ITER: An ITERative Approach for Inter-Core Timing Analysis in Statically Scheduled Cyclic Executive Systems on COTS Multicore Platforms for CRTES

Xavier Palomo [1*] and Carlos Molina [1*]

[1]Department of Computer Science and Mathematics, Universitat Rovira i Virgili, Tarragona, Spain. Phone: (+34) 977558509.

*E-mails: xavier.palomo@urv.cat; carlos.molina@urv.cat;

Abstract

Off-the-shelf multicore systems are increasingly used in real-time fields, but hardware resource contention remains a key challenge. This work focuses on statically scheduled systems, common in real-time computing, and introduces a novel model for calculating worst-case contention delays. Under realistic assumptions, our model provides tight upper limits on these delays, considering the timing interference from other cores and their task sets. We present comprehensive evaluations demonstrating that our approach yields more accurate contention bounds compared to existing solutions. Additionally, we introduce an automated framework that simplifies the integration of our model into real-world scenarios, making it practical for industry professionals. This work not only tackles a critical issue in real-time multicore systems but also offers a precise, practical solution for managing inter-core timing interference.

Keywords: WCET, Contention, COTS, CRTES, Scheduling

1 Introduction

The broad concept of critical embedded systems refers to a diverse array of embedded systems across multiple industrial domains, unified by shared demands for safety, security, and financial feasibility or a blend thereof. Critical Real-Time Embedded Systems (CRTES) are a subset of these systems that are designed to adhere to strict timing requirements. The reliability and suitability of such systems hinge not only on computational functionality, but also on the timeliness of the computation. These systems usually operate under soft or hard timing constraints.

Interestingly, and somewhat unexpectedly, CRTES have become an ubiquitous part of our lives [1] and can find application in various domains such as automotive, avionics, railways, medical systems, etc. Any malfunction of a critical embedded real-time system, depending on its application, can have disastrous consequences, potentially resulting in loss of life or significant economic damage. Consequently, domain-specific qualification and certification standards establish the benchmarks that need to be met in the development of CRTES. Examples of such standards include DO-178B [2] for avionics and ISO26262 [3] for automotive applications. Although CRTES certification standards may differ, their stringent demands on verification and validation (V&V) activities consume a substantial portion of the time and resources devoted to the software development process. In regards to timing correctness, V&V standards mandate reliable assurances on the timing behavior of software functionalities. Timing analysis techniques are typically employed to ascertain Worst-Case Execution Time (WCET) limits for each function in a system. These WCET boundaries are subsequently used in schedulability analysis to ensure that a particular set of functionalities can be effectively scheduled on a given system without exceeding time limits [1].

All CRTES domains are experiencing an unparalleled surge in performance requirements and computational complexity to accommodate advanced, next-generation features such as autonomous driving systems [4, 5] in the automotive domain, or increasingly machine-learning applications. To satisfy these evolving requirements, even the most conservative CRTES are transitioning from relatively simple single-core platforms to sophisticated modern heterogeneous multicore and manycore systems [58] with a range of advanced acceleration features. These transitions aim to provide the required computational power with low Size, Weight and Power (SWaP) solutions. Specifically, CRTES are considering Commercially Available Off The Shelf (COTS) hardware platforms as their reference solution. In addition to performance, COTS are viewed as offering superior cost, flexibility, and time-to-market advantages [6–8] over custom hardware platforms.

Owing to their reliance on accurate timing bounds to ensure hard system deadlines are met, CRTES regard time predictability as an essential factor. Because of their real-time requirements, timing emerges as a critical non-functional property: hence, offering robust timing assurance becomes a fundamental concern in critical real-time systems. Several strategies have been proposed to provide reliable guarantees on the timely execution of software functions [9]. Though not universally, these techniques have been incorporated by domain-specific certification standards. For single core systems, timing analysis primarily involves limiting the execution time of an isolated task or program on a given hardware platform, encompassing some form of low-level architectural analysis and high-level program characterization. The determined bounds are then used to efficiently schedule the task set, ensuring all tasks can meet their deadlines, given the tasks' timing requirements and dependency information. However, when considering multicore architectures, contention on shared hardware resources introduces an additional crucial factor.

Cores within a multi or a manycore system inevitably share some common hardware resources (such as part of the memory hierarchy, the interconnect, I/O, etc.). Proper management of access and usage of these resources is essential. Since multiple cores can potentially (and concurrently) access the same hardware resource, such as the memory controller or the bus, there may be instances where a requested resource is unavailable because it is already serving requests from other cores. The requesting core must then wait for the other core to release the resource, typically arbitrated by a (predictable) protocol. An increase in the number of cores amplifies potential contention

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

effects, jeopardizing predictability. Providing robust performance guarantees on Commercial Off-The-Shelf (COTS) hardware is complicated by the fact that contention can disproportionately affect tasks' execution time, leading to a significant increase in the response time of a program [10, 11]. Although it is advisable to counteract and mitigate all inter-core interference channels [12], no COTS hardware currently exists that can eliminate all sources of interference [13]. Therefore, if timing interference across cores is generally unavoidable, its effect on task execution time must be safely and accurately accounted for to ensure that the timing budgets assumed and enforced by the scheduling framework are never surpassed in runtime.

Numerous algorithms and methodologies have been proposed to calculate the worst-case inter-core interference [14–16], often referred to as Worst-case Contention Delay (WCD), which a task could potentially incur. Including WCD bounds in the allocation of time budgets helps ensure tasks will meet their deadlines, even in the face of contention. However, contention effects are challenging to precisely characterize as they depend on when (and how frequently) the program and its co-runners access a given shared resource. Strategies for the computation of the WCD must rely on conservative assumptions, which often result in excessively pessimistic WCD bounds. Pessimism subsequently translates into a decrease in the system's guaranteed performance—an undesirable outcome in embedded systems. Hence, the research question that arises and that we aim to address in this work is as follows:

Can we establish safe and tighter boundaries on the worst-case contention delay on cyclic-executive, time-triggered multicore platforms compared to state-of-the-art works?

Answering this question, our research provides a methodology that aims to establish safer and tighter boundaries on the worst-case contention delay in cyclic-executive, time-triggered multicore platforms compared to existing state-of-the-art works. We introduce a novel analytical technique that considers the interplay between timing interference from concurrent conflicting requests from other cores and the specific set of tasks operating on those cores. By doing so, we can offer more precise calculations of worst-case delay due to contention over shared hardware resources, which can lead to tighter bounds on contention delays compared to existing solutions. Our approach takes into account worst-case task overlapping and request distribution scenarios simultaneously, allowing for a more accurate assessment of contention effects. This increased precision in estimating worst-case contention delay contributes to safer and more reliable timing assurance for critical embedded real-time systems operating on multicore platforms. Therefore, our research aims to improve the quality of timing analysis and provide tighter boundaries on worst-case contention delay, ultimately enhancing the predictability and safety of cyclic-executive, time-triggered multicore platforms in critical embedded applications.

