or intervals, with emphasis on the bus through which different components communicate and access memory. Differently to our case, these works derive analyses for fixed priority pre-emptive systems.

When hardware isolation is not an option, contention can be controlled at software level by exploiting specific RTOS support (such as PikeOS [7]) to enforce analysis-time utilization bounds. Along that direction, in [67] the authors propose a software thread-based mechanism to access the shared resources minimizing contention. However, addressing the resources quotas management by software unavoidably adds overhead, and furthermore it is not the case that we always can count on RTOS support.

In our approaches, on the other hand, we do not enforce any timing intervals in which tasks must access resources.

2.6 Using Performance Monitoring Counters to Model Contention

In [61] the authors clearly distinguish the three main items that we need to know to derive the effect of tasks accessing a shared resource, namely:

(i) The quantification of the amount of shared resource operations issued by a task and all tasks on a core;

(ii) The analysis of the total latency experienced by a set of such operations on the shared resource; and

(iii) The inclusion of this delay in the response time analysis of each task.

We are assuming that this information can be derived (directly or indirectly) from the Performance Monitoring Counters (PMCs) available on COTS platforms. Previous works [25, 28, 36] have used PMC-based profiling. Dasari *et al.* [25] used PMCs to capture accesses generated on each core in an event-triggered system. For that sake, no caches are shared (as otherwise the number of requests could vary depending on interference). Although they do not perform access pairing (as will be further detailed in Chapter 4), the solution they propose holds some similarities with the hereby presented iterative approach. Instead, in [26] the same author discards an iteration-based approach computation in a cyclic schedule involving computation of the maximum interference generated and suffered, evaluating it as unsafe. This conclusion is justified by the fact that they build on the concept of intervals to define the worst-case interference caused by a core, whilst the approach from this thesis is preserving the analysis assumptions by leveraging

on time-triggered activation and inter-core synchronization. An example which gives more insights and supports the correctness of our approach is given in Chapter 4.

Chapter 3

System Model and Assumptions

Deriving reliable and tight upper bounds to multicore contention is a well-known problem in the real-time embedded systems domain. The (maximum) contention possibly suffered by a task is clearly dependent on the system hardware and software specification. A generic formulation is likely to incur an unacceptable amount of pessimism. For this reason, the effects of contention have been studied under several system and hardware level assumptions.

In this chapter, we present and elaborate on the main assumptions we embraced in the formulation of our approach on both the system (software) model and the characteristics of the underlining hardware platform.

3.1 Hardware Model and Observability

While specific contention bounds can be defined for each shared hardware component and operation, the overall induced timing interference suffered by a task is the cumulative effect of contention over all shared hardware resources accessed by the task. Some COTS platforms exhibit multiple sources of contention: for example, contention in the Infineon AURIX Tricore TC27x and TC29x processor families may happen when accessing the flash memories, the static RAM or any (slave) interface in the interconnect. More generally, the interconnect is the main source of contention in the system. When the interconnect allows accepting and serving multiple requests in parallel (e.g., the SRI cross-bar in the AURIX processors) then contention is moved from the interconnect itself to the destinations of the requests sent over the interconnect.

In this work, we focus on systems where all external (off-chip) requests must pass through a shared bus, which in turn is only capable of serving one request at a time, according to a given arbitration protocol. We assume the bus follows a round robin arbitration protocol. Note, however, that focusing on those systems does not limit the applicability of our methods: the proposed approaches can be easily extended to model different system configurations. For the time being, we will consider a hardware platform where cores are

equipped with private L1 instructions and data caches, and the shared bus interconnect (where contention arises) is used by all cores to access either the main memory or a shared L2 unified cache. We further assume that the L2 is partitioned among cores, so that indirect inter-core interference is excluded a priori from the modeled system. A schematic view of our reference platform is given in Figure 3.1. The data cache implements a write-through no write allocate policy, while the L2 cache is write-back. These cache policies are quite common in COTS platforms in the embedded domains (e.g., the LEON/NGMP and the ARM Cortex A series processors).

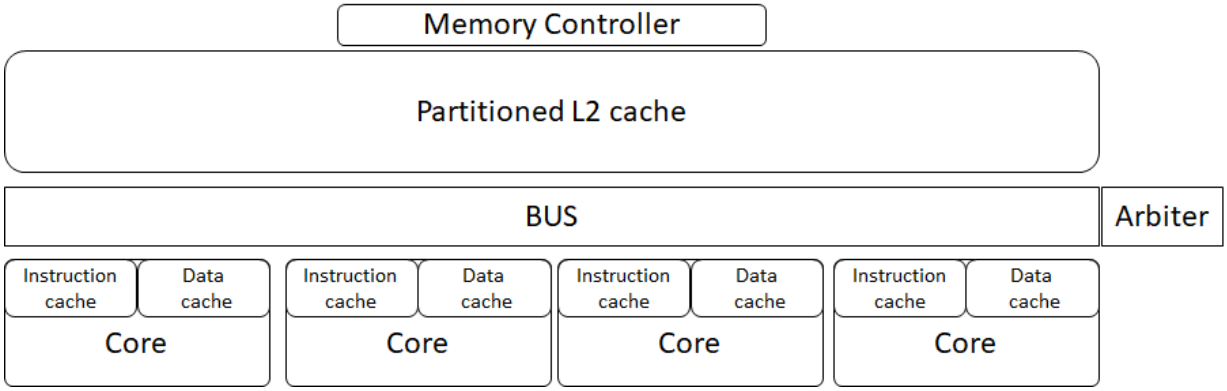


Figure 3.1: Reference processor architecture

Our methods are also exploiting the fact that normally the time required to serve a request may vary depending on the request type. In our reference platform, while all requests go through the bus, the maximum latency incurred by each request depends on the particular access type. The bus is the ultimate source of contention as when a request is granted to access the bus, then it will not release the bus until completely served. Different access types can go through the bus to reach the L2 and possibly the main memory: L2 store hits (*s2h*) and load (*l2h*) hits, L2 clean and dirty misses triggered by load (*l2mc*, *l2md*) and store (*s2mc*, *s2md*) operations, which are the ones tracked by the performance monitoring counters.

3.1.1 Observability assumption

In relation to number and types of requests triggered by a task, we further characterize the reference hardware platform by assuming that the hardware layer offers a sufficient degree of observability. Both of our approaches rely on the knowledge of the request pattern of a task: it is assumed that the number and types of accesses can be derived either by static analysis [65] or dynamically [25, 29].

Each access type then has to be attached to its worst-case latency, depending on the concrete target platform. For the sake of practical applicability, we prefer not to rely on a particular analytical approach (e.g., data flow analysis) to derive per task requests: we

assume this information is made available through a set of dedicated counters, aka Performance Monitoring Counters (PMC), typically implemented in the Performance Monitoring Units (PMU) of modern COTS platforms. With an adequate PMC support, tasks can be easily (dynamically) profiled with respect to access types and number.

Maximum latencies per access can only be derived dynamically [65], to confirm or disprove the information in the platform technical specifications.

3.2 System Model

In this work we consider a set of m independent periodic tasks with constrained deadlines, denoted $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_m\}$. A task τ_i is, therefore, defined as a tuple (c_i, r_i, P_i, D_i), where c_i, r_i represent respectively the worst-case execution and release time of τ_i, and P_i, D_i represent task's period and relative deadline. Our focus is on contention analysis and we are not interested in finding an optimal task schedule and/or assignment of tasks to cores [56]. Therefore, we assume tasks are statically assigned to cores and task migration is excluded. We consider a set of homogeneous cores Π, and for each core in the system, we use $\mathcal{T}_s$ to identify the subset of $\mathcal{T}$ mapped to a given core π_s. That is $\mathcal{T}_s = \{\tau_i \in \mathcal{T} \wedge \tau_i \mapsto \pi_s\}$. Further, tasks on a core are scheduled non-preemptively, following an order that implicitly model the precedence constraints between tasks. The assumed schedule is not work-conserving as a job will not be executed before the end of the reserved slot (timing budget) associated to the job of its predecessor task, even if the latter has already terminated. Finally, it is assumed that there exists no inter-core data dependencies among tasks or, if so, the order of their execution is not altered by this fact.

3.2.1 Types of accesses

Assuming a system consisting of an homogeneous multicore processor, access targets and types are the same across the system. The information on the target of an off-chip access could be relevant in the presence of multiple interconnects or interconnects that support some degree of parallelism. However, for the sake of simplifying the notation, operation targets can be encoded as part of the operation type (e.g. operation type is read or write to a given target). In the specific case of our reference platform, we do not even need to consider the target of a request since all requests go through the same shared bus. Therefore, the target information in the model formulation is being omitted. Different types of requests $\{t^1, \cdots, t^k\} \ni \Gamma$ are served through the interconnect, each request $t \in \Gamma$ exhibiting a different latency, for which an upper bound l^t is known (e.g., obtained through exhaustive measurements). We use the notation a_i^t to address the set of accesses of type t performed by task τ_i. Consistently, a_i defines the cumulative set of accesses, of any type, performed by the same task.

The specific type of accesses that can be considered and tracked are platform-dependent, as well as their latencies. In Chapter 5 we will provide an evaluation of our approaches

on top of a specific platform, for which the different access types and latencies will be obtained. Table 3.1 summarizes the notation used for the task model and the different access types and latencies, as will be used in the remainder of this thesis.

Tasks, cores and mapping	
$\Pi \stackrel{\text{def}}{=} \{\pi_0, \cdots, \pi_{n-1}\}$	Considered set of homogeneous cores
$\mathcal{T} \stackrel{\text{def}}{=} \{\tau_1, \tau_2, \ldots, \tau_m\}$	Considered task set
$P_i \geq D_i$	Period (P_i) and deadline (D_i) of task τ_i
c_i	Worst-case execution time in isolation of τ_i
r_i	Release instant of task τ_i
$\mathcal{T}_s \stackrel{\text{def}}{=} \{\tau_i : \tau_i \mapsto \pi_s\}$	Subset of tasks statically mapped to core π_s
$\Gamma \stackrel{\text{def}}{=} \{t^1, \cdots, t^k\}$	Access types admitted in the system
l^t	Upper bound to the latency incurred by a single access of type $t \in \Gamma$
$l^{\max}$	Worst-case latency incurred by a single access of any type in $t \in \Gamma$
a_i^t	Number of accesses of type t in τ_i
$a_i \stackrel{\text{def}}{=} \sum_{t \in \Gamma} a_i^t$	Total number of accesses issued by task τ_i

Table 3.1: Task and core mapping notation used throughout the thesis

3.2.2 Contention and pairing of accesses

Contention happens when requests are sent to the same shared resource. We model contention using the concept of *paired* accesses among tasks running on distinct cores. In our reference platform, the source of contention is on the bus and pending requests are served according to a predictable policy (e.g., round robin in this case). Due to the round robin arbitration policy adopted, an access can be paired with at most $|\Pi - 1|$ accesses (being Π the set of cores in the system), each from a different core, regardless of the type of such accesses (which in turn will determine the suffered delay). The total number of accesses a_i of a task τ_i is the summation over all access types of that task, i.e. L2 dirty and clean misses, store and load hits. To account for contention effects, the budget of a task is then inflated by adding the (worst-case) latency for each paired access, as determined by the access type of the contenders. We therefore need to model pairings while distinguishing among access types: for each task τ_i the number of issued accesses a_i^t for each access type t is taken into account. We always have to refer to worst-case access counts that hold valid for any path in the program, as argued in [25].

Figure 3.2 shows how a task can exhibit an increase in its execution time (marked in red) because of contention on a shared given resource. There are two scenarios in which

a task accessing a resource can be delayed. One, is that two or more tasks try to use the shared resource at the precise same instant; the other one, is that a task tries to do so while it is already being used by another task. Whilst in the first case the contention incurred will be l^t, in the second one it will only be a fraction of l^t. Despite this differentiation, since it is practically impossible to exactly predict the instant at which tasks will perform their accesses, we are forced to conservatively assume a latency of l^t every time an access is *paired*.

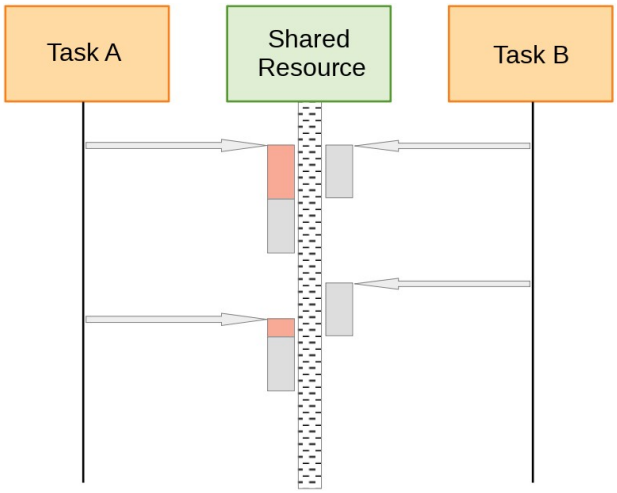


Figure 3.2: Contention: pairing accesses

In Chapter 4, we will further elaborate on how these assumptions are factored in the proposed approaches for the computation of the worst-case contention delay.

3.2.3 Cyclic executive assumption

Real-time systems are either event-driven or time-driven, in the sense that actions may happen either in response to an event or at a specific point in time. In our work we favor the second paradigm and assume tasks are non-preemptively executed according to a time-triggered [41], cyclic static schedule, also known as cyclic executive. Cyclic executive scheduling is widely used in critical embedded real-time systems due to its predictability and low implementation requirements. The basic scheme is to cycle through a statically defined, repeated sequence of activities, at a given frequency. Since the triggering times of each job are computed statically (offline) and tasks are non-preemptive, no RTOS support is actually required for scheduling.

All the tasks/activities are completely executed within one hyperperiod, which is further divided into smaller frames. The timing deadlines are enforced only at each frame boundary. That means, that each task is executed and must fit in a single frame. A frame does not necessarily contain one task, it can incorporate multiple of them. Tasks can also be sliced into smaller parts when possible, to seek for optimal frame allocation.

Within each frame, optionally, lower-priority sporadic activities can be executed during slack slots in the frame, not used by periodic tasks. Most commonly, non-periodic work must be scheduled preemptively to ensure that we are always able to start executing the next frame [1].

Logically, the sum of execution times (including the interference) of all the tasks allocated to a frame has to be less or equal than the frame time slot. In order to discard the possibility of frame overruns, we must ensure that non-preemptive tasks finish their execution within the frame. This can be achieved by precedence constraints (assigning priorities), enforcing task τ_{i-1} to finish before task τ_i by means of tasks' release times.

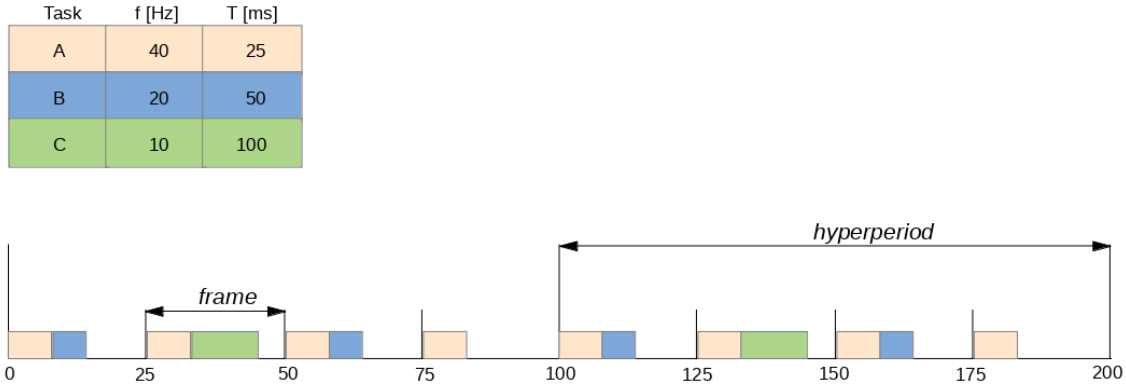


Figure 3.3: Generic example of cyclic executive scheduling

Figure 3.3 illustrates the main concepts of cyclic executive systems. In order to derive a proper schedule, an offline computation phase is required. In the case of this thesis, the offline phase is extended to account for and compute the WCD for each task, as a means to provide safe frame-level timing guarantees.

It is worth noting that the amount of contention, as well as the possibility of contending requests to occur at the same time, largely depend on the system architecture. In this work we consider a multicore instance of the ARINC 653 framework, where a static scheduler is implementing the repeated execution of a major frame (MAF), which is the equivalent to the hyperperiod, further subdivided in a sequence of minor frames (MIF). A statically mapped set of periodic tasks are non-preemptively executed within each MIF on top of a multicore platform.

Figure 3.4 illustrates how tasks are mapped to a given MIF, by allocating a slot corresponding to the pre-computed timing budget. The more the utilization at MIF level (by allocating more tasks) the better. Seeking large utilization goes hand in hand with the need of safe, reliable timing guarantees, ensuring that the MIF deadline will not be overrun under any execution condition or critical alignment of the possible events.

Reliable timing budgets needs to be computed also taking into account the effect of contention. In the example in Figure 3.4, for instance, a 50% of MIF's utilization would mean that the timing budget computed based on the execution times measured in isolation

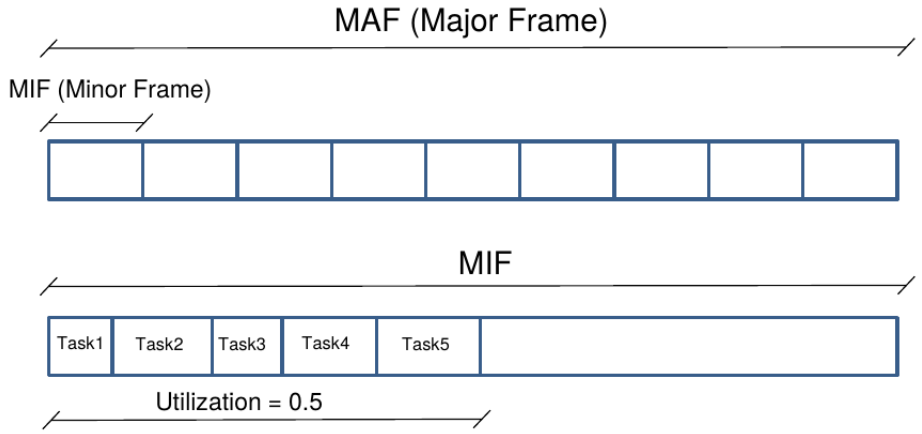


Figure 3.4: Task budgets within the MIF and MAF concepts.

of the set of mapped tasks only cover 50% of the MIF. We do not know, however, how much of the remaining 50% can be used to accommodate more tasks or even whether it will be sufficient to cover the effects of contention. Being able to precisely answer this question is the final objective of our work.

This work refers to IMA and ARINC-653 [13] concepts, but the same reasoning can be extended to cover statically scheduled systems in general. In these systems, tasks or functions are executed within pre-determined scheduling slots of a given duration. In the scope of this work, we consider MIFs as the smallest time interval at which timing guarantees are required to hold. The static schedule is therefore defined by a static allocation of tasks to MIFs that execute in sequence within the MAF.

In this multicore scenario, computational resources in each core are assigned according to the same static scheme (as determined by MIFs and MAF), which means that cores' execution is synchronized at MIF boundaries. Core synchronization at MIF boundaries is featured, for example, in the IMA multicore implementation supported by SYSGO PikeOS [9] RTOS, and in the Multicore Multiple Independent Levels of Security/Safety (MILS) paradigm supported by VxWorks RTOS [10, 51]. Despite sharing common scheduling slots, functions on different cores do not engage in any synchronization or communication protocol, unless they preserve time-composability [16].

Chapter 4

Design and Solution

Deriving safe and tight bounds to the WCD is crucial for the computation and apportionment of reliable timing budgets to the functions provided by the system. COTS multicores feature several shared hardware resources, which in turn represent just as many sources of contention. While all sources of contention are relevant for deriving trustworthy budgets, in the scope of this work we focus on the problem of computing reliable bounds to the WCD potentially suffered by a given program or task when accessing the off-chip memory.

Accesses to the off-chip memory are one of the most critical sources of interference [59, 26, 28]. Every access to the memory hierarchy (either direct or in response to a cache miss) must pass through the interconnect, which easily becomes a performance bottleneck and, equally important, a huge source of timing variability. The worst-case contention delay suffered by a task on bus access is a function of both (i) the number of accesses the task itself performs, and (ii) the number and type of accesses performed by its co-runner tasks [25, 37, 28]. Observation (i) is simply dictated by the fact that, based on conservative assumptions on access alignment and considering the specific arbitration policy on the bus, a task can suffer a contention delay at every single access in the worst case. However, as per observation (ii), the WCD is also determined by the set of potential co-runners of a task: first, the number of contending accesses cannot be larger than the number of accesses of other tasks running in parallel; second, the WCD depends on the type of accesses performed by the contender as not all accesses exhibit the same latency.

In this chapter we first introduce access pairing as the core technique we build on for the computation of conflicting accesses on a given hardware resources. Based on this, we present two approaches for computing the WCD at either task or core level.

4.1 WCD Bounding Based on Access Pairing

In order to conservatively but accurately capture the effect of different types of accesses, the concept of *access pairing* will be used. Pairing models the fact that requests from different cores can collide in the access to a shared hardware resource. Contention causes

an increase in the execution time of the interfered task: the impact on the WCET of the latter is bounded by pairing victim and culprit accesses. Accesses of interfering tasks will cause a contention delay (up to the interfering access latency) on the interfered task for each access they can be paired with. Note that pairing can only happen when the interfered task and the interfering task, in different cores, overlap in time (i.e. run in parallel). As an example, task τ_1 in Core 0, in the topmost part of Figure 4.1(a), performs two memory accesses, symbolized with •, that can be paired with two accesses in τ_3, executing in parallel in Core 1. Once paired, these two accesses in τ_3, will not be available for pairing with other accesses from Core 0. In consideration of the fact that contenders' accesses may exhibit different latencies depending on the access type, it is important to conservatively pair accesses starting from the most interfering ones (higher latency).

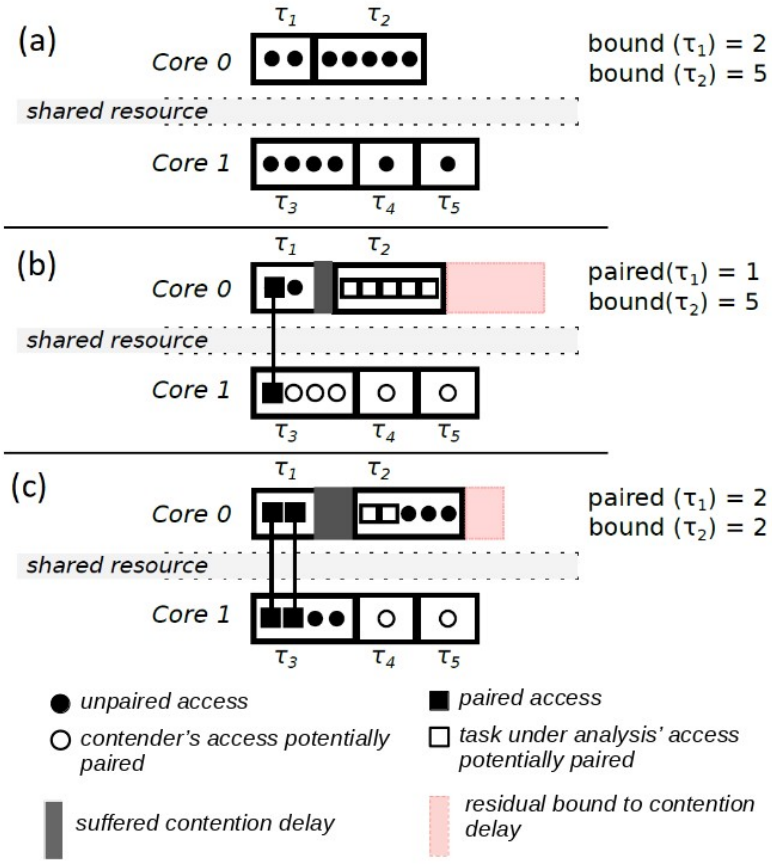


Figure 4.1: Access pairing: motivation

Access pairing, as a metric to compute contention, and tasks overlapping are in a *mutual relationship* as pairing is determined on the accesses from the set of overlapping tasks, and the latter is affected and potentially altered by the increase in execution time caused by the WCD. At a system level, considering all tasks in each core, such relationship can be sometimes counterintuitive as local worst-case pairing for a task may lead to a shorter

makespan. Figure 4.1 provides an illustrative example of such effect, avoiding any consideration on access types, for the sake of simplicity. Tasks τ_1, τ_2 execute on Core 0 and perform respectively 2 and 5 memory accesses, while tasks τ_3, τ_4, τ_5 execute on Core 1, with 4, 1 and 1 memory accesses, respectively (see Figure 4.1(a)). In this example, let's focus on the WCD of tasks running on Core 0 and the produced makespan. In principle, 6 out of 7 accesses in τ_1 and τ_2 can potentially be paired with accesses from tasks running on Core 1. However, this ultimately depends on how tasks overlap in time, which is influenced by how much contention tasks suffer. Let's assume a locally-good scenario, Figure 4.1(b), in which only 1 access in τ_1 is paired with an access in τ_3 (i.e. τ_1 execution time will suffer a delay while waiting for τ_3 to perform its 'paired' access). The resulting overlapping is still compatible with τ_2 having all its 5 accesses paired with accesses from τ_{3-5}. The local worst-case scenario, where τ_1 has all its accesses paired, instead, leads to a task overlapping that is compatible with τ_2 having at most 2 accesses paired with accesses from τ_{4-5}. Thus a local worst-case assumption leads to a shorter, potentially optimistic worst-case makespan.

As a motivation to this work, this simple example shows that trying to compute the WCD pursuing the local worst-case conflicts, on a purely per-task basis, can lead to unsafe system-level bounds. In this thesis we present two safe approaches for the computation of the WCD:

- As part of the iterative proposed method, we avoid the above limitation by not focusing exclusively on the local worst-case. When considering a pair of overlapping tasks, we still capture the local worst-case pairing of accesses, but we do not prevent already paired accesses to be paired with accesses from other tasks on the same core. Considering the example in Figure 4.1, we allow τ_3 accesses to be paired with τ_2 ones, despite 2 τ_3 accesses were already paired with τ_1. This is indeed an unrealistic condition as we know that one access cannot conflict with more than one access triggered on the one and same core; however, it conservatively guarantee safe WCD results. The resulting approach proceeds iteratively, propagating the effect of WCD computation on task overlappings, and determining the worst access pairing scenarios under given task alignments. In order to determine the initial task alignment as well as the ones after each iteration, different assumptions will be described later in this chapter.

- The second method builds on the observation that tasks overlapping conditions and access pairing lend themselves to be captured with an Integer Linear Programming (ILP) formulation, to compute the worst (longest) possible makespan. Hence, an ILP formulation is used to model and explore all possible (feasible) combinations of tasks overlapping and access pairing, thus providing a safe system-level bound (considering pairing across task boundaries) to the worst-case contention delay. System level bounds are particularly relevant in those real-time systems whose execution is clearly organized as the cyclic succession of time frames, such as cyclic-executive and Integrated Modular Avionics (IMA) systems, which are the main focus of this work.

4.2 Modeling Pairing of Contenders Accesses

The computed WCD tightness will rely on the concept of access pairing. This refers to how we align task overlapping and collide tasks' bus accesses in order to derive contention. To compute the WCD, we will make use some notation to identify the different pairing events. At any moment, τ_j is assumed to generate interference on τ_i: i.e., τ_j's request always precedes τ_i.

In this section we focus on the definition of accesses and pairing thereof. For what concerns task definition and notation, we assume the reader to be familiar with the notation introduced in Table 3.1, in Chapter 3. The WCD caused by τ_j on τ_i is computed based on the number (and type) of paired accesses since the latter are conservatively assumed to always generate interference on τ_i. To compute the WCD for τ_i we need to consider all possible access parings for τ_i's accesses, which depend on, and at the same time potentially affect, the set of (overlapping) contender tasks τ_j. The WCD analysis is performed per contender core (i.e., $\mathcal{T}_s : \pi_s \nleftrightarrow \tau_i$) and per MIF. The interference suffered due to each core are summed up to derive the global WCD for τ_i. When not otherwise specified, this analysis applies at the granularity of scheduling intervals or MIFs: the analysis has to be separately applied to all MIFs in the MAF. Since the model and its formulations apply to a given MIF, overloading the notation is avoided and an indicator of the considered MIF is not added either.

Table 4.1 illustrates the notation employed to describe the items that will be used to define the effect of contention.

Worst-case delay computation	
$\bowtie_i(\mathcal{T}_s)$	Set of tasks $\mathcal{T}_s$ mapped to core $\pi_s \nleftrightarrow \tau_i$, potentially overlapping with the task under analysis τ_i
$a^t_{j \triangleright i}$	Number of accesses of type t in $\tau_j \in \bowtie_i(\mathcal{T}_s)$ that are *paired* with accesses in τ_i
$a_{j \triangleright i}$	Total number of accesses (of any type) in $\tau_j \in \bowtie_i(\mathcal{T}_s)$ that are *paired* with accesses from τ_i.
$\Delta_{j \triangleright i}$	Interference suffered from τ_i because of contention triggered by *paired* accesses of **any** type in τ_j
Δ_i	Overall interference suffered from τ_i
$c_i + \Delta_i = e_i$	Execution-time budget reserved for multicore execution of τ_i after accounting for WCD effects

Table 4.1: WCD computation notation used throughout the thesis

The tightness of both approaches directly depends on their capability to exclude infeasible pairings. For that reason, $\bowtie_i(\mathcal{T}_s)$ is defined, as not every single task of the system can be a contender to a certain task under analysis (due to running at different instants under any possible contention scenario).

Furthermore, as it can be noted, $\bowtie_i(\mathcal{T}_s)$ is a reflexive relation: if task τ_i runs in parallel with task τ_j, naturally τ_j does so with τ_i: $\tau_j \in \bowtie_i(\mathcal{T}_s) \iff \tau_i \in \bowtie_j(\mathcal{T}_{s'})$. Further, since tasks running on the same core do not interfere each other, it holds that $\tau_i \in \mathcal{T}_s \Rightarrow \bowtie_i(\mathcal{T}_s) = \emptyset$.

When considering access pairings, slightly different assumptions may result in different solutions, even within the same approach, as we shall see with some examples. While some constraints are particular to a specific approach, some constraints are shared between the iterative and ILP approaches. Next we identify those constraints on access pairings that apply to both approaches.

Bounds on task-to-task access pairing

The total number of accesses in τ_j *paired* (hence conflicting) with accesses in τ_i is bounded by the access counts in both tasks:

$$a_{j \triangleright i} \leq \min(a_i, a_j) \tag{4.1}$$

where $a_{j \triangleright i}$ is a generalization of $a^t_{j \triangleright i}$ and indicates the total accesses (of any type) in $\tau_j \in \bowtie_i(\mathcal{T}_s)$ that are *paired* with accesses from τ_i. When considering different access types, the above constraint can be redefined on a per-type basis (to allow a consistent pairing of latencies) as follows:

$$\sum_{t \in \Gamma} a^t_{j \triangleright i} \leq \min(a_i, a_j) \tag{4.2}$$

More precisely, the above equation can be further constrained by considering that the number of paired accesses of a given type in the interfering task (τ_j) is limited by the number of accesses of that type in the interfering task and the number of accesses in the interfered task:

$$a^t_{j \triangleright i} \leq \min(a_i, a^t_j) \tag{4.3}$$

Bearing in mind the constraint defined by Equation 4.3, let's present the two approaches in more detail.

4.3 Iterative Approach

In this section, the iterative approach is presented. First, we will go through its foundations and principles. Then, we will describe in detail the steps to be followed for its application. We will enumerate the required assumptions and provide a simple and short example of

the application process. Finally, we will prove the safeness of the approach.

The iterative approach focuses on the computation of reliable WCD bounds at task-level. That means, it calculates the highest contention that each single task can suffer under certain assumptions. Aiming at task-level guarantees instead of system-level ones, comes at the price of larger makespans. In this method, an access of a contender is not immediately discarded when being paired against another task. Instead, we consider it along all the tasks it is overlapping (i.e., co-running) with. We do that in the impossibility of knowing which specific task/s it will be contending. Furthermore, when accesses of different tasks can happen at the same time, we must consider each one being the last one granted access to the shared resource. The reason for that, similarly, is that it is not possible to guarantee that accesses from different cores will be always served according to a given order at run time and conservative assumptions must be made to cover all possible scenarios.

These conservative assumptions, although safe, imply some extra pessimism at makespan level. More concretely, we identify two main sources of pessimism/overestimation in the iterative approach. First, the approach builds on pessimistic assumptions on **alignment of accesses** as it conservatively assumes that if an access can cause contention, then it will. Clearly, this is not necessarily true as it depends on run-time jittery execution. Secondly, and in part as a consequence of the above, the approach is **over-accounting** for accesses while pairing: at run-time, if a contender access is already paired (i.e., conflicting) with a task in the core under analysis, then it cannot be paired again with another access from another task on the same core (due to the round-robin arbitration). Instead, when a contender is overlapping with more than one task in the core under analysis, the iterative approach cannot assume a priori which is the *local* access pairing that will lead to the worst-case paring for all involved tasks. Therefore, as the analysis is applied to one task at a time, it conservatively assumes that contender accesses are conflicting with all overlapping tasks.