The rest of this work is organized as follows: Section 2 reviews and analyzes the related works on multicore contention modeling, with emphasis on how we distinguish our approach from them. Section 3 outlines our assumptions, introduces the framework on which this work is based, and presents the formal notation used to describe our analytical model. Section 4 introduces and discusses our solution, which is subsequently evaluated extensively and compared with other existing approaches in Section 5. Finally, Section 6 summarizes the main conclusions and outlines the potential future directions to refine and extend the line of research established by this work.

2 Related Work

The increasingly prevalent transition from single to multicore systems in critical embedded real-time domains brings disruptive effects to the consolidated practice for V&V in general, and timing analysis in particular. Over the past few decades, numerous efforts have been made to capture the effect of hardware resource sharing in multicore systems [17], which has been unequivocally identified as the main source of timing interference. In this work, we do not consider the interference resulting from software resource sharing and inter-core synchronization protocols, for which analyzable protocols have already been proposed (e.g., [18]).

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, primarily focusing on a shared bus as the most common and extensively studied source of contention. Proposed approaches have been based either on precisely analyzing the effects of contention or attempting to control or eliminate them altogether. In the following, we briefly review the most relevant techniques, particularly in relation to our approaches. We will first define interference and the contention delay that it produces. Then, we will review works which have proposed approaches to remove or limit the contention effects. Approaches which do not aim at reducing contention but quantifying it will be reviewed later. Following that, we will go over similar works but which are based on different scheduling assumptions, others that enforce quotas for shared resources among tasks, and finally, how other works have utilized performance monitoring counters as well.

2.1 Inter-Core Interference and Contention Delay

To better understand the focus of this work, we categorize inter-core timing interference as direct and indirect. Direct interference refers to the delay a task τi may experience when accessing a shared hardware resource because another task τj, executing on a different core, is using that resource (or will be granted access to it before τi). Thus, direct inter-core interference captures the effects of contention when accessing a resource. Unless otherwise specified, we will refer to this type of interference, and the contention delay accounted for will only be produced by direct interference. However, inter-core interference is not limited to contention effects but may include other types of indirect interference. For example, a co-running task might cause the eviction of useful cache content from a shared cache, which will subsequently cause the analyzed task to incur additional cache misses (compared to execution in isolation). Indirect interference is indeed a significant problem in some types of real-time systems [19].

Precisely characterizing the effects of indirect interference, however, is a challenging task. Interference cannot usually be bounded without resorting to strong conservative assumptions, leading to a high degree of pessimism in the results [20]. 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 mechanisms are typically deployed.

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

2.2 Controlling or Removing Contention Effects

The current literature shows a strong trend towards reducing or even entirely avoiding inter-core interference. Numerous studies build on the PRedictable Execution Model (PREM) [21]—comprising read, execute, and write phases—to prevent (or at least minimize) simultaneous bus accesses of different tasks. This model assumes that the instances at which tasks access memory can be predicted and isolated into phases. Moreover, it proposes that these accesses are not evenly distributed throughout the task execution, but occur in bursts: a read phase, an execution phase, and finally, a write phase. This model generally relies on loading the task working set into local scratchpad memories, and presupposes some degree of resource partitioning in the platform. If these assumptions hold, contention effects can be reduced by aligning task phases so that read and write phases do not occur simultaneously on different cores. This means a task performs its bus accesses during the computation time of its contenders. Such a model typically works in conjunction with task-to-core mapping objectives.

Similarly, authors in [22] try to avoid overlapping task phases with high memory exchange rates. Blagodurov et al. [14] suggest a task-to-core mapping algorithm based on several heuristics and classify tasks according to their miss rates, avoiding tasks with high cumulative miss rates running together. Similarly, [23] proposes an algorithm based on bank partitioning and mapping

of memory-intensive tasks to the same core. By partitioning the banks, contention can potentially be reduced since each core accesses different memory regions without interfering with others. Our work primarily differs from these in that we do not necessarily assume isolation of phases will be possible, and we do not address task-to-core mapping, although our model could be adjusted to support these assumptions too.

A scheduling framework for avionics multicore platforms is presented in [24]. The 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 proposed, 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 approach does not build on a preemptive model; thus, an identification of critical parts within tasks is not addressed.

The work in [25] 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, this method builds on pessimistic task-level bounds. Further, it does not support pairwise mapping of different request types.

Authors in [26] 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. Apart from the differences stemming from the memory-controller perspective, we share some similarities with this work, mainly regarding 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. Though striving to reduce potential contention is clearly a reasonable approach, it can lead to a reduction in flexibility as it relies on strong assumptions about application semantics. Typically, avoiding task access collisions builds on a phased task model, which is not always sustainable. For this reason, our work focuses on cases where this is not possible (for instance, in legacy systems), or when we need to quantify the maximum contention delay that the system can suffer, even if phased task models are possible.

The comprehensive work presented in [27], offers a profound analysis of hardware Quality of Service (QoS) mechanisms within the Xilinx Zynq UltraScale+ MPSoCs. Their qualitative and quantitative insights into QoS provide a valuable foundation for understanding and leveraging these mechanisms in critical systems, such as those in avionics and automotive domains. Particularly, the paper's elucidation of individual and coordinated QoS mechanisms shines a light on the complex interplay of traffic control within multicore architectures. In contrast, our research takes a different trajectory, one that is not predicated on the presence or exploitation of QoS mechanisms. While [27] seeks to master the QoS features to control and mitigate contention effects, our approach is designed to accommodate systems where such hardware-assisted QoS controls may not be present or cannot be utilized to their full potential. Our methodology focuses on a software-based iterative process for the calculation of WCET, which is inherently system-agnostic. Instead of attempting to control or remove contention effects, we aim to model and account for them accurately within the WCET analysis. This approach provides a means to predict contention delays and compute WCET in environments where contention is a given, not a variable to be managed or mitigated through hardware support. This fundamental difference underscores the versatility of our approach, especially in scenarios where hardware QoS cannot be guaranteed or is non-existent. By not relying on hardware QoS, we extend the applicability of our method to a broader spectrum of real-time systems, offering a solution that maintains system predictability and timing correctness in the face of inevitable hardware contention.