The final objective behind the iterative approach is to calculate a safe triggering time for each task, being certain that the previous job executing on that core will have already terminated, even if the maximum possible contention was to occur. The fact that we are enforcing the analysis assumptions (on task overlapping) by setting appropriately tasks' triggering times, will rule out many otherwise feasible overlapping scenarios, which in turn guarantees a very moderate computational complexity. Besides, in order to better understand the basis of the approach, the concept of *time composability* has to be clarified.

4.3.1 Time-composable worst-case delay bounds

Time composability refers to the premise that individual units such as tasks can be incrementally composed with others preserving the timing behaviour experimented when executed in isolation. When applied to a multicore processor, this can be translated into the property that the timing behavior of a task (or at least an upper bound thereof) is not affected by the activity of its co-runners [32]. However, due to the inter-core dependencies,

we must account for highly variable-timing behaviour, hence we can not straightforwardly assume that safe timing bounds in isolation will also hold in a multicore architecture. Therefore, it must be guaranteed that the assumptions on timing behaviour are still valid under the composition with a new set of tasks.

A strict way to meet that requirement is **full composability**, which implies that a system requirement (e.g. timing bounds) will hold no matter which components it is assembled with. From the perspective of multicore timing analysis, it means that all the tasks deadlines will be met, independently from the actual contenders tasks, and without any further restriction or assumption. To obtain fully time-composable contention bounds, one needs to assume the maximum possible contention at each instant: every single time that tasks attempt to perform a bus access, the maximum number of contenders will be encountered (with the highest latencies). Additionally, the task under analysis will be the last one granted access to the bus, and each contender will make use of the shared resource for the maximum possible amount of time. The concept of full time composability is related to WCD computation in [32] and further studied in [29], under the name of *fully Time-Composable* (fTC) bounds. Despite being safe by construction, fTC suffers from a huge amount of pessimism.

Time composability is generally considered as an all-or-nothing metric [32]: when joined with a set of components, a system will either hold its timing guarantees or not. No middle term exists. For that reason, the WCD bounds that we derive must take into consideration the composition of all the possible components (i.e. co-running tasks).
Instead, in order to avoid (unusable) overly-pessimistic WCD bounds, a **partially** time-composable guarantee can be provided on the system timing behaviour by, for example, reducing the spectrum of possible contenders, and guaranteeing that the provided bounds are valid for these determined set of components. That means that we are able to guarantee deadlines for a certain set of co-running tasks, even under the worst possible alignment, but after having discarded many unrealistic scenarios (as we know our task can only overlap with some predefined contenders). However, in order to do so, we require information on the actual contenders at run time. The same authors in [29] provide an alternative formulation to derive a more realistic bound without losing sight of the required safety guarantees. They call this composition *partial Time Composability* (pTC), which takes into account only the number (and type) of accesses to the bus possibly performed by the actual set of potentially overlapping tasks.

The pTC paradigm is at the basis of our approaches, and the iterative approach presented in this thesis computes pTC bounds, as will be further described in the remaining of this chapter. An important factor to consider (even in the scope of pTC bounds) is that the hardware resource (bus in this case) supports different access types. If we disregard this, we are forced to build up on the worst-case assumption that each access will always incur the maximum (worst-case) latency, which is not generally the case. We explicitly recognize that requests of a task may generally exhibit variable latencies, depending on the request type: we take into account the different request types to tightly compute the worst-case contention effects.

fTC contention model

To derive an fTC WCD bound, we have to assume that every single access to a shared resource that a particular task performs will encounter the maximum number of contending tasks. That is, $|\Pi|-1$, since only one task per core can be contending at a precise instant, and the contention delay will be maximum (mcd) as well. This guarantees that no runtime scenario will be able to cause a worse delay. The fTC model can serve as an upper bound to the contention and slowdown that a task can suffer.

Therefore, to compute the WCET of task τ_i (e_i) under the fTC assumption, we will have to assume that every single access to the bus (a_i) incurs the maximum penalty possible in the system ($l^{\max}$). This is formalized in Equation 4.4 below:

$$e_i = c_i + \Delta_i = c_i + a_i \times (|\Pi|-1) \times l^{\max} \tag{4.4}$$

Where e_i represents τ_i's WCET after considering the WCD effect, as defined by fTC.

pTC contention model

However, the fTC conditions are obviously too pessimistic and unlikely to happen in real scenarios. For that reason, in pTC, we bound the number of conflicting accesses, based on available information on tasks' access profiles. We consider actual accesses to better assess which contenders and accesses might cause conflicts and incur contention delays. Overall, by reducing the WCD, we obtain a consistent reduction in the WCET as only the possible contending accesses and types are considered in a given scenario.

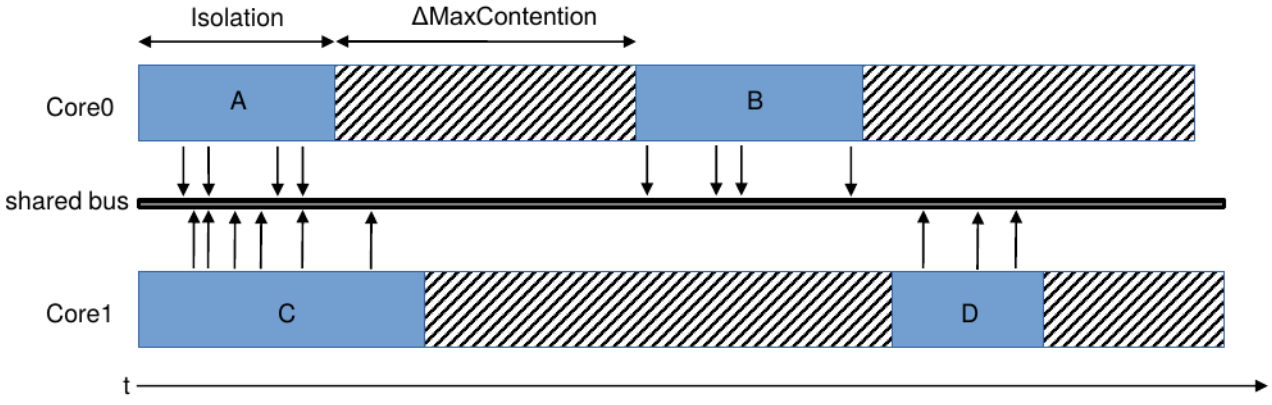


Figure 4.2: Contention effect

For instance, in the scenario reported in Figure 4.2, we no longer need to assume that every access is conflicting with accesses from every contender (each causing an $l^{\max}$ delay). In fact, most of the accesses are distributed along tasks' execution in instants when no contenders are found. However, for the sake of safeness, when contention happens, we

will still have to assume that our task under analysis is the latest one to be served. That will ultimately result in being able to safely reduce the considered maximum contention, leading to tighter task budgets.

The iterative approach aims at finding pTC WCD bounds. Given an initial set up (task WCET and overlapping), the technique consists in repeatedly applying the analysis until a fixed point is reached, that is the WCD bounds computed at iteration I_k did not change compared to iteration I_{k-1}.

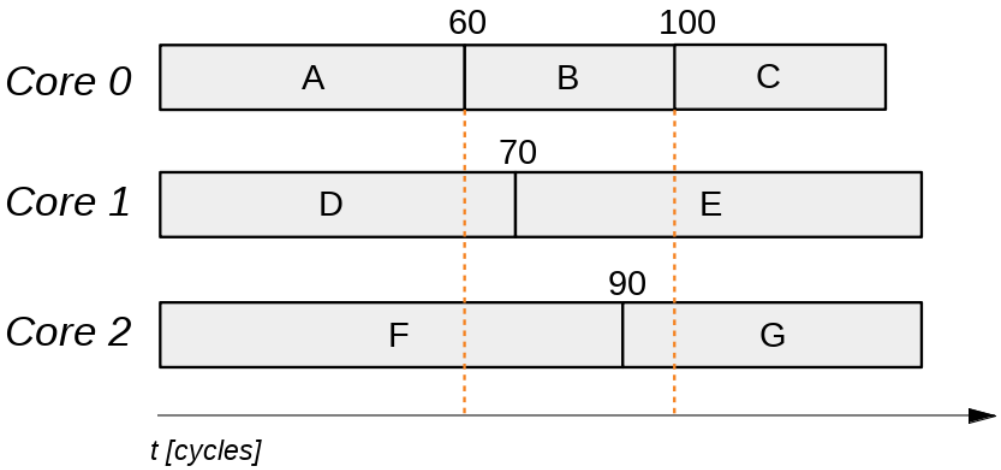


Figure 4.3: Overlapping

The following steps are performed at each iteration:

1. Join up the tasks' budgets (without any slack time in between) mapped to each core and "pair" their accesses in the most pessimistic manner. To do so, we first derive, for each task, the set of potential contenders across all other cores. For instance, as Figure 4.3 shows, task B is running together with D, E, F and G. Secondly, we start pairing B's accesses against the ones in each contender that incur the highest latency. If $a_B > a^t_{j \triangleright B}$ being t the type of access incurring l^{max}, we keep pairing remaining B's accesses with the accesses in the contender that cause the second-highest contention delay, and so on so forth.

2. Considering the analyzed access pairing, calculate the new budget inflation that these access pairing would imply, provided the latency of each access type can be obtained.

3. Update the tasks budget to form a (possible) new alignment scenario.

To perform these steps, a key point to analyze is how we initially allocate tasks to start performing the pairing analysis in the first iteration. Two main cases were studied: start joining up tasks considering their maximum execution time in isolation, and their fully time composable triggering times.

The next simple example illustrates that in fact, different starting points lead to different computed makespans. The example characteristics are as follows:

1. Only 2 cores are considered.
2. Only 1 type of request is assumed, incurring a maximum latency of 10 cycles.
3. Tasks A and B are allocated to Core0 (π_0), and C and D to Core1 (π_1).

For each task, we assume that the following information is available: core mapping, number of accesses and WCET in isolation (from which we also derive the fTC bound). Table 4.2 summarizes the data assumed in our example.

Task	**Accesses**	**Time Isolation [cycles]**	**WCET (fTC) [cycles]**
A $\rightarrow \pi_0$	4	60	100
B $\rightarrow \pi_0$	3	100	130
C $\rightarrow \pi_1$	2	70	90
D $\rightarrow \pi_1$	3	80	110

Table 4.2: Example input data

The WCET fTC times are calculated by applying Equation 4.4. Bearing in mind that in this example we count on only 2 cores, and that there is only one latency kind, the WCET of task A is calculated as: $60 + 4 \times 1 \times 10 = 100$. Analogously, the rest of WCETs are obtained.

4.3.2 Execution time with fTC as starting point

fTC is by definition the maximum contention a certain task under analysis can suffer, regardless of which contenders it encounters. It therefore represents a safe starting point for the analysis. Nevertheless, we want to understand whether different starting points will converge and lead to a common solution and, if they do not, whether the smaller bound is still safe.

Figure 4.4 displays the initial starting point if we consider fTC bounds. In that case, we start assuming the maximum contention and will keep reducing it by checking the actual tasks' alignment. Task τ_i is only triggered after the budget of task τ_{i-1} expires, which boundary is marked with the stripped-lines area. In that scenario, we consider which tasks are overlappings with the task under consideration. Among the tasks that overlap, we have to pair their accesses in the most pessimistic way (as per Equation 4.3). That means, that we have to start assuming that each access of our task under analysis clashes with those in overlapping tasks, if any, that incur the longest latency. For the sake of simplicity, in

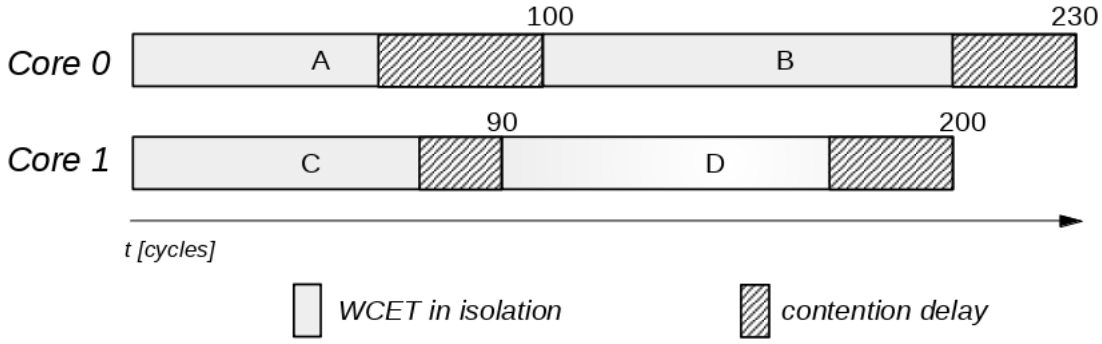


Figure 4.4: pTC from fTC - Iteration 1

this example only one type of access is considered.

Referencing the presented equations in Section 4.2, a certain task under analysis will only pair the minimum between its accesses and the contenders'. Moreover, a task under analysis can not collide more than its own accesses against tasks allocated to a same core. For instance, in Figure 4.4, D can at most pair 3 accesses globally to π_0. In contrast, from π_0's perspective, both A and B must account for the 3 accesses of D (over-accounting effect).

Bearing that in mind, we start pairing tasks accesses. We can appreciate that the WCET of task A (100 cycles) is paired with C and D from π_1. A is accessing the bus 4 times, hence can pair at most 4 accesses with a singular task, and at most 4 accesses within tasks of a contending core. This is a safe assumption since the contention effect on the task is the same no matter whether the clashing access occurs with task τ_j or with task τ_{j+1}. As a result, we sum the accesses of task C and D and take the minimum between that sum and A's accesses, that is, 4 contending accesses for A. Analogously, the rest of contending accesses per task is calculated.

A $\rightarrow$ Its 4 accesses are paired with either C or D ones (as they sum 5).

B $\rightarrow$ Task B performs 3 accesses. Since it only co-runs with D, which also accesses the bus 3 times, all B's accesses are paired.

C $\rightarrow$ This task only runs in parallel with task A from π_0, and since this contender performs more accesses, we can pair all C's accesses with that core.

D $\rightarrow$ We see that task D which counts on 3 accesses, and since it runs in parallel with A and B, it can clash its 3 accesses with either of them too.

The next step requires to realign the timing budget to the new contention bounds and update tasks' triggering time accordingly. The new budgets can be computed as the time

in isolation of each task plus the calculated contention (pTC from now on). The triggering times of successor tasks, as always, is adjusted to previous task's budget. As it can be noted, the triggering time of a task is the summation of the budgets of the predecessor tasks. In this simple example, the alignment is exactly the same, and there is no need to reiterate the process (a fixed point has been reached). The computed budgets and triggering times define a safe schedule for the MIF. A further iteration would yield the exact same results.

Task	**pTC**$_1$	**Triggering Time**$_1$
A	100	0
B	130	100
C	90	0
D	110	90

Table 4.3: pTC after iteration 1 - fTC as starting point

Note that if the task pairs were the same but the release times had slightly been modified, we would repeat a further iteration to recompute the budgets, as it could be the case that more or less accesses would collide as a result of this incremented or decremented budget.

Although these computed tasks' budgets and release times represent a safe solution, the fact of having started assuming the maximum contention delay per each task introduces some extra pessimism, which can be avoided by taking the opposite assumption (i.e., times in isolation with no contention) as a starting point, as will be made evident in the following Section.

4.3.3 Execution time in isolation as starting point

On the other hand, the same exact process can be applied but varying the first initial budgets and consequently pairings.

Next, we present a counter-example showing how the makespans can be shortened if starting from time in isolation. The execution time of tasks in isolation and the number of accesses they perform are the same from the previous example.

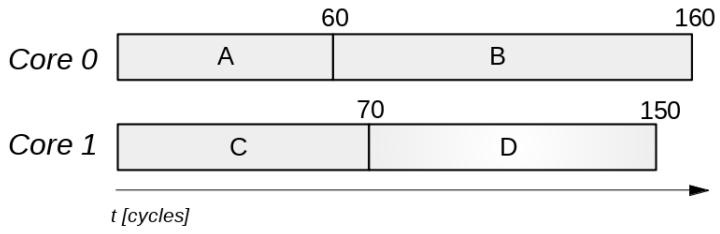


Figure 4.5: pTC from isolation time - Iteration 1

Similarly, we perform the core step consisting in pairing accesses:

A → Only 2 of its 4 accesses are paired with the ones from C.
B → Since C and D co-run with B and they perform more accesses than B, B pairs all its accesses.
C → Analogously, C can pair its 2 accesses with A and/or B.
D → D and B run together and they both perform 3 accesses which are paired.