Differing from the Multicore Response Time Analysis (MRTA) framework in [28], which is applied to ARM Cortex A5 architectures under fixed-priority preemptive scheduling with an assumption of constant memory access delays, our iterative approach is tailored for the LEON architecture using cyclic executive scheduling. Unlike the MRTA framework, which simplifies

shared resource contention by assuming immediate bus access without contention-induced delays, our method meticulously accounts for the dynamic nature of bus contention. It assesses the impact of variable request latencies on task execution times, enhancing the precision of WCET estimations in critical real-time systems where predictability of task execution amidst resource contention is paramount.

Furthermore, when compared to the detailed models of bus arbitration within the Kalray MPPA-256 architecture described in [29] and the performance enhancements noted in [30], our approach offers distinct advantages. It is adapted to a different hardware environment without the intricate hardware QoS mechanisms or multi-level arbiter featured in the Kalray system. Our methodology not only adjusts for varying latencies across different request types but also introduces the concept of 'pairing', directly addressing bus contention. This approach avoids the assumption of constant delays for memory retrieval, providing a nuanced perspective on contention-induced delays. This tailored approach to WCET computation underscores our focus on flexibility and adaptability to specific hardware configurations and real-time constraints, including the potential to incorporate task phases with predominant memory accesses as demonstrated in our experimental evaluations.

2.3 Accounting for the WCD and WCET

There are several approaches that attempt to analyze and derive tight bounds on the Worst-Case Delay (WCD) that leads the to Worst Case Execution Time (WCET) under various assumptions. We review some of these works in this subsection, paying special attention to two trends that are similar to our approach.

The work in [31] operates exclusively at the task level. The authors consider the worst alignment for each task to compute the WCD, but do not take into account the mapping of tasks to cores, hence they do not consider the scope of the overall system. They do not consider the actual real (or potentially possible) alignment of co-runners. Moreover, they are forced to assume the largest latency request of contenders at each point, as every running task of the system (on other cores) is potentially a contender. This is equivalent to considering the type of bus access that incurs the worst latency, adding considerable 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.

Early works focused on bounding the number of requests to a shared hardware resource for each single task in isolation: assuming a uniform distribution of accesses, with a minimum distance between consecutive accesses [32], or assuming dedicated task phases [33]. 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 [34], [16], 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. This information is then 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. A common trait in these approaches is that contention analysis is primarily focused on the activity of co-runners, rather than on the computation of co-runners of the task under analysis.

[16] 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. 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 our 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, the task under analysis' and co-runners' requests. Our proposal will be compared to this work, to show how considering tasks' bus requests can tighten the results.

Another approach [35] introduces parametric WCET analysis for real-time systems, focusing on the impact of procedure arguments on control-flow. Building upon prior works on symbolic WCET computation and abstract interpretation, the method automatically generates a WCET formula from binary code, representing the execution time as a function of procedure arguments. The approach demonstrates adaptability, embeddability, and automation, offering potential benefits for adaptive scheduling in real-time systems. However, concerns arise regarding the potential computational overhead introduced, posing challenges for its suitability in Critical Real-Time Embedded Systems where minimizing latency is crucial. In CRTES, where meeting strict timing constraints is paramount, any additional computational burden poses a risk of impeding the system's ability to deliver predictable and timely responses. Some other works [36] [37] share a common emphasis on WCET analysis for real-time embedded applications, particularly addressing the challenges posed by multithreaded programs. Particularly, [36] introduces a static analyzer designed to operate across various programming environments, transforming multithreaded programs into equivalent Hoare's Communicating Sequential Processes (CSP) specification program. On the other side, [37] highlights the relative scarcity of WCET analysis efforts for multithreaded programs compared to sequential ones by introducing a principle to enhance the precision of shared instruction cache analysis by shrinking the set of interferences using static analysis extended to barriers. However, both of them may not be entirely suitable for CRTES as they introduce complexities that might lead to increased computational overhead, potentially compromising the stringent latency and predictability requirements of CRTES. Adapting these methods for CRTES might require further optimization and validation efforts to ensure compatibility with the specific demands of critical real-time systems.

2.3.1 Modeling the distributions of accesses to the shared resource

Beyond the quantity, type, and latency of the interfering and interfered accesses, the timing of these accesses is a critical consideration. With information regarding the timing of accesses, one can devise more favorable task alignments and core mappings. Moreover, specific timing information negates the need to assume the worst-case alignment scenario, which can lead to overly pessimistic conflict assumptions.

However, it is practically impossible to precisely acquire this timing information, making it a significant source of pessimism in these models. Nevertheless, different approaches have been devised with the aim of minimizing this pessimism. As previously discussed in Section 2.2, some approaches rely on an execution model where task execution is divided into memory and computation phases. Accesses to shared resources, such as memory systems, exclusively occur during the memory phases. This assumption has been utilized in many works [10], [21], [38], [39], [22] that have more optimally addressed task-to-core mapping in comparison to full task considerations. Similarly, the work of Rosen et al. [40] assumed that memory accesses occur only at the beginning and end of tasks, and they scheduled tasks to avoid overlaps in memory phases.

Other approaches assume that tasks' accesses to shared resources are evenly distributed over the task's execution [26], [31]. In reality, this is often not the case, and is hard to validate.

Although the assumption of phased execution may be reasonable for parallel applications, it lacks generality and is incompatible with the use of legacy code. Regardless of the assumptions regarding access distribution, the approach presented is flexible enough to model different application characteristics and to apply contention analysis at various granularity levels (i.e., full tasks, task sections or phases, etc.) as will be explored in 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 Machine Learning and Probabilistic Methods

Recent advancements in WCET estimation are reshaping approaches to real-time system design. One notable direction involves Measurement-Based Timing Analysis (MBTA), as explored in [41]. This study addresses the complexities introduced by multi-core systems, instruction pipelines, branch prediction, and cache memory. By leveraging machine learning, the correlation between

cycles per executed instruction and hardware-related events is established, providing a practical alternative to traditional static timing analysis. Another approach, explored in [42], is the development of WE-HML, a hybrid WCET estimation technique. Addressing the challenge of limited knowledge about processor microarchitecture, WE-HML combines static techniques with machine learning, operating directly on binary code for enhanced precision. The focus on cache modeling yields substantial improvements in WCET estimates, particularly on ARM Cortex-A53 processors. Furthermore, [43] addresses the critical need for early WCET determination in the initial stages of system development. Utilizing machine learning models at the source code level, this approach successfully predicts WCET without relying on hardware specifics or compiled binaries. These collective endeavors contribute valuable methodologies for more accurate and practical WCET estimation, considering the intricate landscape of modern processors and emphasizing the significance of early insights in real-time systems development. The approach of estimating WCET early in the development cycle at the source code level through machine learning models offers a promising perspective for real-time systems. However, the accurate prediction of WCET at this early stage becomes challenging due to the lack of explicit knowledge about hardware specifics and the absence of compiled binaries. In CRTES, where the interactions between hardware and software play a crucial role, the reliance on source code alone may not capture the intricacies of the underlying hardware architecture.

çOn the other side, [44] explores Measurement-Based Probabilistic Timing Analysis (MBPTA) on a real multi-core platform considering the challenges of interference from the operating system and concurrent activities. While Extreme Value Theory (EVT) is employed, the study highlights that EVT may not consistently yield adequate models and coherent probabilistic Worst-Case Execution Time (pWCET) estimations. This work raises awareness about potential risks when applying MBPTA-EVT in real-world scenarios, emphasizing the need for cautious implementation and verification of probabilistic WCET in safety-critical environments. In contrast, [45] addresses uncertainties in probabilistic WCET estimation by introducing a region of acceptance model based on statistical test procedures. It provides strategies for balancing safety and tightness in WCET estimation. Despite the improvements, the article acknowledges the inherent challenges and potential errors associated with probabilistic WCET analyses, especially in safety-critical environments. Both works contribute insights into the complexities of probabilistic WCET estimation, highlighting the importance of careful consideration and validation when applied to real-world embedded systems. However, both of them present challenges for CRTES similar to the machine learning approaches addressed before as they acknowledge errors even when all applicability hypotheses are met. In this way, while they are suitable for certain contexts, especially when attempting to obtain an early stage estimation of the WCET, they fall short in CRTES, where the need for accurate, tight, and deterministic WCET estimates is crucial for ensuring system safety and meeting stringent timing constraints. Uncertainties and the trade-off between safety and precision make these methods less viable in the critical real-time domain.

2.4 Contention Analysis and Scheduling Assumptions

Contention analysis approaches can also be categorized based on the underlying scheduling model. Dynamically scheduled systems rely on some kind of real-time decision-making, while statically scheduled (also known as offline) systems do not rely on any dynamic factors. Instead, they operate based on a pre-established schedule, defined before the beginning of their execution.

Some works [21], [10], [34], [16] are based on dynamic (real-time) scheduling, ensuring that critical tasks will have sufficient availability to complete their execution. For instance, Inam et al. implemented Behnam's proposal [46], [15] for 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 uses a Fixed Priority Pre-emptive Scheduling (FPPS) policy for both node scheduling and server scheduling. In [8], real-time monitoring is used to ensure that co-runners' requests stay under a given utilization threshold determined at analysis time. However, these works differ from the scope of ours. The increase in execution time in these works

is mainly due to real-time monitoring, the overhead during tasks' preemption, and the context switching effort.

In contrast, static (offline) scheduling aims at providing a schedule before starting the execution of the system [25], [47], [33]. For example, the work in [25] assumes preemptive scheduling and avoids making a work-conserving assumption to preserve analysis results at runtime. On the other hand, [33] considers super-blocks of tasks that are executed sequentially or according to a time-triggered schedule under different paradigms: a generic access model and dedicated phases (e.g., read-execute-write). They evaluated these paradigms and provided empirical evidence that dedicated phases lead to better response times than addressing tasks as a whole.

We can also distinguish tasks' triggering nature between event-triggered and time-triggered tasks. Event-triggered tasks are tasks that are activated sporadically due to significant events [48]. It is practically unfeasible to statically predict when these tasks are going to be executed. As a result, event-triggered tasks are typically dealt with dynamic scheduling [49], [50], [34], [16], and in most (but not all) cases, event-triggered task scheduling is associated with task pre-emption.

Conversely, time-triggered tasks are those whose activation occurs at predefined intervals in time [51]. In this work, we address the case where this occurs cyclically.

2.5 Enforcing Resource Usage Quotas

Regulating how hardware resources are allocated to cores has been explored as a method to prevent or minimize conflict. Some studies depend on specific assumptions about application semantics and hardware mechanisms, or they exploit RTOS-level mechanisms to enforce predetermined resource usage quotas.

A memory bandwidth management system to enforce memory usage quotas has been proposed in [52]. The authors differentiate memory bandwidth into two components: guaranteed and best-effort. The premise of this work is assigning priority to the task phases reclaiming guaranteed bandwidth, causing these tasks to run in temporal isolation. Meanwhile, only best-effort activities are run together under contention.

To determine the response time in multicore architectures, the authors of [50] suggest an extension of the method described by Tindell et al. [53] which depends on the use of busy windows or intervals, with emphasis on the bus through which different components communicate and access memory. Unlike our case, these works derive analyses for fixed priority preemptive systems.

When hardware isolation is not an option, contention can be controlled at the software level by leveraging specific RTOS support (such as PikeOS [54]) to enforce analysis-time utilization bounds. In line with this, the authors of [55] propose a software thread-based mechanism to access shared resources while minimizing contention. However, managing resources quotas via software inevitably adds overhead, and it is also not always possible to rely on RTOS support.

In contrast, our approach does not enforce any timing intervals in which tasks must access resources. This provides more flexibility in terms of task scheduling and resource management.

2.6 Using Performance Monitoring Counters to Model Contention

In [32], the authors distinctly outline the three primary factors needed to ascertain the effects of tasks accessing a shared resource, which are:

(i) The measurement of the volume of shared resource operations issued by a task and all tasks on a core; (ii) The examination of the total latency experienced by a set of such operations on the shared resource; and (iii) The integration of this delay into the response time analysis of each task.

We postulate that this information can be inferred (either directly or indirectly) from the Performance Monitoring Counters (PMCs) available on Commercial Off-The-Shelf (COTS) platforms. Earlier works [34], [56], [57] have employed PMC-based profiling. Dasari et al. [34] utilized PMCs to capture accesses generated on each core in an event-triggered system. For this purpose, no caches are shared (as otherwise the number of requests could fluctuate depending on interference).

Although they do not perform access pairing (which will be elaborated on in Chapter 4), their proposed solution bears some similarities with the iterative approach presented here. In contrast,

in [16] the same author dismisses an iteration-based approach computation in a cyclic schedule involving computation of the maximum interference generated and suffered, appraising it as unsafe.