Table 4.4: Access pairings iteration 1 - Isolation as starting point

In this case, the tasks' budgets keep increasing most of the times, instead of reducing as it used to happen when starting from the fTC bound. Table 4.5 summarizes the new budgets and release times after one iteration:

Task	**pTC**$_1$	**Triggering Time**$_1$
A	80	0
B	130	80
C	90	0
D	110	90

Table 4.5: pTC after iteration 1 - Isolation as starting point

Since there has been a variation in tasks' budgets, we must proceed with another iteration, placing the tasks with their new computed budgets:

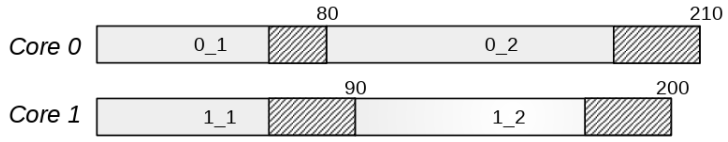


Figure 4.6: pTC from isolation time - Iteration 2

Despite budgets have been increased with respect to the previous iteration, the tasks' alignment has not been altered, and hence the pairing will be the same, leading to the same budgets calculated in iteration 1 (see Table 4.6).

Once again, since $I_2 == I_1$, a fixed point has been reached, meaning that no further iterations would either add or remove contention delays.

4.3.4 Comparison, benefits and safeness of the approach

Considering the overall makespans computed under the two methods, we observe that the makespan in π_0 is 20 cycles shorter when starting from isolation execution times. It is

Task	**pTC$_2$**	**Triggering Time$_2$**
A	80	0
B	130	80
C	90	0
D	110	90

Table 4.6: pTC after iteration 2 - Isolation as starting point

important to understand whether the lower makespan value can still be considered a safe bound, i.e. if a run-time alignment could possibly lead to a worse execution. This also corresponds in understanding whether, by starting from the fTC bound, we are introducing impossible scenarios.

The reason why starting from fTC is more pessimistic is that it includes unrealistic scenarios. In fact, in order for a certain task to be considered to be overlapping with another from another core, we have to take into account the effects of contention **without** considering the contention among these same tasks. Those cases where overlapping occurs only because we have already accounted for some degree of contention between the considered tasks, are definitely not feasible and can be safely discarded. Although the pessimism in this case was of only 20 cycles, it becomes clear that with far more accesses, different latencies, more tasks, and more cores, the unnecessary pessimism introduced would be much larger. Applying this reasoning to the above example, if we start aligning the tasks considering the fTC, task A would be paired with task D, accounting for additional contention. Nevertheless, this alignment is already considering the access pairings with D, which will never happen on a real case if A and C are triggered at the exact same cycle. Since C only performs two accesses to the bus, and A is only running in parallel with C, 20 cycles is the maximum contention delay that A can experience. As a conclusion, starting from the isolation execution times provides a tighter – and still safe – result.

In any case, we still have to justify why the resulting computed WCD with this method is indeed safe. We could intuitively suspect that it could be the case that by not pairing the maximum number of accesses at each moment as we currently do, could lead to a larger system-level makespan. Even though, with this approach this can not happen.

To clarify that, let's see the previous example with a variation in tasks' accesses such as Table 4.7 denotes.

We could at first glance think, that if A is the latter one to get granted accesses to the bus in case of a collision with C, that would increase its budget by 20 cycles whereas C would finish its execution within 70 cycles, provoking A to run in parallel with D as well, and hence pairing its remaining 8 accesses with D (and as a result incurring a larger makespan). However, that case is excluded with this method, since we are *forcing* the release times of tasks statically, so task D will be triggered at cycle 90 (91 more precisely, after C's calculated budget), making it impossible for task A to ever face task D, because

Task	Accesses	Time Isolation [cycles]	WCET (fTC) [cycles]
A $\rightarrow \pi_0$	10	60	160
B $\rightarrow \pi_0$	4	130	170
C $\rightarrow \pi_1$	2	70	90
D $\rightarrow \pi_1$	8	120	200

Table 4.7: Sample input data: safety justification

task A completes at cycle 80. Here lies the main point of this approach: the real tasks alignment on a platform would be the one we are statically enforcing based on the analysis results. Under the non-work-conserving assumption, when a job terminates before its allocated budget, the next job of the subsequent task in the same core, will still not be triggering until the statically scheduled time, guaranteeing that no pairing scenario can happen that was not considered in the analysis.

4.4 Modeling via ILP

In this approach, alternatively, we are interested in computing tight **system-level** WCD bounds (makespan of the MIF functions) as a means to increase the guaranteed performance on each MIF, and hence to be able to accommodate more functions while preserving the timing constraints. In contrast to its task-level counterpart, the system-level WCD computation implies considering the WCD over a sequence of tasks (under the non-preemptive assumption, common in cyclic executive systems). This allows discarding the possibility of considering an interfering access to cause conflicts on more than one task in the same core.

Since in this case we are performing a system-level analysis, the problem gets a bit more complicated, since as we have seen in a previous example, a locally bad alignment does not necessarily bring us to the worst case execution time. To that end, the only solution is to analyze all the possible alignment scenarios and compute the most unfavorable one to ensure safety in a brute-force manner. Doing so in bare code would be computationally expensive, and for that reason, an Integer Linear Programming was chosen. The reason of that choice, is that ultimately, what we need to obtain is the number of cycles which we need to book to ensure that tasks can execute safely. This, can be modelled as an integer value, and so the number and combination of accesses paired to reach that number of execution cycles.

ILP consists of 4 phases:

1. Define variables $\rightarrow$ we define the variables for which we want to find an optimal value (e.g. number of dirty L2 misses from τ_j delaying τ_i).

2. Define constraints → limiting the value of the variables by means of the presented equations.

3. Specify objective → we must identify which result we want to optimize (e.g. maximize the makespan).

4. Solve problem → run one of the many existing solvers to solve our problem.

In order to model all that, we have set up the problem in Python. An interface library named PuLP serves as an intermediate layer for one of the solvers to translate the programming language as Figure 4.7 depicts. Several solvers were tested, being IBM's CPLEX the one which calculated a solution the fastest.

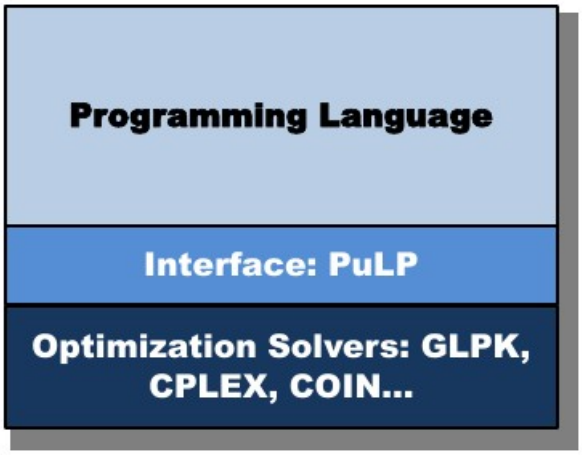


Figure 4.7: ILP layers

To properly capture all the system-level requirements, we have to add further constraints to our model which will be detailed below.

4.4.1 Bounds on task-to-task access pairing

In contradistinction to the iterative approach, in which we are setting the triggering times and hence must add the aforementioned sources of pessimism at task-level, in this approach we do can consider that in fact only one of the tasks will access the shared resource the latest, in the event that more than one are trying to access it at the same instant. For instance, if tasks τ_i and τ_j (assumed to be allocated to different cores and running in parallel) both perform 10 accesses and they happen to perform them all at the same instants, in the iterative approach we would have to consider that **both** can be contended by 10 $\times l^t$, whereas in ILP we can model that this contention is the overall contention in both (as either one of them will be the first one accessing it, i.e. not incurring any delay).

More precisely, task pair-wise interference (i.e. the sum of the paired accesses on one direction or the other) cannot be greater than the access counts in both tasks:

$$a_{i \triangleright j} + a_{j \triangleright i} \leq \min(a_i, a_j) \tag{4.5}$$

4.4.2 Bounds on core-to-core access pairing

While the above and the previously defined task-level constraints are fundamental blocks in the model, they still fail to accurately model the way accesses are paired from a system perspective. The system level perspective considers pairings over the set (sequences) of tasks mapped to a core. Core-level constraints allow modeling the fact that one specific access of a given task cannot be delayed (i.e., be paired) with more than one conflicting access per each contender core. For example, a given task access cannot suffer contention from two accesses performed by two distinct tasks, mapped to the same contender core. These constraints are considerably improving the tightness of the computed WCD, and are one of the aspects that differentiate this approach from related works [56, 47, 28].

The constraint is formulated first from the standpoint of the interfering task, observing that any of its accesses cannot be paired multiple times with tasks on the same interfered core (it can be paired with at most one access per core). Accordingly, a constraint must be added for the sum of (interfering) accesses of an interfering task τ_j, that are *paired* with accesses of tasks on a given core $\pi_s \leftthreetimes \tau_j$, not to exceed the number of accesses in τ_j itself:

$$\sum_{i:\tau_j \in \bowtie_i(\mathcal{T}_s)} a_{j \triangleright i} \leq a_j \tag{4.6}$$

Dually, we also have to enforce that the number of accesses paired with accesses triggered by the set of overlapping tasks from the same interfering processor can never exceed the number of accesses in the interfered task τ_i:

$$\sum_{\tau_j \in \bowtie_i(\mathcal{T}_s)} a_{j \triangleright i} \leq a_i \tag{4.7}$$

In a scenario admitting different types of requests (and latencies) as it is the case, we can redefine the constraints by modeling also access types as a further dimension to the problem, as done for the task-to-task constraints. The number of *paired* accesses of a given type, triggered by a given contender (τ_j) on the overlapping tasks on a given core cannot exceed the number of accesses of that type in the contender itself.

$$\sum_{i:\tau_j \in \bowtie_i(\mathcal{T}_s)} a^t_{j \triangleright i} \leq a^t_j \tag{4.8}$$

In its dual formulation, the number of *paired* accesses of a given type and triggered by the set of overlapping contender tasks on a given processor, is bounded by the overall number of accesses of any type in the interfered task (τ_i):

$$\sum_{\tau_j \in \bowtie_i(\mathcal{T}_s)} a^t_{j \triangleright i} \leq a_i \tag{4.9}$$

Generalizing the constraint in Eq. 4.9, we can also assert that the number of *paired* accesses **of any type** and triggered by tasks on a given processor is still bounded by the number of accesses of the interfered task itself:

$$\sum_{t \in \Gamma} \sum_{\tau_j \in \bowtie_i(\mathcal{T}_s)} a^t_{j \triangleright i} \leq a_i \tag{4.10}$$

It is worth noting that a necessary conservative assumption in the analysis of the WCD *at task level* would be to assume that the task under analysis always suffers the worst-case possible contention. As a consequence, pairing in the presence of typed accesses should conservatively pair those accesses that incur higher-latency first (see Eq. 4.3, 4.10). However, we do not need to model this constraint as the worst-case pairing (of a request of type t) to each task request is already induced by maximizing the overall *makespan*, as part of the objective function.

4.4.3 Objective function

For a given task τ_i, Δ_i is defined as the WCD suffered by that tasks when executed within the considered MIF. The inflated execution-time budget reserved for τ_i in multicore execution is denoted $e_i = c_i + \Delta_i$. Consequently, the objective function consists in maximizing the makespan of the set of tasks mapped to a core and executing within a given scheduling interval (MIF). This is equivalent, in terms of maximization, to the overall cumulative effect of contention suffered by those tasks. For example, considering core π_s (and a MIF):

$$\max makespan_s \equiv \max \sum_{\tau_i \mapsto \pi_s} e_i \equiv \max \sum_{\tau_i \mapsto \pi_s} (c_i + \Delta_i) \tag{4.11}$$

An alternative objective function would be that of maximizing the cumulative makespan across all cores within a MIF, such as $\max \sum_{\pi_s \in \Pi} \sum_{\tau_i \mapsto \pi_s} e_i$. However, this formulation would only provide realistic makespan figures, which cannot be considered as valid upper bounds for the individual cores.

The term Δ_i, used in the objective function to indicate the WCD suffered by τ_i, is the cumulative result of the contention caused by (paired) accesses from overlapping tasks mapped to contender (interfering) cores. For each task τ_i, $\bowtie_i(\mathcal{T}_s)$ identifies the set of tasks mapped to (interfering) core $\pi_s \nleftrightarrow \tau_i$ that can overlap in time with τ_i. At a task-to-task level, the variable of interest is $\Delta_{j \triangleright i}$ to represent the interference suffered by τ_i because of accesses triggered by $\tau_j \in \bowtie_i(\mathcal{T}_s)$.

The interference depends on the number *and* types of accesses in τ_j that are assumed to collide with accesses in τ_i. In order to model the different latencies entailed by multiple access types, $\Delta_{j \triangleright i}$ is defined as follows:

$$\Delta_{j \triangleright i} \stackrel{\text{def}}{=} \sum_{t \in \Gamma} a^t_{j \triangleright i} \times l^t \tag{4.12}$$

where $a^t_{j \triangleright i}$ represents the number of accesses of type t in τ_j that are *paired* with accesses in τ_i, and l^t represent the maximum latency for that type of access.

In fact, either a task is causing interference or is suffering from it. This constraint, however, will be automatically invalidated by maximizing the contention on one of the two tasks (i.e., setting one term on the left side of Eq. 4.5 to 0).

Under these conditions, pairing should conservatively assign higher-latency accesses first. This is not explicitly modeled as the worst-case pairing (of a request of type t) to each task request is already induced by maximizing the overall *makespan*, as part of the objective function.

Accumulating the interference generated by all overlapping tasks in all contender (interfering) cores:

$$\Delta_i = \sum_{\mathcal{T}_s : \pi_s \not\perp \tau_i} \; \sum_{\tau_j \in \bowtie_i(\mathcal{T}_s)} \Delta_{j \triangleright i}$$

4.4.4 Model constraints: modeling task overlapping

We need to include in the ILP model a convenient definition of the overlapping condition between tasks. For each scheduling interval, the static schedule defines the set of tasks running on each core, which in turn identifies the set of possible task overlappings. At any instant, each task will only execute in parallel with at most one task per contending core. Further, since tasks follow statically-defined precedence constraints, only a subset of overlappings are possible in practice.

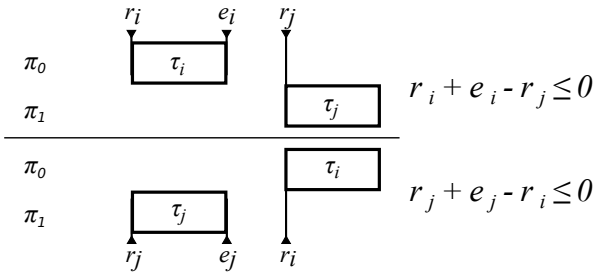


Figure 4.8: Non-overlapping tasks

The ILP formulation aims at identifying the worst-case overlapping among the feasible ones. In order to do so, we need to define an overlapping condition. We can formalize the

negative sufficient condition for task τ_j overlapping with τ_i as follows:

$$\tau_j \notin \bowtie_i(\mathcal{T}_s) \iff (r_i + e_i \leq r_j) \vee (r_j + e_j \leq r_i) \tag{4.13}$$

That is, τ_i ends in the worst case no later than τ_j release instant or vice versa, as illustrated in Figure 4.8. We can also observe that Eq. 4.13 is a reflexive relation, so that $\tau_j \in \bowtie_i(\mathcal{T}_s) \iff \tau_i \in \bowtie_j(\mathcal{T}_{s'})$. Moreover a task cannot overlap with tasks mapped to the same core: $\tau_i \in \mathcal{T}_s \Rightarrow \bowtie_i(\mathcal{T}_s) = \emptyset$.

The overlapping condition, however, has to be *linearized* in order to be included in the ILP model. Modeling conditionals is relatively simple when modeling binary variables as conditions can be easily obtained by using boolean algebra, which unfortunately did not apply to this case.

For each pair of potentially overlapping tasks τ_i, τ_j we can model the overlapping condition by encoding a constraint on $\Delta_{j \triangleright i}$ (equivalent to using $\Delta_{i \triangleright j}$), which is set to be null when $\tau_j \notin \bowtie_i(\mathcal{T}_s)$ (and $\tau_i \mapsto \pi_s$). To this extent, a pair of auxiliary boolean variables $B_{i,j}, B'_{i,j}$ is used, as well as a large-enough constant (which will be noted as M), which is enough when set to an order of hundred thousands.

The condition that needs to be modeled also builds on integer variables e_i, e_j and r_i, r_j that represent tasks' inflated execution time budget and release instant respectively. When overlapping occurs, are affected by the contention effect on the tasks themselves and their predecessors in the scheduling slot. In fact, the time in isolation and the WCD will determine the budget allocated to each task in the slot. Given a task τ_i and the triggering time of its scheduling slot t_{MIF}, we can define a recursive relation to compute the release time of a task, where $pred(i)$ is the predecessor of τ_i, executing right before τ_i on the same core.

$$r_i = \begin{cases} t_{MIF} & \text{if } \tau_i \text{ is the first task in its MIF} \\ r_{pred(i)} + e_{pred(i)} & \text{otherwise} \end{cases}$$

Note that e_i represents the execution time budget assigned to τ_i including the WCD interference (i.e., $e_i = c_i + \Delta_i$). The interval $r_i + e_i$ is thus irrevocably reserved for τ_i execution (as per not work-conserving assumption).