This conclusion is substantiated by the fact that they rely on the concept of intervals to define the worst-case interference caused by a core. However, our approach maintains the analysis assumptions by capitalizing on time-triggered activation and inter-core synchronization. An example that provides more insights and validates the correctness of our approach is presented in Chapter 4.

In our earlier work [58], we introduced an innovative contention modeling technique tailored for a system model consistent with the assumptions presented in this study. The central objective of that research was to derive precise bounds at the hypercycle level, offering an intricate understanding of the system-wide contention dynamics. However, the current methodology delves into a more granular level of analysis, focusing on determining the utmost upper-bound for each individual task. This is achieved by meticulously monitoring and identifying all potential concurrently executing tasks that could coincide with a specific task under scrutiny. It is important to highlight that such a detailed examination, while providing valuable insights, naturally induces a conservative outlook at the hypercycle level. Nevertheless, this approach is not just an academic exercise but holds practical relevance, especially in settings where there's a pressing need for rigorous task-level assurances. When juxtaposed, the methodology from [58] and the one presented here serve as complementary tools, each addressing unique aspects of contention modeling in statically scheduled multicore systems. In Sections 3 and 4, we delve into the intricacies of our methodology. We elucidate the mechanisms by which we identify and account for all potential co-runners of a task under analysis. Furthermore, we provide a comprehensive breakdown of our strategy to accurately compute the worst-case execution time for each task. This systematic approach offers a deeper understanding of the task's runtime behavior in the presence of concurrent tasks on multicore platforms.

3 Model and Assumptions

The primary step in determining the boundaries for the maximum potential delay caused by contention in cyclic-executive and time-triggered multicore platforms is to establish the software model of the system and elucidate the inherent characteristics of the underlying hardware platform. This challenge of accurately and tightly defining upper bounds for contention in multicore systems is a recognized issue in the field of real-time embedded systems. The extent of contention a task might endure is intrinsically tied to the specific hardware and software attributes of the system. Using a broad formulation could lead to an undesirable level of conservatism. As a result, the impacts of contention have been explored under a range of distinct system and hardware premises. In this section, we delineate and expand upon the primary assumptions we have adopted in shaping our methodology.

3.1 Hardware Model and Observability

Contention bounds for each shared hardware component and operation can be specifically defined. However, the cumulative interference experienced by a task due to contention is the combined effect of contention across all shared hardware resources accessed by that task. Some COTS platforms reveal numerous contention sources, such as the Infineon AURIX Tricore TC27x and TC29x processor families. In these systems, contention can occur when accessing flash memories, static RAM, or any interface within the interconnect. Broadly speaking, the interconnect is often the primary contention source within the system. If the interconnect can handle and service multiple simultaneous requests (e.g., the SRI cross-bar in the AURIX processors), contention is shifted from the interconnect itself to the request destinations sent over it.

In this study, our attention is centered on systems where all off-chip requests are routed through a shared bus, capable of processing only one request at a time, as dictated by a specific arbitration protocol. We assume a round-robin arbitration protocol for the bus. However, concentrating on such systems does not confine the relevance of our methodology. They can be effortlessly extended to represent various system arrangements.

At this stage, our consideration is on a hardware platform where cores possess private L1 instruction and data caches. The shared bus interconnect (where contention originates) is utilized by all cores to access either the main memory or a shared L2 unified cache. Additionally, we presuppose that the L2 cache is partitioned among cores, preemptively excluding any indirect inter-core interference from the system model.

In our model, it is important to note the absence of an L3 cache. While an L3 cache is a common feature in numerous COTS, certain specialized processors and platforms cater to critical real-time applications. These specialized systems often rely solely on L1 and L2 cache configurations to ensure consistent and predictable performance. These platforms are typically tailored for specific industries such as aerospace, defense, automotive, and industrial automation. Their design principles prioritize predictable system behavior and low-latency responsiveness over sheer processing power. Consequently, these platforms may opt for processor architectures that minimize or eliminate the inclusion of an L3 cache, as the presence of an L3 cache can introduce variability in memory access times and enhance unpredictability.

Figure 1 provides an illustrative view of our reference platform. The data cache follows a write-through no write allocate policy, whereas the L2 cache uses a write-back approach. Such cache strategies are frequently found in COTS platforms within the embedded domains, like the LEON/NGMP and ARM Cortex A series processors.

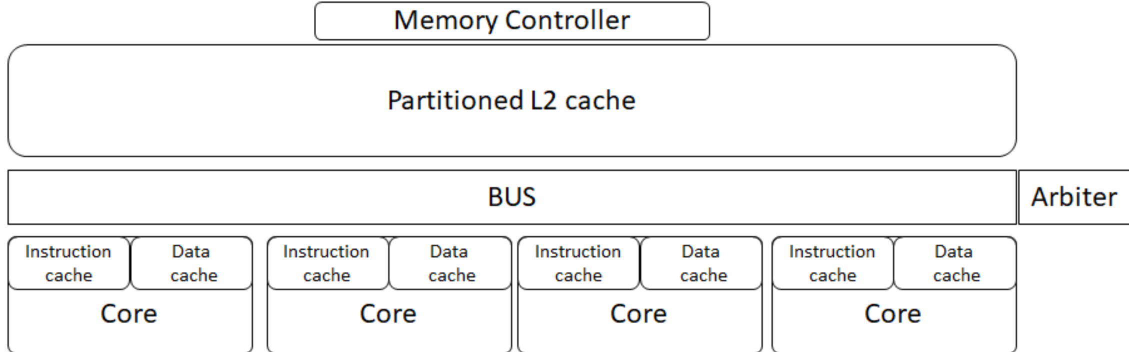


Fig. 1: Reference processor architecture

Our technique also leverages the common characteristic that the time needed to process a request can vary based on the type of request. In the reference platform we are considering, all requests pass through the bus, but the maximum latency incurred by each request is dictated by the specific access type involved. The bus acts as the fundamental source of contention since a request that gains access to the bus will occupy it until fully processed. Various types of access can utilize the bus to reach the L2 cache and potentially the main memory. These include L2 store hits (*sh*) and load hits (*lh*), along with L2 clean and dirty misses activated by load (*lmc*, *lmd*) and store (*smc*, *smd*) operations. These types of access are precisely the ones that are monitored by the performance monitoring counters.

3.2 Observability Assumption

Pertaining to the number and types of requests instigated by a task, our reference hardware platform is further described by presuming that the hardware layer provides an adequate degree of observability.

The methodology developed in this work hinge on an understanding of a task's request pattern. Specifically, we assume that the number and kinds of accesses can be determined either through static analysis [9] or dynamically [59, 60].

Each access type must then be correlated with its worst-case latency, contingent on the specific target platform. For practical applicability, we opt not to depend on any particular analytical

method (e.g., data flow analysis) to deduce per-task requests. Instead, we assume that this information is accessible via a collection of dedicated counters, commonly referred to as Performance Monitoring Counters (PMC), often integrated into the Performance Monitoring Units (PMU) of contemporary COTS platforms. Provided sufficient PMC support, tasks can be effortlessly profiled dynamically (that is, at run time) concerning access types and numbers.

The maximum latencies per access can only be dynamically derived [9], to either validate or refute the information stipulated in the platform's technical specifications.

3.3 System Model

The study encompasses a collection of m discrete periodic tasks subject to constrained deadlines, symbolized as $\mathcal{T} = \tau_1, \tau_2, \ldots, \tau_m$. Each individual task τ_i can be characterized by a quartet (c_i, r_i, P_i, D_i), where c_i and r_i are respectively indicative of the worst-case execution time and the earliest release time of τ_i, while P_i and D_i outline the task's repeating period and its relative deadline. In our model and assumptions, we exclusively focus on the consideration of worst-case execution times (c_i) for individual tasks (t_i). This deliberate simplification allows us to narrow the scope of our study, emphasizing the impact of task execution durations without delving into the complexities associated with additional parameters, thereby facilitating a more focused analysis of contention effects in the context of the given discrete periodic tasks. Therefore, the primary emphasis of this work lies in the exploration of contention effects, excluding the quest for an optimal scheduling or core task assignment [47]. As such, a static assignment of tasks to cores is presumed, and migration of tasks across cores is not considered. A uniform set of cores, denoted as Π, is contemplated, and for each core within the system, $\mathcal{T}_s$ is employed to single out the specific subset of $\mathcal{T}$ allocated to a particular core π_s, defined as $\mathcal{T}_s = \tau_i \in \mathcal{T} \wedge \tau_i \mapsto \pi_s$. In the context of the statement, the arrow symbol $\mapsto$ is used to represent a mapping or allocation from an element to a specific core. The expression $\tau_i \mapsto \pi_s$ can be read as τ_i is mapped or allocated to core π_s.

Additionally, the scheduling of tasks on each core is orchestrated non-preemptively, abiding by an implicit sequence that mirrors the underlying precedence restrictions between tasks. The scheduling mechanism assumed here is not work-conserving, meaning that a job will remain idle until the conclusion of the reserved time frame (or timing budget) corresponding to the previous task's job, even if completion has already been achieved. Lastly, it is taken for granted that inter-core dependencies amongst tasks are either nonexistent or do not influence the sequential execution order of tasks.

3.4 Types of Accesses

In the context of a homogeneous multicore processor system, the nature and targets of accesses remain uniform throughout the structure. The specifics of an off-chip access target might carry significance if multiple interconnects are present or if the interconnects offer some level of parallel processing capabilities. Nevertheless, to ease the notation's complexity, targets of operations can be integrated within the operation type itself (for instance, the operation type could be categorized as reading or writing to a particular target). Specific to our reference architecture, there is no necessity to distinguish the target of a request since all requests are routed through a singular shared bus. Consequently, target details have been excluded from the model's formulation. The different types of accesses, symbolized as $t^1, \cdots, t^k \ni \Gamma$, are processed via the interconnect, and each request $t \in \Gamma$ is characterized by a distinct latency. An upper boundary for this latency, denoted l^t, is ascertained (for instance, through comprehensive testing). Moreover, $l^{\max}$ is the maximum time it takes for a single access of any type in the system to complete. Therefore, it represents the worst-case scenario for latency, indicating the upper limit on the time a system may take to respond to a specific type of access. The notation a_i^t is utilized to pinpoint the set of access types t executed by task τ_i. Accordingly, a_i represents the total set of accesses, irrespective of their type, carried out by that task.

The particular kinds of accesses that may be taken into account, as well as their corresponding latencies, are dependent on the platform. Details concerning various access types and latencies

will be acquired and evaluated on a specified platform, as will be demonstrated in Section 5. Table 1 encapsulates the notations pertinent to the task model, various access types, and latencies, adhering to the conventions to be followed throughout this paper.

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 1: Task and core mapping notation used throughout the paper

3.5 Cyclic Executive Assumption

Real-time systems can be broadly categorized into event-driven or time-driven, based on whether actions are triggered in response to specific events or at designated time intervals. In this work, we align with the latter paradigm, assuming that tasks are executed non-preemptively according to a time-triggered approach [61], often referred to as cyclic executive scheduling. This method is prevalent in critical embedded real-time systems, attributable to its inherent predictability and minimal implementation requirements.

The underlying principle of cyclic executive scheduling involves cycling through a statically predetermined, recurring sequence of activities at a fixed frequency. Given that the initiation times for each job are calculated statically (offline) and that tasks are non-preemptive, there is no actual need for RTOS support for scheduling.

The entire set of tasks or activities is fully executed within a single hyperperiod, further segmented into smaller frames. Timing deadlines are imposed exclusively at the boundaries of these frames. In other words, every task must be executed and confined within an individual frame. While a frame might comprise multiple tasks, it can also include segmented portions of tasks, offering opportunities for optimal frame allocation.

Within each frame, lower-priority sporadic activities may optionally be executed during the available slack slots, which are not occupied by periodic tasks. Typically, the scheduling of non-periodic work must be preemptive to ensure the timely commencement of the subsequent frame [62].

Logically, the cumulative execution times (including any interference) of all tasks assigned to a frame must be less than or equal to the time slot allocated for that frame. To preclude the possibility of frame overruns, it is imperative that non-preemptive tasks conclude their execution within the designated frame. This can be accomplished through the application of precedence constraints (such as priority assignment), compelling task τ_{i-1} to complete prior to the initiation of task τ_i, facilitated by the orchestrated release times of the tasks.

Figure 2 delineates the essential principles of cyclic executive systems. To derive an appropriate schedule, an offline computation phase is indispensable. For the context of this paper, the offline phase encompasses the calculation of the WCD for each task, serving as a means to furnish secure frame-level timing assurances.

Task	f [Hz]	P [ms]
A	40	25
B	20	50
C	10	100

Fig. 2: Generic example of cyclic executive scheduling

It merits emphasis that the magnitude of contention, as well as the probability of simultaneous contending requests, is predominantly governed by the system architecture. In this work, we scrutinize a multicore instantiation of the ARINC 653 framework, wherein a static scheduler orchestrates the recurrent execution of a major frame (MAF)—analogous to the hyperperiod—subsequently partitioned into a series of minor frames (MIF). Within each MIF, a statically allocated ensemble of periodic tasks is executed non-preemptively atop a multicore platform.