Using this relation, we are able to encode the overlapping condition as a pair of constraints (one for each operand in the disjunction in Eq.4.13) as follows:

$$\Delta_{j \triangleright i} \leq 0 + M * (1 - B_{i,j}) \tag{4.14}$$
$$\Delta_{j \triangleright i} \leq (r_i + c_i - r_j) * \max(a_i, a_j) * l^{\max} + M * B_{i,j} \tag{4.15}$$

and

$$\Delta_{j \triangleright i} \leq 0 + M * (1 - B'_{i,j}) \tag{4.16}$$
$$\Delta_{j \triangleright i} \leq (r_j + c_j - r_i) * \max(a_i, a_j) * l^{\max} + M * B'_{i,j} \tag{4.17}$$

The correctness of the above formulation is shown by cases, focusing on the constraints modeling the first operand (Eq. 4.14 - 4.15) as the same reasoning applies to the dual operand. It can be recalled that $r_i + c_i - r_j \leq 0$ meets the not overlapping condition.

Case $r_i + c_i - r_j < 0$:

If τ_i and τ_j are not overlapping $r_i + c_i - r_j$ will take a negative value. Under all scenarios, the ILP solver will strive to maximize the bounds on $\Delta_{j \triangleright i}$ (i.e., having looser bounds). Therefore, for example, if $r_i + c_i - r_j < 0$, the solver will set $B_{i,j} = 1$ (thus activating the big M), to avoid $\Delta_{j \triangleright i}$ being upper-bounded by a negative number in Eq. 4.15. As a side effect, Eq. 4.14 will set $\Delta_{j \triangleright i} \leq 0$, which is exactly what we wanted to model.

Case $r_i + c_i - r_j = 0$:

In this case the tasks are not overlapping either. In fact, $\Delta_{j \triangleright i} \leq 0$ for any choice of $B_{i,j}$.

Case $r_i + c_i - r_j > 0$:

In this case tasks might be overlapping (depending on the dual conditions in Eq. 4.16 - 4.17). The solver will set $B_{i,j} = 0$, to avoid $\Delta_{j \triangleright i}$ being upper-bounded by 0 in Eq. 4.14. By setting $B_{i,j} = 0$, Eq. 4.15 will simply bound the variable $\Delta_{j \triangleright i}$ to a quantity that is *in all cases* larger than the theoretical maximum contention delay. In fact, a task cannot suffer more collisions than the number of its requests (a_i) or the requests from the contender (a_j), and for each colliding request it cannot incur an additional latency larger than the maximum latency for any request type ($l^{\max}$). That is, $\Delta_{j \triangleright i} \leq x \times \max(a_i, a_j) \times l^{\max}$ is a tautology for any value of x, and will be simply superseded (discarded) by the constraints on access pairing.

Chapter 5

Evaluation

In this chapter, we evaluate the presented approaches by assessing them against other similar and comparable state-of-the-art works in term of tightness of the computed WCD bounds. In doing so, we show that our methods allow deriving a much tighter makespan by relying on two main principles: (i) addressing the aforementioned circular dependence between tasks overlapping and WCD computation; and (ii) considering and tracking multiple memory request types. We also quantify the benefits of splitting tasks into phases, thereby also showing the flexibility of the presented solutions. In the evaluation we will (i) clarify the differences between the two approaches, (ii) identify the respective pros and cons, and (iii) reason on the variability of their results. The assessment of our techniques is completed by a proof-of-concept evaluation on a real board, to demonstrate their effectiveness and industrial applicability.

5.1 Hardware Platform, Access Types and Latencies

Both the iterative and the ILP-based approaches are parametric with respect to admitted access types and respective maximal latencies. In order to run our experiments, we considered the access types and associated latencies exhibited by a concrete hardware platform, while being compatible with the hardware assumptions we made in Chapter 3. The Cobham-Gaisler 4-core LEON4 platform [5], extensively used in the embedded space domain, has been selected as the reference platform to conduct our experiments (both for feeding the models and for running the proof of concept evaluation). In particular, we focused on an FPGA implementation of the LEON4, and exploited the available performance monitoring counters (PMCs) to extract useful information from the tasks memory accesses profiles, necessary to the WCD computation models. The main reason for selecting this platform, is that it is highly representative for the space domain, as it has been extensively adopted in several European Space Agency activities.

The LEON4 (see Figure 5.1) comprises 4 homogeneous cores operating at 250MHz, all counting on L1 private caches for data and instructions. An AMBA bus, implementing

round-robin arbitration, connects the four cores to a shared L2 cache, which can be configured to support different associativity levels (1/2/4), way size (1-256 KB) and AHB bus width (64 to 128 data bits). In the considered configuration, the L2 cache is partitioned, which means that each core counts on its own L2 cache partition to avoid further, inordinate timing interference among tasks in different cores. The L1 data cache implements a write-through, no-write-allocate policy, while the L2 cache is write-back. Requests in the bus block the execution until they are completely served. For that reason, the bus is the main source of contention in this architecture.

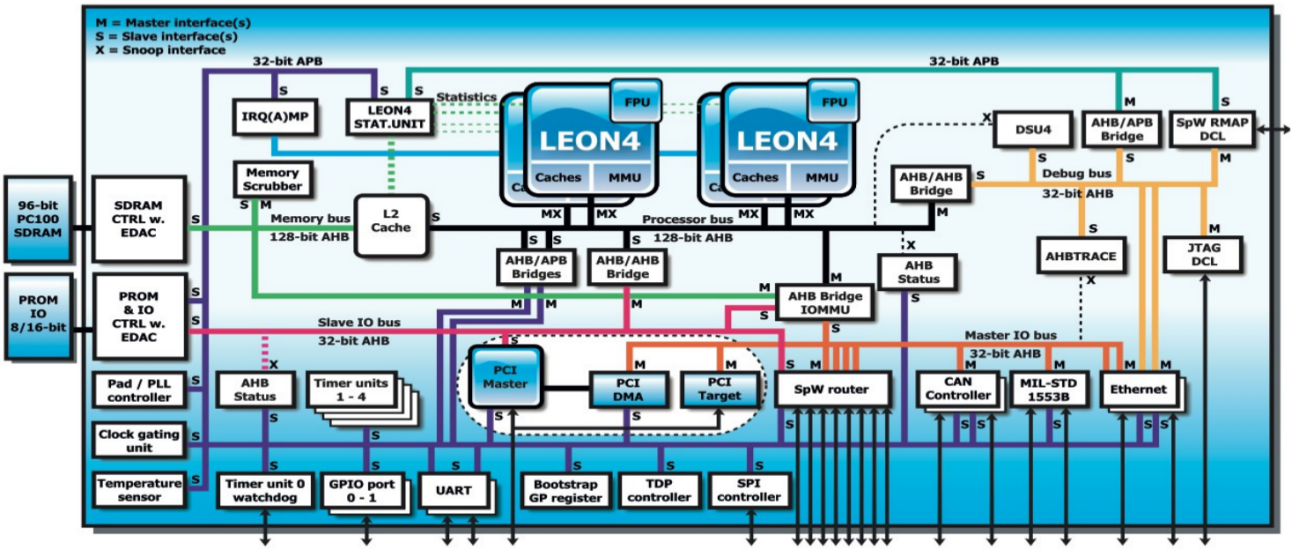


Figure 5.1: LEON4 development board [6]

Different types of memory request can go through the bus to reach the L2 (and possibly the main memory). The following access types are possible in the platform [29]: L2 store hits (*s2h*) and load (*l2h*) hits, L2 clean and dirty misses triggered by load (*l2mc*, *l2md*) and store (*s2mc*, *s2md*) operations. Access types were directly (or analytically) monitored by the performance monitoring counters, while latencies per access type were already experimentally derived in a recent work by Diaz *et al.* [29], exploiting the available PMC support. Table 5.1 summarizes the maximum latencies experimented when each of the described events happen.

In order to monitor the required events, four PMCs from the LEON4 debug support unit have been used, which means that for sure not all relevant events can be directly monitored. Concretely, Table 5.2 lists the bus access types that can be captured with the available PMC support. As can be observed by comparing Tables 5.1 and 5.2, not all access types can be directly tracked with the current PMC support. However, missing counters can be analytically (and conservatively) derived.

Table 5.3 below shows all the possible events that are related to bus accesses. Note that *"l"* and *"s"* stand for load and stores respectively in the superindexes, *"h"* and *"m"* do for hit and miss, and *"c"* and *"d"* stand for clean and miss. The maximum contention delay

Type	mcd [cycles]	Description
sh	$l^{sh} = 1$	L2 store hit
lh	$l^{lh} = 8$	L2 load hit in L2
lmc	$l^{lmc} = 28$	L2 load clean miss
smc	$l^{smc} = 28$	L2 store clean miss
lmd	$l^{lmd} = 31$	L2 load dirty miss
smd	$l^{smd} = 31$	L2 store dirty miss

Table 5.1: Latencies of request type

PMC	Description
pmc^{icm}	Bus reads caused by instruction cache (ic) misses
pmc^{dcm}	Bus reads caused by data cache (dc) misses
pmc^{st}	Writes to L2
pmc^{m}	Misses in the L2

Table 5.2: PMCs available in the reference processor

(mcd) is measured in clock cycles. As already observed, with the current PMC support, we are not able to track all the above events. However, we show how we can analytically derive (or upper-bound) the untraceable events from the available PMCs with a simple set of formulas.

L2 cache	hits (n^h)	misses (n^m)	
		dirty (n^{md})	clean (n^{mc})
loads (n^l)	n^{lh}	n^{lmd}	n^{lmc}
stores (n^{st})	n^{sh}	n^{smd}	n^{smc}

Table 5.3: L2 cache events

First, we observe that the events monitored by the L2 miss counter (pmc^m) do belong to four types of misses, depending on the operation type. Equation 5.1 unfolds the total number of L2 cache misses depending on whether or not the cache miss was triggered in response to a load or store operation, and whether or not it was involving the write back of dirty cache content (dirty or clean miss).

$$pmc^m = n^{lmc} + n^{smc} + n^{lmd} + n^{smd} \tag{5.1}$$

With the current support we are not able to derive the precise number of dirty misses that occurred. Since dirty misses imply a larger delay than clean misses, it is important to be able to precisely (and safely) upper bound their number. In order to do so, we exploit the fact that a dirty miss can only happen if it has been preceded by a store operation (and there cannot be more dirty misses than misses). Equation 5.2 safely upper bounds the number of dirty misses by considering the minimum between store operations (available as pmc^{st}) and the overall number of L2 misses (pmc^{m}).

$$\hat{n}^{md} = min(pmc^{m}, pmc^{st}) \tag{5.2}$$

Consequently, Equation 5.3 uses the bound on dirty misses to derive a lower bound on the number of clean misses (n^{mc}).

$$\check{n}^{mc} = pmc^{m} - \hat{n}^{md} \tag{5.3}$$

In the considered platform, the latency incurred by a cache miss does not depend on the operation type (i.e., load or store). The dirty miss latency is identical for both loads and stores (31 cycles), and the same goes for clean misses (28 cycles). Therefore, it is not necessary to differentiate them.

Instead, since L2 load hits produce a worst case delay of 8 cycles whereas store hits of only 1, the number of load hits must be safely upper bounded. By applying the same reasoning we followed for bounding the number of misses, we observe that there cannot be more load hits than the number of hits or loads themselves. As a result, Equation 5.4 upper-bounds the number of load hits by considering the minimum between the overall number of L2 hits (n^{h}) and the number of loads (n^{l}). Note that the former can be derived by subtracting the number of L2 misses from the total number of bus accesses, while the latter can be derived by summing L1 instruction and data cache miss counters.

$$\begin{aligned} n^{h} &= pmc^{icm} + pmc^{dcm} + pmc^{st} - pmc^{m} \\ n^{l} &= pmc^{icm} + pmc^{dcm} \\ \hat{n}^{lh} &= min(n^{h}, n^{l}) \end{aligned} \tag{5.4}$$

The remaining store hits are thus modeled by Equation 5.5 below:

$$\check{n}^{sh} = n^{h} - \hat{n}^{lh} \tag{5.5}$$

Exploiting the above formulas we can derive all four relevant events that influence the latency of requests in the bus, which is exactly the information we need to evaluate the presented approaches.

5.2 Evaluation of the Proposed Contention Models

After the description of how we obtained the bus event counts and latencies for the LEON4 platform, and before entering into the actual evaluation of the proposed models for the WCD computation, we describe the experimental framework and setup used in the evaluation itself.

5.2.1 Experimental framework and setup

The experiments we conducted mainly consisted in comparing different techniques to compute the WCD for a task set and under a given contention scenario, with the goal to assess how the overall makespan may vary depending on the adopted approach. When not otherwise specified, the analysis is always applied within the scope of a single scheduling slot (MIF) as it is at its boundaries that timing budgets must be enforced. The analysis can be straightforwardly extended to fit the hyperperiod (MAF) boundaries by applying it to each MIF individually.

Experiments were performed on a large number of tasks sets. We assumed a fixed scheduling slot per MIF of 100 milliseconds, which is a representative value for statically scheduled systems [46], to allow defining experiments over varying overall MIF-level utilizations (computed based on tasks' timing requirements in isolation). However, any other slot size can be considered and the same procedure applies. Task sets were randomly generated using the UUniFast algorithm [19]. In particular, 4,000 task sets were generated for each utilization threshold in the range [0.1, 1] with 0.05 steps, summing up a total of 76,000 task sets. The number of tasks produced per task set to fit in each utilization threshold was up to 8. As an example, willing to generate an experiment with a MIF utilization of 50%, the UUniFast algorithm would be used to generate a random number of tasks, so that the sum of their execution time in isolation would be equivalent to the utilization of the MIF (50ms in this very case).

From the standpoint of the inter-core interference, the amount of bus activity is a critical aspect of the problem. The tightness and its improvement with respect to other techniques may vary depending on much memory-intensive the task sets under analysis are. In order to warrant a fair evaluation, some access profiles were derived, based on the memory access and cache statistics of benchmarks in the EEMBC [54] and mediabench [45] suites. Accesses and L2 misses *per kilo (thousand) instructions* are considered, which are abbreviated into APKI and MPKI respectively. For each generated task set, four variants were produced, where tasks were characterized according to the access profiles summarized in Table 5.4. Profiles range from the CPU-bound profile (CPU), relatively robust in terms of contention, to the MEM- and BUS-bound profile ($B+M$), which instead are prone to consistent contention effects. Task's bus accesses are determined proportionally to the number of instructions in the task itself.

Both the ILP and the iterative method models are populated with the data from the

Profile	Description
CPU	≤ 75 APKI ≤ 1 MPKI
BUS	> 75 APKI ≤ 1 MPKI
MEM	≤ 75 APKI > 1 MPKI
B+M	> 75 APKI > 1 MPKI

Table 5.4: Categories created from per-access profiles

task specification and access profiles. Experiments are analogously repeated for all access profiles. For a given utilization threshold, each experiment consists in selecting 4 task sets (one per core in the LEON4): one task set is selected as the focus of the analysis, whereas the 3 remaining sets are assumed to be mapped to the contending cores. The original task set makespan is compared against the one obtained by factoring in the WCD for each task in the set. In particular, what is interesting to be observed here, is the ratio of task sets whose makespan does not overrun the slot boundaries after accounting for the WCD. The tighter the WCD bounds the smaller the increase in the makespan and the lower the probability of overruns.

These solutions have been compared against similar approaches in the state-of-the-art, as will be detailed in the description of the individual experiments. Each experiment is specifically designed to underline an aspect that differentiates the proposed approach from the state-of-the-art.

It is important to emphasize that the inputs required to run the experiments are quite limited. The only inputs required by both approaches are: (i) the number of cycles that tasks take to execute when run in isolation; and (ii) the number of each tracked bus access event, obtained via PMCs readings. All inputs can be derived from the tasks running in isolation (without contention effects).

It is assumed that the number and types of accesses can be derived either by static analysis [65] or by collecting information from the performance monitoring counters, similarly to [25]. The latter approach, which was presented in the previous section, was used in our proof of concept evaluation in Section 5.3. Each access type is characterized by a worst-case latency. In the upcoming experiments, the per-type latencies, provided as an input to the model, are those described and obtained from the LEON4 board. Similar experiments could be derived for any other access types and latencies.

5.2.2 Iterative approach

In the following set of experiments, the tightness of the iterative approach is evaluated. Since it does not rely on any assumption on the application semantics (e.g., separated memory and computation phases) or specific support from the underlying hardware or RTOS, it is first assessed against approaches with similar and comparable assumptions [59, 26, 52]. However, for the sake of completeness, the evaluation also considers a more re-

strictive computational model where tasks execution is divided into phases, to assess the flexibility of the technique to adapt to and support different scenarios.

All the experiments of this approach were run on a laptop system, consisting of a 4x Intel i7-5600U CPU running at 2.60GHz, with 16GB of RAM memory. The approach is implemented by means of a python script, which executes and produces the computed result within a negligible time.