Fig. 3: Task budgets within the MIF and MAF concepts.

Figure 3 depicts the mapping of tasks to a specific MIF, where each task is allocated a slot corresponding to the pre-computed timing budget. Achieving high utilization at the MIF level (by allocating more tasks) is desirable, but this goal must be balanced with the need for safe, reliable timing guarantees. These guarantees ensure that the MIF deadline will not be overrun under any execution conditions or critical alignment of possible events, even when considering the effects of contention.

In the example shown in Figure 3, a 50% utilization of the MIF would indicate that the timing budget, computed based on the isolated execution times of the mapped tasks, only covers 50% of the MIF. It leaves an open question as to how much of the remaining 50% can be utilized to accommodate additional tasks, or whether it will be sufficient to absorb the effects of contention. Providing a precise answer to this question is the ultimate objective of our work.

Although the underlying reasoning can be generalized to encompass statically scheduled systems at large, this work draws on the concepts of Integrated Modular Avionics (IMA) and the ARINC-653 standard [63]. This standard, established by the Aeronautical Radio, Inc. (ARINC)

organization, specifies the criteria for creating avionics software applications that are required to meet real-time and safety-critical standards, ensuring proper functionality even in the event of failures or abnormal conditions. In such systems, tasks or functions are executed within predetermined scheduling slots, each of a specific duration. Within the context of this work, MIFs are considered the smallest time intervals for which timing guarantees must be ensured. Thus, the static schedule is characterized by a fixed allocation of tasks to MIFs, which are then executed sequentially within the MAF.

In this multicore setting, each core's computational resources are allocated following the same static pattern (as defined by the MIFs and MAF), leading to synchronization between cores at the MIF boundaries. This kind of core synchronization at MIF boundaries is exemplified in the IMA multicore implementation supported by the SYSGO PikeOS Real-Time Operating System (RTOS) [64], as well as in the Multicore Multiple Independent Levels of Security/Safety (MILS) paradigm endorsed by VxWorks RTOS [65, 66]. Even though functions on different cores may share the same scheduling slots, they do not participate in any synchronization or communication protocol, maintaining time-composability, as per the principles laid out in [67].

3.6 Contention and Pairing of Accesses

Contention occurs when multiple requests are directed towards the same shared resource. In our approach, contention is modeled using the concept of *paired* accesses between tasks running on separate cores. In the reference platform under consideration, the bus serves as the primary source of contention, with pending requests being handled according to a well-defined policy (in this case, a round robin strategy). Owing to the implementation of the round robin arbitration policy, a given access may be paired with at most $|\Pi - 1|$ other accesses (where Π symbolizes the set of cores in the system). This pairing can occur with accesses from different cores, irrespective of the nature of these accesses, which will ultimately determine the ensuing delay. The collective number of accesses a_i for a task τ_i is the cumulative total over all specific access types associated with that task, including L2 dirty and clean misses, as well as store and load hits. To incorporate the impact of contention, the budget allocated to a task is expanded by adding the (worst-case) latency corresponding to each paired access, as dictated by the access type of the competing requests. Therefore, it becomes necessary to devise a model that accounts for pairings while also differentiating among the various access types: for each task τ_i, the number of dispatched accesses a_i^t for every access type t must be considered. It is crucial to consistently reference the worst-case access counts, which must be valid across all possible paths in the program, as justified in [59].
A task might experience an elongation in its execution time as a result of contention for a specific shared resource. Two scenarios may lead to a delay in a task attempting to access a resource: either two or more tasks try to utilize the shared resource at the exact same moment, or a task attempts to do so while the resource is already engaged by another task. While in the first scenario the incurred contention will be l^t, in the second scenario it may only constitute a fraction of l^t. Nevertheless, given the virtual impossibility of precisely forecasting the moments at which tasks will initiate their accesses, a conservative approach necessitates assuming a latency of l^t whenever an access is *paired*. Finally, we also assume that individual tasks must wait for their starting time, even if the preceding task finishes earlier than expected. This requirement is typically implemented for the purpose of maintaining a consistent and predictable scheduling scheme. This constraint ensures that each task adheres to a predefined temporal order, which is crucial for achieving reliable and deterministic system behavior. By enforcing this waiting period, the system can avoid unexpected deviations from the predetermined schedule, which is particularly important in scenarios where tasks are interacting or sharing resources.

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

4 Motivation, Design and Solution

In this section, we first motivate the intricate relationship between task overlap and potential delays due to contention as successfully managing this interaction is key to obtaining precise WCD bounds. Later on, we 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 then present an iterative approach for computing the WCD at task level.

4.1 Motivation

To derive tight WCD bounds, we combine task-level information on resource accesses with system-level details on tasks that execute concurrently on different cores. A circular dependence arises from the interplay between task overlapping and the worst-case execution time inflation caused by contention effects. Effectively capturing this circular dependence is crucial for obtaining accurate WCD bounds. Figures 4 and 5 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 4 and 5, 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 4). 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 5). On the other hand, still in Figure 5, 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.

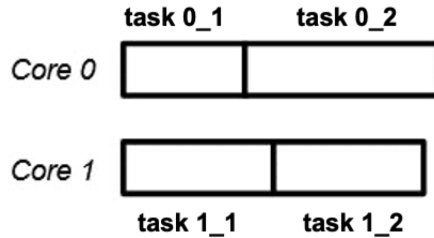


Fig. 4: Circular dependence: without contention

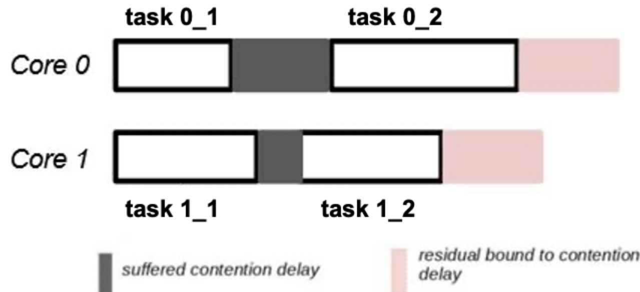


Fig. 5: Circular dependence: with contention

From a system-level perspective, varying contention effects complicate the static analysis of WCD. It becomes challenging to find overlapping within all feasible scenarios, causing 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. Besides that, not every single alignment is necessarily a real possibility, leading to excessive conservatism.