Multiple access types distinction

The first case we wanted to analyzed, is the benefits that can be obtained by distinguishing several types of access in the WCD computation. The approach in [26] assumes a single request type, which is equivalent to assuming that all requests incur the longest (worst) latency, as a safe WCD bound must to be provided. The amount of pessimism stemming from this limitation is platform dependent. In relation to our case and platform of choice, this approach is forcing to assume that all requests take 31 cycles (dirty misses) as shown in Table 5.1.

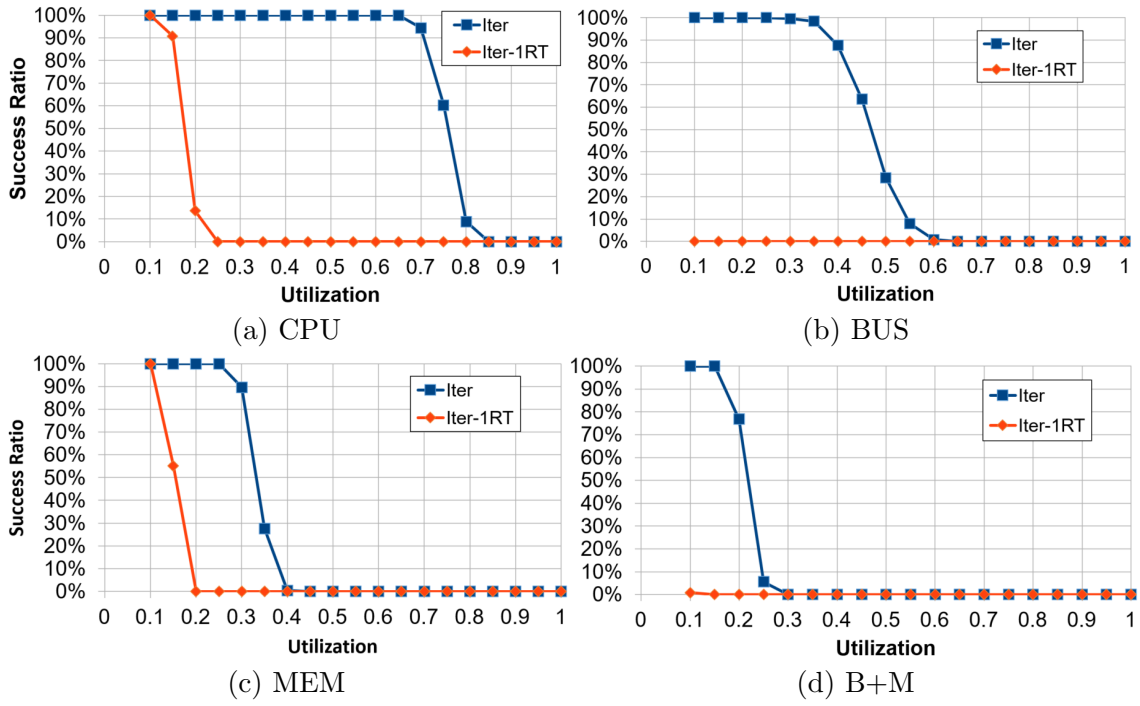
In the next experiments, we compare our iterative approach against its variant considering just one access type, as in [26]. Both approaches are evaluated by observing the ratio of task sets whose timing requirements end up overrunning the schedule slot, when the WCD is added to their timing budget computed in isolation. Figure 5.2 shows the success ratio obtained for each utilization threshold, and all described workload types (CPU, BUS, MEM, and B+M). Each sample consisted in applying the WCD analysis to 1,000 randomly generated task sets (as explained in Section 5.2.1). The results of the iterative approach are referenced to as *"Iter"*, whilst the iterative approach only considering a unique request type of bus access is referred to as *"Iter-1RT"*.

Figure 5.2 clearly reflects the benefit brought by being able to distinguish and track different types of accesses. As expected, under all access profiles, *Iter* outperforms *Iter-1RT*. It is interesting to observe that, for memory intensive access profiles, not differentiating between request types results in not being able to allocate tasks for even a 10% of MIF utilization. This confirms how important it can be to be consider bus access types to avoid excessive pessimism in the WCD computation.

It is worth noting that the utilization threshold refers to the ratio between slot size and tasks WCET in isolation, i.e. not factoring in multicore contention. As a result, as the pressure on the shared resources increases – which mainly happens for MEM and B+M workloads, given that dirty misses and hence accessing the memory is the event which incurs the highest latency – the resulting multicore CPU utilization goes beyond 100% (at core level) simply making the task set not schedulable any more.

Multiple execution phases distinction

To complete the assessment of the suggested approach, we can also experiment on the flexibility of this approach by extending the underlying model to represent execution phases



(a) CPU (b) BUS

(c) MEM (d) B+M

Figure 5.2: Iterative vs *Iterative-1RT*

within tasks. Several studies [53, 52, 12, 17, 20] assume application semantics that clearly separate phases dedicated to data exchange (E) (usually from/to a local on-chip memory) to other devoted to pure computation (C). For instance, it is common that a certain application or program loads data in the first place, then operates and executes on that data, and finally stores back variables and the results of these operations. Phases are usually exploited to devise scheduling solutions aiming at conflict avoidance, but they also naturally constrain the possible task overlappings and, hence, access pairing. We are not interested here in imposing any scheduling restriction, instead, in evaluating the reduction in WCD that can be achieved under a favorable – and more restrictive – scenario, where additional details are available. The baseline of the iterative approach against its adaptation that supports a scenario where tasks are split into three phases is as follows: a relatively large computation phase is preceded and followed by exchange phases, with accesses to shared resources exclusively occurring during the latter. In an exemplary fashion, we can assume a uniform and fixed duration of each phase in the proportion of 20% and 60% of the task execution budget, for the E phases and the C phase respectively. Tasks carry out half of their accesses in each E phase. The evaluation in this case is only restricted to the workloads with highest and lowest pressure on shared resources, i.e. *B+M* and *CPU* respectively. These profiles are already significant enough and extrapolable conclusions can be derived for the intermediate access profiles described.

As expected, splitting tasks into partitioned phases of code where memory exchanges happen, allows for lower WCD bounds. The WCD reduction is entirely ascribable to

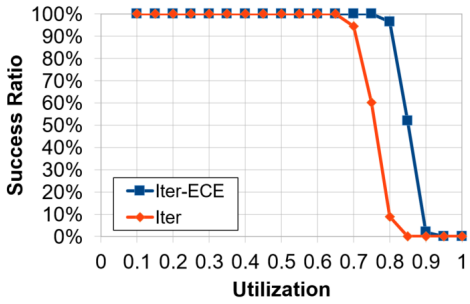


Figure 5.3: Iterative vs *Iterative-ECE* CPU profile

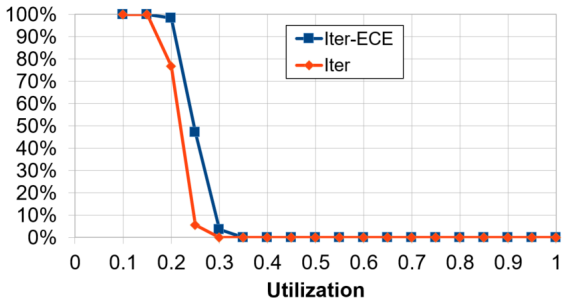


Figure 5.4: Iterative vs *Iterative-ECE* B+M profile

the restrictive assumptions on access distribution. The separation into phases rules out several pairing scenarios that are instead necessarily considered when accesses can happen according to any possible distribution within the task execution, leading in that way to better results.

5.2.3 ILP approach

In this section, the evaluation of the ILP formulation is presented. This multicore contention analysis (named ILP-WCD from now on), focuses on computing system-level WCD bounds rather than task-level ones, as a means to minimize the worst-case *makespan* within each MIF boundary, while providing strong performance guarantees. A shorter (guaranteed) makespan allows, in fact, to safely accommodate more functions within the same schedule slot.

Similarly to the previous experiments, a set of experiments has been analogously repeated for the ILP-WCD, in which the cumulative effect on the MIF makespan of the WCDs computed using this ILP formulation is compared against those obtained with comparable state-of-the-art approaches. The set of inputs employed is the same for each case with respect to the iterative approach.

As opposed to the iterative script, these experiments required considerably more time. For that reason, experiments were in this case executed on a cluster borrowed by the *Universitat Politecnica de Catalunya* (UPC), consisting of a 2x Intel Xeon E5-2630L v4 running at 2.2GHz, with 128GB of RAM memory. Nonetheless, after several improvements and refinements, *ILP-WCD* took only 2 minutes in the average case to compute the cumulative WCD for a given core, using the IBM CPLEX solver [3]. The experimental framework implements a fully-automated generation of the ILP inputs, based on the task set descriptions. Experiments were conducted by running utilization batches in parallel for each given utilization and task set access profile, so the computation process could be significantly reduced.

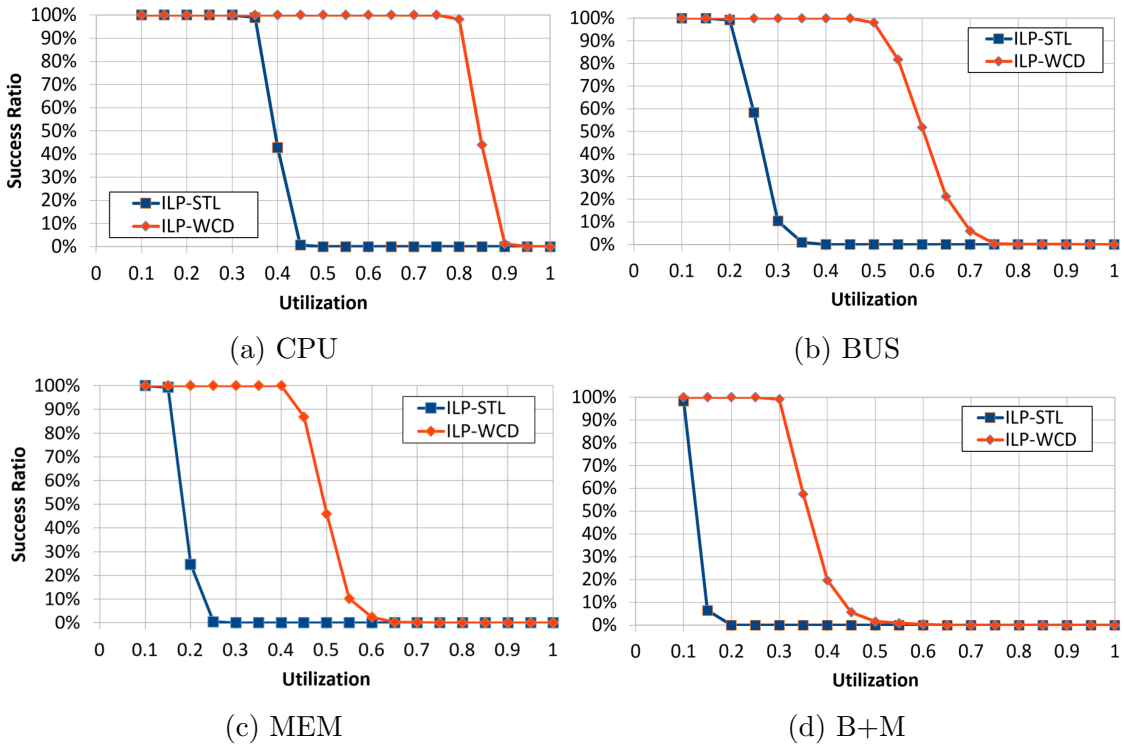
ILP-specific generation process

The evaluation of the ILP-based method required additional steps with respect to the iterative approach. ILP-WCD requires to define an input-dependent ILP model for each experiment. It would not be reasonable to manually encode the ILP model for each experiment, considering the vast amount of tests that had to be run. We automated the experimental framework by developing a *code generator* script (programmed in Python) that automatically generates the required code to be executed with the IBM CPLEX solver [3], provided the inputs of each core in a CSV format are given (randomly generated, in this case). Just to give an idea of the magnitude of an ILP model, only one single experiment with an average of 7 tasks per core would require around 15,000 lines of ILP-specific code for the makespan to be calculated. The implemented code generator sets the experiment up in less than a second. The number of lines of code grows exponentially as we add tasks since all the possible dependencies among tasks have to be encoded. What is more, as it will be noted in the upcoming experiments, the effect of splitting tasks into phases is also analyzed, hence the number of inputs further increases, causing the lines of ILP code to easily reach a magnitude of hundreds of thousands. For that reason, automating this process was of paramount importance.

Once the required ILP code is produced, it can be provided as an input to the ILP solver to calculate the optimal solution (makespan maximization of the core under analysis). The resulting output consists of the maximum number of cycles the makespan could take, and the required collisions of tasks accesses and consequently budgets of each task individually. All this process was automated to produce the thousand of experiments for each access profile and utilization.

Focusing on system-level bounds

The ILP-based WCD formulation proceeds by pairing accesses at system-level, across job boundaries, by considering the sequence of tasks executing in a core. This allows to capture the core-to-core pairing constraints introduced in Section 4.4.2. Mainly, the model benefits with respect to the iterative approach from the fact that we are preventing the accesses of a task to interfere with more than one access on the same interfered core (it can be paired with at most one access per core). To evaluate this aspect, *ILP-WCD* is compared against the baseline approach presented in [59], which instead operates exclusively at task level, without considering the actual contending tasks. The approach in [59] assumes that all accesses in a task can be paired with the accesses of all tasks in each contender core, regardless of the concrete task overlapping actually occurring at run-time. Basically, the approach only exploits task-level constraints and does not consider overlapping conditions. To perform the evaluation, the ILP model was briefly adapted to mimic [59] in this particular setting by disregarding task overlapping and core-to-core constraints. The implemented technique is referred to as *ILP single-task-level* (*ILP-STL*), as it mainly exploits detailed information on the task under analysis but uses only core-level cumulative information on the interfering tasks.



(a) CPU (b) BUS

(c) MEM (d) B+M

Figure 5.5: *ILP-WCD* vs. task-level interference (*ILP-STL*)

We can observe that *ILP-WCD* dominates *ILP-STL*, achieving good success ratios across all utilizations. In fact, the drop in success ratio ('knee' in the figure) for *ILP-STL* occurs in the utilization range $[0.15 - 0.25]$ across all workloads whereas for *ILP-STL* in the range $[0.35 - 0.85]$. Moreover, in terms of workloads, we can also notice that, as the pressure on the shared resources increases (from CPU to B+M) all approaches suffer a proportional reduction in success ratios, always with *ILP-WCD* improving over *ILP-STL*. The source of the pessimism of *ILP-STL* when compared to *ILP-WCD* stems from the fact that the latter works at MIF (makespan) level, preventing that a given request from a given core is paired more than once. Instead, *ILP-STL* works at task level not keeping track of which requests from a given task have already been paired. This assumption is closer to the proposed iterative approach in that sense. As a result, in the worst case, one request of a task in a core can be paired up to K times, where K is the number of tasks in the considered contender core.

We highlight that this method has not been considered in the evaluation of the iterative approach as the latter already assumes a given task overlapping and identifies the set of potential contenders. Therefore, a comparison with an approach which not only considers all feasible alignments but also considers that any task in the system is a contender, would have been unfair and would not have provided additional insights on the method.

Multiple access types distinction

Another interesting characteristic of *ILP-WCD* is that it captures the fact that the latency incurred by a conflicting access typically varies, even within the same resource, as it depends on the type of the interfering request. This proposed solution considers several request types and their associated latencies, as can be observed in all typed constraints in Chapter 4. To evaluate this aspect, ILP-WCD is compared against the technique presented in [26], which improves over [59] by considering information on both, interfering and interfered tasks, but does not consider that interconnects typically exhibit variable latencies, depending on access types. To conduct the evaluation, the ILP was modeled to mimic [26] in this particular setting. Analogously to the previous experiments, this technique will be referred to as *ILP with one-request-type* (*ILP-1RT*).

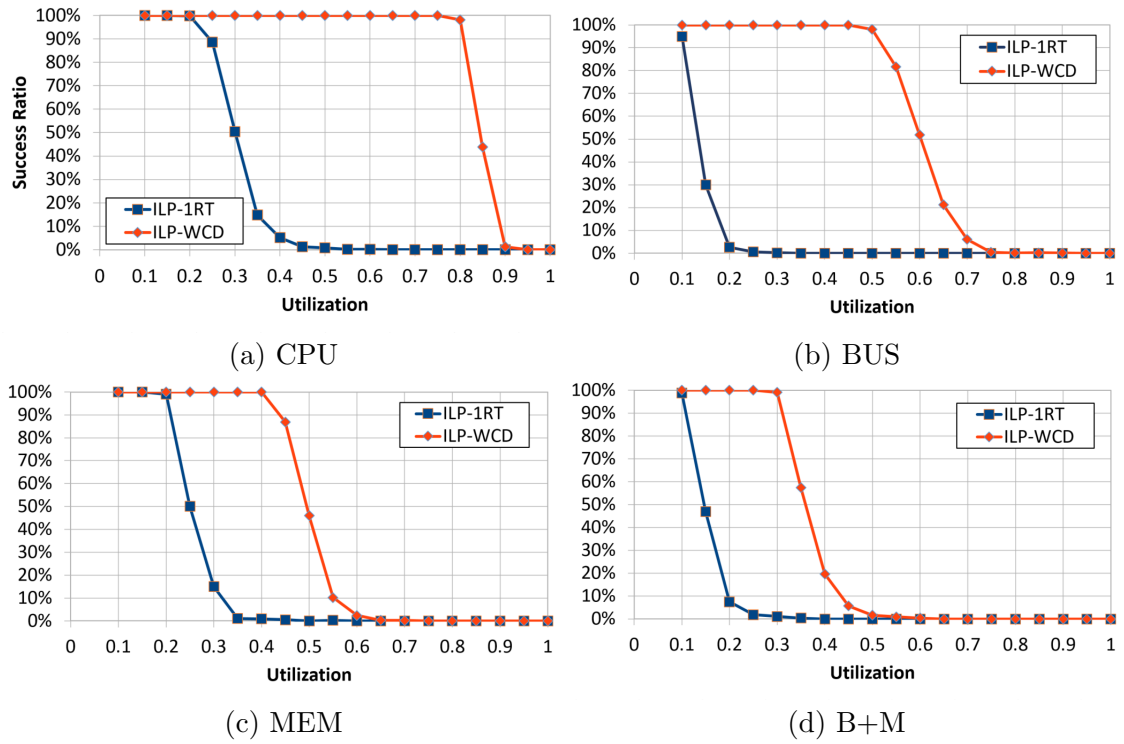
Again, we evaluate the ratio of task sets whose timing requirements do not exceed the schedule slot when the WCD is accounted for. Figure 5.6 shows the success ratio obtained for *ILP-WCD* and *ILP-1RT* for workload types and utilization thresholds. As expected, the *ILP-WCD* model outperforms *ILP-1RT* across all workload types and utilizations. The gap between the two curves identifies the increase in pessimism incurred by *ILP-1RT* with respect to *ILP-WCD*. Interestingly, the pessimism gap between *ILP-1RT* and *ILP-WCD* is more sensitive to the amount of bus requests rather than memory ones. This is explained by the fact that each request is assumed to take the longest latency: for bus requests, either loads or stores, that reach L2 but do not miss in the cache (hence not contributing to the MPKI), the incurred pessimism is larger than that associated to actual memory requests. In other words, *BUS* profiled tasks perform more accesses than *MEM* ones. The worst-case latency $l^{\max}$ in this platform is 31, which corresponds to the latency incurred in case of dirty L2 cache misses (either loads or stores). We can compare the overestimation incurred by each bus request as follows. For stores hits ($s2h$), the pessimism added is $l^{\max} - l^{s2h} = 31 - 1 = 30$ cycles, whereas for loads hits it results in $l^{\max} - l^{l2h} = 31 - 8 = 23$ cycles. Instead, *ILP-1RT* incurs notably less pessimism for L2 clean misses, where $l^{\max} - l^{s2mc} = 31 - 28 = 3$ (the same holds for l^{l2mc}). Finally, no additional pessimism is introduced on dirty misses, as $l^{s2md} = l^{l2md} = l^{\max}$. This behavior can be observed comparing Figure 5.6(b) for BUS and Figure 5.6(c) for MEM. In the former, tasks trigger a larger amount of bus requests than in the latter, resulting in reduced success ratio for *ILP-1RT*.

Multiple execution phases distinction

Finally, it is also shown that as with the iterative approach, this approach is flexible enough to deal with phases of tasks to obtain an even tighter estimate.

As done for the previous experiments, ECE phases are assumed with a proportion of 20%, 60% and 20% respectively as well, assuming bursts of memory accesses at the beginning and end of each task.

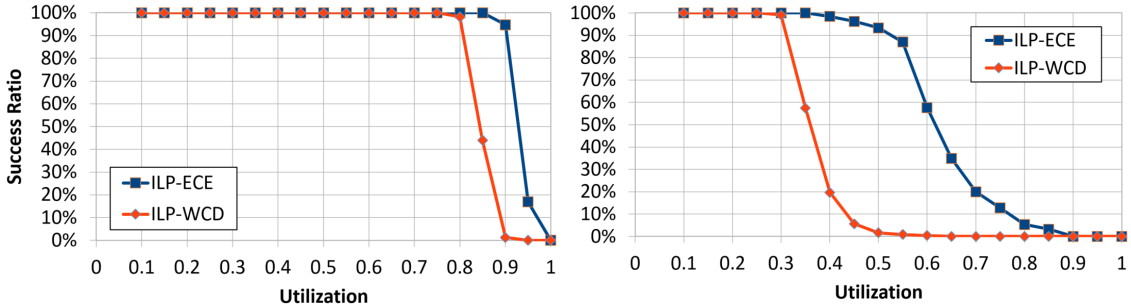
Figure 5.7 compares the success ratio of *ILP-WCD* and *ILP-ECE*. As expected, *ILP-ECE* always outperforms *ILP-WCD*. However, the ECE execution model requires changes



(a) CPU (b) BUS

(c) MEM (d) B+M

Figure 5.6: *ILP-WCD* vs. single access type (*ILP-1RT*)



(a) CPU (b) B+M

Figure 5.7: Comparison of *ILP-WCD* and *ILP-ECE*

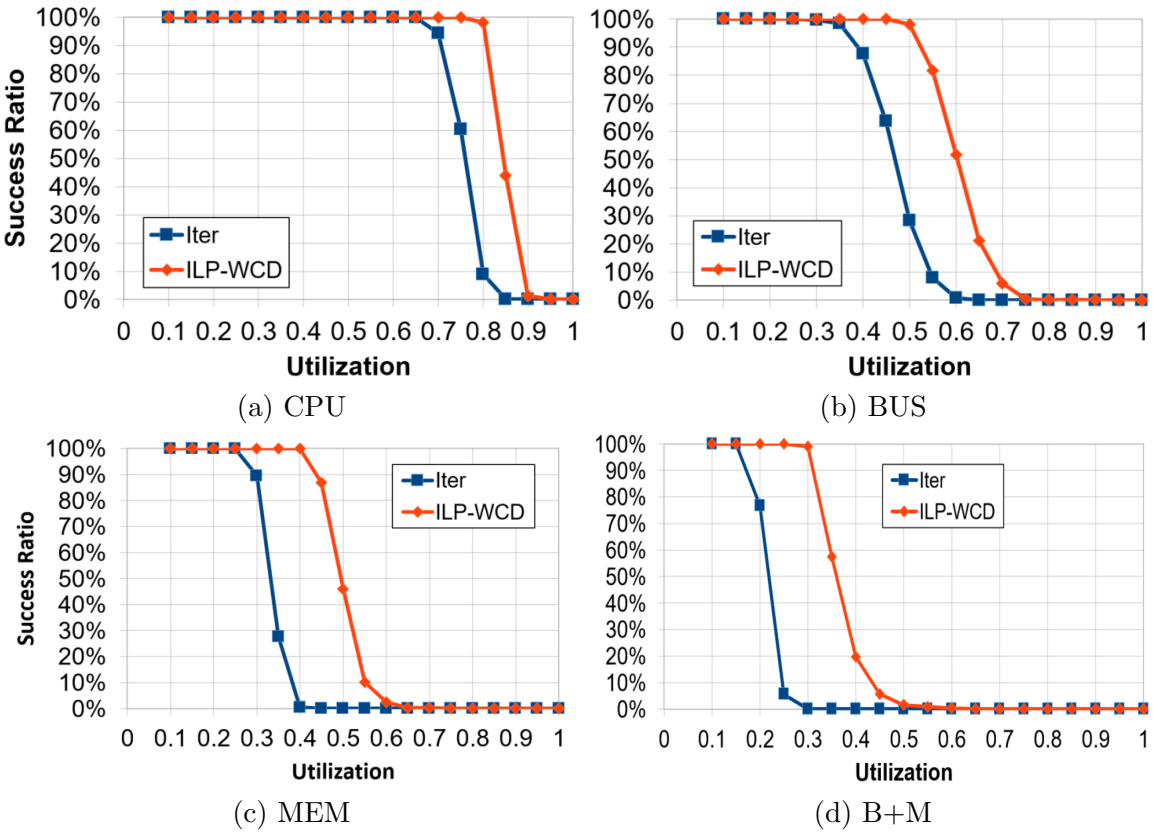
to the application and it is hard to apply on cache-based systems, where accesses to the bus cannot be scheduled at will.

More importantly, this experiment demonstrated that the ILP-WCD formulation also supports it and is flexible enough to adequately model phases with any kind of accesses distribution. For the particular workloads and setup in this experiment, we can see that the improvement of *ILP-ECE* over *ILP-WCD* is considerably larger under the *B+M* workload. This is in line with the characterization of the workloads: *B+M* is meant to incur the highest contention effects and the advantage it takes from the *ECE* setting is proportional

to the number of accesses.

Comparison of the proposed approaches

Finally, we compare the two proposed approaches. Results are reported in Figure 5.8.



(a) CPU (b) BUS

(c) MEM (d) B+M

Figure 5.8: Iterative vs. *ILP-WCD*

As expected, since the ILP method computes system-level bounds and avoids over-accounting of accesses, the obtained WCD bounds outperform those obtained with the iterative approach. It must be noted, however, that the iterative approach's main strength does not lay on the tightness of the overall cumulative bounds at system-level, but on the tasks' budget guarantees. As a result, it can not be as competitive as the ILP in that sense, which aims instead at providing tighter system-level bounds. However, despite the sources of pessimism that are inherently introduced in the iterative method, as detailed before in this chapter, it can be appreciated that it still provides reasonably tight results at significantly lower analysis time. The sources of pessimism in the iterative approach (and hence the slack time between tasks' activation), gets partially compensated by the fact that in ILP we are not discarding any overlapping scenario, which in turn can lead to extremely adverse overlapping and alignment for all tasks in a core.

It can also be noted, that the overall makespan computed differs as tasks' access profiles are higher in memory access (MEM and B+M) terms, which is explained by the fact that the over-accounted safely-reserved budget is higher per each task in the iterative approach (since the system experiences more dirty misses).

5.3 Proof-of-Concept Evaluation

At last, a proof-of-concept is provided. We consider practical applicability as a main concern: it was therefore important to provide evidence that the proposed methods are easily applicable without relying on unrealistic assumptions. Furthermore, an evaluation on a real board served as a whole validation process, from extracting the PMCs during execution in isolation, to defining task set partitions and mapping them to different cores, to executing them under potential contention, and finally, to calculating safe WCD bounds. With respect to the latter aspect, we wanted to provide empirical evidence that real executions on a board can never exceed the calculated bounds (as this would imply that they do not provide safe guarantees) and the provided bounds are at the same time realistic (not exceeding in pessimism). So, in a nutshell, this experiment proves that given a set of tasks running on real hardware, both of the hereby proposed models end up computing a makespan that ensures that the reserved slot is enough to accommodate the timely execution of all applications.

In order to carry out this experiment, the following steps were followed:

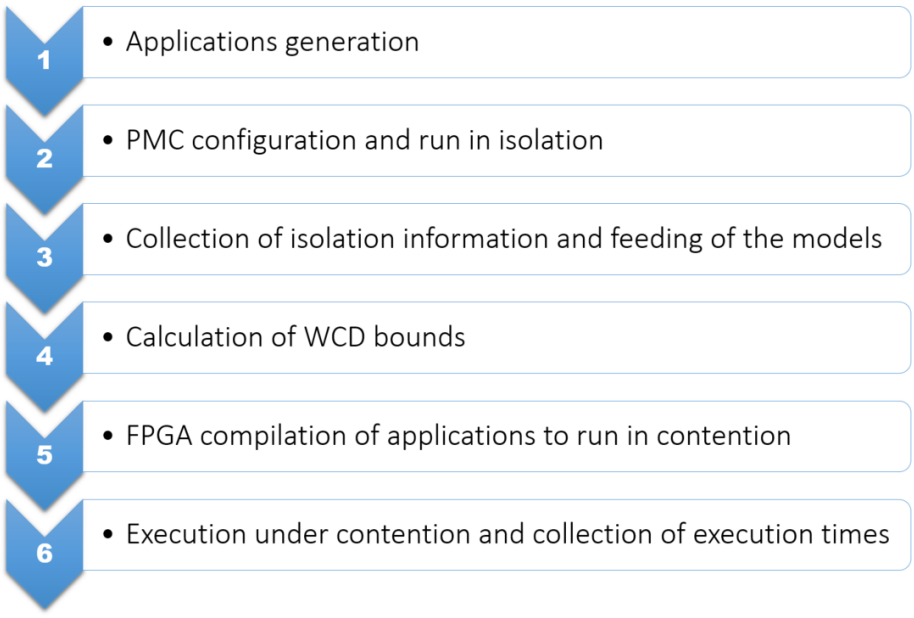


Figure 5.9: Board experiment flow

1. As a first step, an automatic code generator had been configured to generate code simulating application tasks, up to (approximately) half of a MIF utilization (i.e., 50ms). The generated code included both tasks with evenly distributed accesses along their execution, and tasks that performed accesses to memory exclusively at the beginning and end of their execution. The output was in the form of independent C functions, which were joined up in a single sequence (to mimic the static schedule), ready to be compiled and ported to the board.

2. The PMCs of the board were configured to track the events that were of our interest. This was done by means of a library internally developed in the group. Then, each application was compiled and executed in isolation in the FPGA, tracking the number of cycles they took to execute, as well as the bus events, using the aforementioned PMCs.

3. The collected information was provided as an input to both models, in the form of a CSV.

4. By having this information, both approaches were used to calculate their WCD bounds respectively. Additionally, the fTC budgets were computed as a reference. The iterative approach script is configured to provide this information as well, if required.

5. In the subsequent step, the applications were allocated to different memory regions in the FPGA, so that each core could start executing its application concurrently, possibly causing contention.

6. Finally, the applications were run under contention. The only output collected was the number of cycles which each core needed to execute its allocated applications when run in parallel with contenders. To do so, a hardware breakpoint was set at the very end of each application, and the number of cycles obtained when these instants were reached.

As expected, the time was significantly increased. In order to cope with the slight jitter variations which are expected to occur, which in turn could lead to slightly different task overlapping scenarios, the experiments were performed 1,000 times, simulating numerous executions of the same MIF, and enabling fluctuating jitters to play a role in each run. Actually, the variation among runs was quite insignificant as one could have expected. Furthermore, it had been accurately checked that no single MIF was overrun.

In Figure 5.10, a summary of the measured execution times versus the calculated makespans is displayed, relatively normalized to the maximum observed execution time on the board. The median execution time of these runs is presented. Both approaches as well as fTC execution times (used as a reference) are normalized to the real board execution time. For example, if the execution of one MIF took 1.5 million cycles to execute in the board, a 2 in the y-axis would mean 3 million cycles. As it can be appreciated (and as one could have expected), *ILP-WCD* is the method which provides the most realistic (tighter)

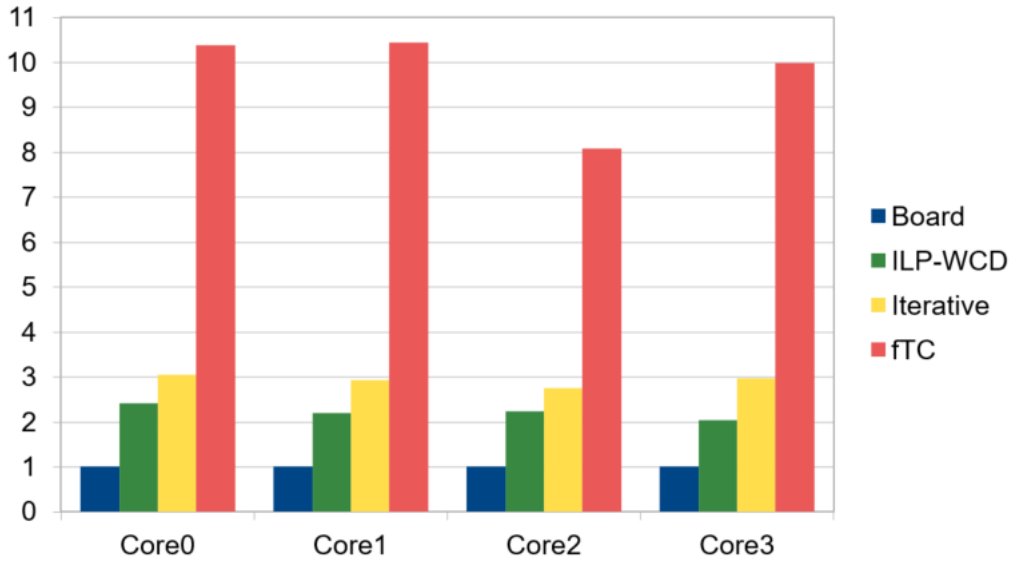


Figure 5.10: Summary of the results

WCD bounds, resulting in approximately just 2 to 2.5 times the maximum observed execution time. It is worth noting that the observed execution times are not representative of the worst-case scenario, as it is not possible to control the experiments to incur the worst-case overlapping of bus requests. However, we must consider the possibility of its occurrence, and hence the enlarged computed bounds.

With regard to the iterative approach, we can observe that it is more pessimistic than *ILP-WCD*, providing an overall budget of slightly less than 3 times the observed one in the board. Interestingly, the difference between both of the hereby presented approaches is not as much as we could have initially thought. This talks in favor of the iterative method, meaning that the sources of pessimism it incurs are not excessively compromising the results, bearing in mind that the ILP solution grants us with the absolutely worst possible alignment considering the model variables.

Much more pessimistic are the fTC bounds. These results would be the ones obtained if we were only able to obtain information of the tasks run in isolation on (i) the total number of accesses that each task performs, and (ii) the longest latency experienced by these accesses.

As a final remark on the results, we want to highlight how tight the results of both approaches are. The benefits of the iterative approach with regard to task-level timing guarantees have already been observed. When it comes to the *ILP-WCD* approach, we have to bear in mind that for each core we were assuming that **every single task access** was always the last one to be granted access to the shared resource, and furthermore, it did so in the worst alignment possible, always accounting for the maximum latency. Despite the restrictive assumptions, the inflated budget (i.e., accounting for the WCD) was close

to 2x the worst-case observed timing on the board, which are inherently far from the worst case. A relatively small difference between static bounds and observed values is thus an indicator of how tight are the bounds we were able to derive on the makespan.

Chapter 6

Conclusions

In this final chapter we summarize the work that has been conducted in the scope of this thesis. We also identify how the proposed approaches contribute to and improve over the current state of the art on the computation of the worst-case contention delay. Finally, we delineate some promising directions which could be explored to further improve the obtained results.

6.1 Tightening the Bounds on WCD Computation

In multicore systems, the way multiple tasks on different cores may overlap in time largely determines the contention they can potentially suffer at run time, when accessing shared hardware resources. Conversely, the delay incurred as an effect of contention changes tasks WCET and response time, ultimately affecting tasks overlapping.

Throughout this thesis, it has been pointed out that whilst multicore and multi-level cache systems can bring up great benefits in timing and performance terms, timing analysis models must be updated to provided safe and tight bounds on the contention effects, to avoid jeopardizing the potential benefits.

This thesis, proposes two complementary approaches for the computation of a bound to the Worst-case Contention Delay (WCD) suffered by tasks in statically scheduled, bus-based, multicore systems: an ILP formulation (ILP-WCD) for the computation of worst-case contention delay for all tasks, deriving very tight system-level bounds; and secondly, an iterative methodology focusing on WCD bound for each tasks in the system, trading some pessimism (i.e. conservative assumptions) for task-level guarantees, as opposed to system-level ones. Within both approaches, particular emphasis is put on capturing the circularity of contention effects on task overlapping.

In the context of statically scheduled, cyclic-executive multicore systems, and in particular of a multicore instance of an ARINC-like system, we highlighted how the capability of ILP-WCD to operate at core level, on the entire scheduling makespan, allows to outperform other techniques that restrict WCD computation to the task level. Lower WCD bounds than those computed with ILP-WCD, can only be obtained by exploiting addi-

tional assumptions on access distribution, which are hardly available in general. This was also the main reason why such assumptions were not exploited in this work. Even the iterative methodology has been shown to be effective in finding a safe task schedule (by computing appropriate tasks' release times). Our evaluation shows that the iterative approach surpasses previous works on WCD tightness. This is explained by the fact that the iterative approach (and the ILP-WCD as well) exploits access information from both the task under analysis and the potential contenders, and distinguishes among different access types.

As a by-product of their evaluation, we also demonstrated the flexibility of our approaches. We showed how they can be successfully adapted to model different WCD assumptions and scenarios, such as those approaches that build on distributing shared resource accesses in different task phases: as part of our evaluation, we showed how easily our models can be tailored to account for additional information that might be available on the system and task set.

Furthermore, a highly automated experimental framework has been implemented, greatly minimizing the time required for performing the experiments and analyzing the results. The designed framework is platform-independent, easily configurable, and does not rely on any RTOS support, requiring as inputs only minimal information on applications run in isolation.

Finally, the performance, safeness and industrial applicability of the presented approaches have been demonstrated on a real hardware platform, detailing all the steps to be followed from scratch for its proper application.

6.2 Recapitulation of this Thesis Contributions

The initially intended contributions have been addressed:

1. Review of the state-of-the-art: in first place, the state-of-the-art of real-time multicore timing analysis especially with regard to inter-core contention modeling has been assessed, outlining its current trends, benefits and limitations of the hitherto proposed works. We have studied why critical real-time embedded systems are usually scheduled statically; what is required in order to limit or control contention effects; which methods exist to account for the WCD and their limitations; and how assessing task-to-core mapping and task-partitioning can effectively reduce contention.

2. A method to compute WCD bounds with task-level guarantees: we have shown how an iterative approach enables us to detect safe and reasonably (in the order of 3 times the observed execution time) tight time budgets for each single task.

3. A method to compute tight WCD system-level bounds: an ILP-based approach exploiting system-level information has been presented to derive even tighter WCD bounds at the level of the overall makespan, as opposed to the task-level method.

4. A fully automated framework for the execution of both approaches has been implemented. The ILP-specific code to each test is being automatically generated and optimized in terms of performance, the iterative method has been implemented by means of a script, and the testing environment eases experiments parallelization.

5. Proof-of-concept: both methodologies have been tested with applications running on real hardware, to asses their performance and industrial applicability.

6. Finally, we have provided a comparative evaluation which positively answers to the initial research question of this Thesis.

6.3 Possible Extensions and Future Directions

The contributions of this thesis can be extended along several lines of work. In the following, we comment on some promising directions:

Using actual overlapping time to limit the number of conflicts. This work could be further improved by quantifying tasks alignment. Currently, a pair of tasks are modelled to either overlap or do not. Depending on that, their accesses are allowed to mutually interfere or not. An improvement could be made in the line of quantifying overlapping, accounting not only for the overlapping scenarios of tasks but also for the time they do overlap. By doing that, the number (and type) of the possible contending accesses to the bus could be restricted. As an example, if two tasks do overlap but even in the most conservative scenario they only run in parallel during 100 cycles, we can restrict our model not to pair ten accesses incurring a delay of 25 cycles each. In the ILP formulation, this could be modelled by means of a variable associated to each task, which would evaluate all the previous jobs execution times. In a similar fashion to what Figure 4.8 depicts, we could be able to obtain the number of cycles tasks overlap, and hence the number and type of allowed access pairs. Regarding the iterative approach, since it consists of a script, this information would be simply encoded in its logic. The safeness of the approach would still be guaranteed, since we would only be ruling out unrealistic scenarios (in the same way we were able to discard scenarios when starting the first iteration from execution times in isolation).

Events distribution and ordering. Another potential source of improvement, would consist in getting more information on the *order* in which tasks' requests are triggered. This additional knowledge could be used to refine the possible collisions of accesses among tasks, by keeping track of the already paired and pending to be paired types of access. For instance, if we would know that the first L2 cache miss does not occur until after ninety load hits, we could restrict this miss to be paired against other accesses before having paired these ninety load hits. The order could be fed into the models and consequently add precedence constraints.

A very similar reasoning could be applied if we had further information on the access distribution. To do so, tasks run in isolation should be further statically analyzed in parts instead of as a whole. For instance, we could split a task into N parts and use PMC values at part level to feed our models. The analysis of the distribution would be something very similar to what has been explored in the experiments when tasks were divided into phases, which as explained, is likely to rule out certain cases potentially leading to worse contention delays.

It has to be noted though, that the solving time would in both of these cases be greatly penalized, especially when it comes to the ILP solution due to the exponential increase in variables to be considered by the solver.

Improved classification of accesses. As we discussed in Section 5.1, not all events can be directly monitored and some conservative bound on access types had to be considered. Additional PMCs (just one counting dirty misses in our case) would have allowed us to precisely derive the number of all the relevant events (access types). That would result (most of the times) in a significant reduction in computed WCD, since the number of events incurring the highest latencies are likely to be reduced.

Bibliography

[1] Cyclic executive scheduling. https://pubweb.eng.utah.edu/~cs5785/slides-f10/22-1up.pdf.

[2] HiPEAC Vision 2017. https://hal.inria.fr/hal-01491758/file/HiPEAC_Vision_2017.pdf.

[3] IBM CPLEX solver. https://www.ibm.com/analytics/cplex-optimizer.

[4] Intel GO Automated Driving Solution Product Brief. https://www.intel.es/content/dam/www/public/us/en/documents/platform-briefs/go-automated-accelerated-product-brief.pdf.

[5] Leon4 datasheet. https://www.gaisler.com/doc/LEON4-N2X-DS.pdf.

[6] Leon4 summary. http://www.sjalander.com/research/pdf/sjalander-dasia2009.pdf.

[7] PikeOS Hypervisor. https://www.sysgo.com/products/pikeos-hypervisor/.

[8] QUALCOMM Snapdragon 820 Automotive Processor. https://www.qualcomm.com/products/snapdragon/processors/820-automotive.

[9] SYSGO PikeOS RTOS. http://www.sysgo.com/, 2018.

[10] WIND RIVER VxWorks MILS Platform 3.0. https://www.windriver.com/, 2018.

[11] J. Abella et al. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *SIES*, 2015.

[12] A. Alhammad, S. Wasly, and R. Pellizzoni. Memory efficient global scheduling of real-time tasks. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 285–296, April 2015.

[13] ARINC. *Specification 651: Design Guide for Integrated Modular Avionics*. Aeronautical Radio, Inc, 1997.

[14] ARM. ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade. https://www.arm.com/about/newsroom/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade.php, 2015.

[15] AUTOSAR. *Specification of RTE Software - AUTOSAR CP Release 4.3.1*, 2017.

[16] A. Baldovin, E. Mezzetti, and T. Vardanega. A time-composable operating system. In *12th International Workshop on Worst-Case Execution Time Analysis, WCET 2012*, pages 69–80.

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

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

[19] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, May 2005.

[20] A. Biondi and M. Di Natale. Achieving predictable multicore execution of automotive applications using the LET paradigm. In *24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2018.

[21] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4):8:1–8:45, December 2010.

[22] A. Burns and A. J. Wellings. A Schedulability Compatible Multiprocessor Resource Sharing Protocol – MrsP. In *25th Euromicro Conference on Real-Time Systems*, pages 282–291, July 2013.

[23] Certification Authorities Software Team. Multi-core Processors - Position Paper. Technical report, CAST-32A, November 2016.

[24] S. Chattopadhyay, C. L. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A Unified WCET Analysis Framework for Multi-core Platforms. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*.

[25] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee. Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. In *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1068–1075, Nov 2011.

[26] D. Dasari and V. Nelis. An Analysis of the Impact of Bus Contention on the WCET in Multicores. In *IEEE 14th International Conference on High Performance Computing and Communication & IEEE 9th International Conference on Embedded Software and Systems*, pages 1450–1457, 2012.

[27] Dakshina Dasari and Vincent Nelis. An analysis of the impact of bus contention on the wcet in multicores. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), IEEE 14th International Conference on*, pages 1450–1457. IEEE.

[28] E. Diaz, E. Mezzetti, L. Kosmidis, J. Abella, and F. J. Cazorla. Modelling Multicore Contention on the AURIX™ TC27x. In *Design& Automation Conference (DAC)*, 2018.

[29] Enrique Díaz, Mikel Fernández, Leonidas Kosmidis, Enrico Mezzetti, Carles Hernandez, Jaume Abella, and Francisco J Cazorla. MC2: Multicore and Cache Analysis via Deterministic and Probabilistic Jitter Bounding. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 102–118, 2017.

[30] N. Diniz and J. Rufino. ARINC 653 in Space. In *DASIA - Data Systems in Aerospace*, ESA Special Publication, 2005.

[31] G. Fernandez, J. Abella, E. Quiñones, C. Rochange, T. Vardanega, and F. J. Cazorla. Contention in Multicore Hardware Shared Resources: Understanding of the State of the Art. In *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 31–42. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014.

[32] Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Tullio Vardanega, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. Introduction to partial time composability for COTS multicores. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1955–1956, 2015.

[33] Rafia Inam, Nesredin Mahmud, Moris Behnam, Thomas Nolte, and Mikael Sjödin. The multi-resource server for predictable execution on multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 1–12.

[34] Intel. Next-Generation Transportation. http://www.intel.com/content/www/us/en/automotive/automotive-overview.html., Intel Press Release, 2017.

[35] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

[36] J. Jalle, M. Fernandez, J. Abella, J. Andersson, M. Patte, L. Fossati, M. Zulianello, and F. J. Cazorla. Bounding Resource Contention Interference in the Next-Generation

Microprocessor (NGMP). In *8th European Congress on Embedded Real Time Software and Systems (ERTS)*, 2016.

[37] J. Jalle, M. Fernandez, J. Abella, J. Andersson, M. Patte, L. Fossati, M. Zulianello, and F. J. Cazorla. Contention-aware performance monitoring counter support for real-time mpsocs. In *11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, May 2016.

[38] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Bus-aware multicore WCET analysis through TDMA offset bounds. In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 3–12. IEEE.

[39] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, April.

[40] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding and reducing memory interference in cots-based multi-core systems. *Real-Time Syst.*, 52(3):356–395, May 2016.

[41] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 1st edition, 1997.

[42] Hermann Kopetz. Event-triggered versus time-triggered real-time systems. In *Operating Systems of the 90s and Beyond*, pages 86–101. Springer, 1991.

[43] Hermann Kopetz. The time-triggered model of computation. In *RTSS*, page 168. IEEE, 1998.

[44] Kwei-Jay L., Swaminathan N., and Jane W.-S. Liu. Imprecise results: Utilizing partial comptuations in real-time systems. In *RTSS*, 1987.

[45] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *30th Annual International Symposium on Microarchitecture*, pages 330–335, 1997.

[46] D. Locke, David R. Vogel, L. Lucas, and J. Goodenough. Generic avionics software specification. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.

[47] S. Martinez, D. Hardy, and I. Puaut. Quantifying WCET reduction of parallel applications by introducing slack time to limit resource contention. In *International Conference on Real-Time Networks and Systems (RTNS)*, International Conference on Real-Time Networks and Systems, October 2017.

[48] A. Melani, R. Mancuso, M. Caccamo, G. Buttazzo, J. Freitag, and S. Uhrig. A scheduling framework for handling integrated modular avionic systems on multicore platforms. In *IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, Aug 2017.

[49] Mircea Negrean, Simon Schliecker, and Rolf Ernst. Response-time analysis of arbitrarily activated tasks in multiprocessor systems with shared resources. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 524–529. European Design and Automation Association, 2009.

[50] J. Nowotsch, M. Paulitsch, D. Bhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In *26th Euromicro Conference on Real-Time Systems*, pages 109–118, July 2014.

[51] P. J. Parkinson. Multicore mils. In *9th IET International Conference on System Safety and Cyber Security*, pages 1–8, 2014.

[52] 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 *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, April 2011.

[53] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of CPU and I/O Transactions in COTS-Based Embedded Systems. In *Real-Time Systems Symposium*, pages 221–231, Nov 2008.

[54] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.

[55] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*, 2007.

[56] B. Rouxel, S. Derrien, and I. Puaut. Tightening Contention Delays While Scheduling Parallel Applications on Multi-core Architectures. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1 – 20, October 2017.

[57] RTCA and EUROCAE. *DO-178C / ED-12C, Software Considerations in Airborne Systems and Equipment Certification*, 2011.

[58] Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. The shift to multicores in real-time and safety-critical systems. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, pages 220–229. IEEE Press, 2015.

[59] S. Schliecker, M. Negrean, and R. Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Conference on Design, Automation and Test in Europe*, DATE, pages 759–764, 2010.

[60] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Response time analysis on multicore ecus with shared resources. *IEEE Transactions on Industrial Informatics*, 5(4):402–413, 2009.

[61] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proceedings of the conference on design, automation and test in Europe*, pages 759–764. European Design and Automation Association, 2010.

[62] A. Schranzhofer, R. Pellizzoni, J. J. Chen, L. Thiele, and M. Caccamo. Worst-case response time analysis of resource access models in multi-core systems. In *Design Automation Conference*, pages 332–337, June 2010.

[63] Ken W Tindell, Alan Burns, and Andy J Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[64] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), April, 2016*, pages 1–12.

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

[66] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, pages 55–64, 2013.

[67] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM Sigplan Notices*, volume 45, pages 129–142, 2010.