The approach involves deriving contention information from isolated resource access profiles of tasks and then using it to build an analytical model of 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 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, an iterative approach has been derived in order to capture access-type-aware task-level WCD bounds. 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. The 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, for which alternatives exist such as [58].

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 [16, 31, 56]. 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 [11, 56, 59]. 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.

Our methodology adapts to various scenarios, computing tight WCD bounds without major modifications. It outperforms state-of-the-art approaches in overall utilization [16, 31], enabling the deployment of more functionalities while maintaining strong performance guarantees. The approach's flexibility is evident, easily adapting to diverse assumptions and scenarios, including access distribution. As a proof-of-concept, our approach is validated for industrial applicability by evaluating its performance on real hardware with real task sets. Results demonstrate realistic applicability and the provision of safe tight bounds.

4.2 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 6(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).

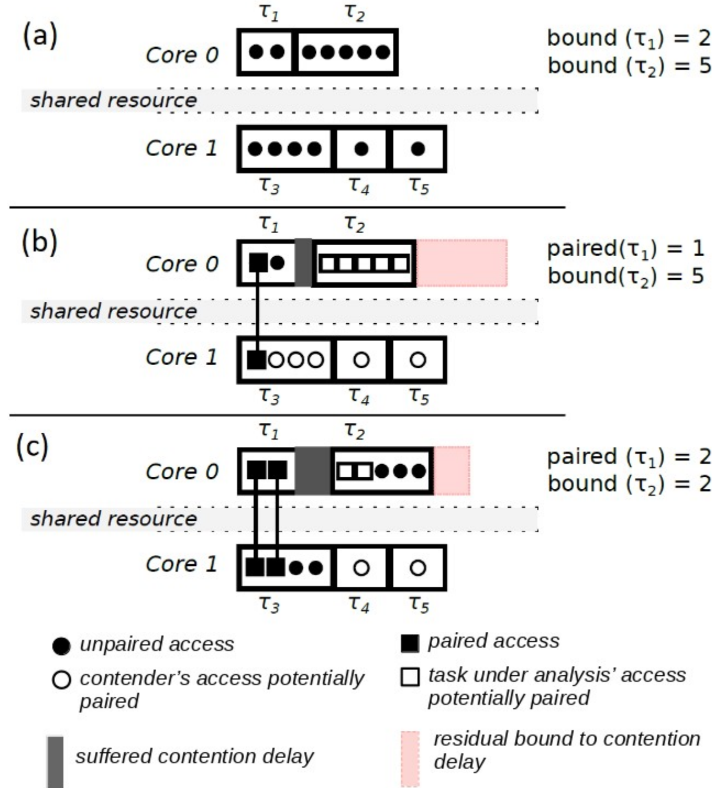


Fig. 6: Access pairing: motivation

Access pairing depends on overlapping tasks, influenced by the increased execution time due to 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 6 illustrates this effect without considering access types, for 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 6(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 6(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. 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 6, 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 section.

4.3 Modeling Pairing of Contenders Accesses

The computed WCD tightness relies on the concept of access pairing, aligning task overlapping and colliding tasks' bus accesses to derive contention. For WCD computation, notation is used to identify various 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 1, in Section 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 $\mathcal{T}_s$. The WCD analysis is performed per contender core (i.e., $\mathcal{T}_s : \pi_s \nleftarrow \tau_i$) 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 2 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 in $\mathcal{T}_s$ mapped to core $\pi_s \nleftarrow \tau_i$, 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 2: 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 $\tau_i \in \mathcal{T}_s \Rightarrow \bowtie_i(\mathcal{T}_s) = \emptyset$.

Next we identify the constraints on access pairings that apply to our approach. In this way, 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:

$$a_{j \triangleright i} \leq \min(a_i, a_j) \tag{1}$$

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:

$$\sum_{t \in \Gamma} a^t_{j \triangleright i} \leq \min(a_i, a_j) \tag{2}$$

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, given that $a^t_{j \triangleright i}$ cause contention to τ_i regardless of the concrete event that caused τ_i to access the bus, without the need of being of the same type a^t_i:

$$a^t_{j \triangleright i} \leq \min(a_i, a^t_j) \tag{3}$$

Bearing in mind the constraint defined by Equation 3, let's present the approach in more detail.

4.4 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.

Algorithm 1 formally presents the ITERative approach. The types of bus requests and the associated latencies are read from a configuration file. This makes the approach modular and adaptable to other architectures and events. Initially, the tasks in each core are characterized, and afterwards the initial tasks alignment is identified. Without any contention applied, τ_i's budget is c_i. Then, the iterative process starts by checking the co-running tasks' alignment and the pairing of accesses among tasks is performed. In the first iteration, r_i happens right after c_{i-1} for each task. For the given alignment, the contention caused is calculated iteratively for each task in each core, by incrementally matching each task under analysis' access to the most contending one available, as per the pairing principle. When this process has been done for all tasks, the new budgets and release times are updated, potentially leading to a new alignment scenario. This process is repeated until a fixed point is reached. The fixed point is reached when further iterations would not lead to new pairings and budgets inflation.

Algorithm 1 ITERative Approach for WCET Computation

```
1: procedure CalculateWCET(cores, tasks, events, latencies)
2:     Read task execution profiles, bus access patterns and their latencies
3:     for each core in cores do
4:         Characterize bus requests for each task
5:     end for
6:     Identify initial tasks alignment                    ▷ Initially, r_i based on c_{i-1}
7:     Initialize the iterative process
8:     repeat
9:         for each core in cores do
10:            for each task in core do
11:                Calculate contention based on task pairings
12:                Inflate task under analysis' budget
13:                Update next task's release time based on task under analysis' contention
14:            end for
15:        end for
16:        Check for a fixed point where no contention changes occur
17:        if fixed point not reached then
18:            Update task execution times with new contention data
19:            Identify new task pairings considering updated times
20:        end if
21:    until fixed point reached
22:    return WCET for each task computed
23: end procedure
```

In each iteration of the algorithm, the execution budget of every task is subject to potential inflation to account for the worst-case contention with co-running tasks. This conservative strategy

stems from the indeterminacy inherent in real-time multicore systems, where it is infeasible to precisely predict which tasks will contend and which will be contended at runtime. Therefore, by iteratively expanding the execution budgets to reflect the possibility of mutual contention across all cores, the algorithm converges to a state where no further budget increases are necessary. This point of convergence is reached when the algorithm's conservative assumptions have sufficiently accounted for all potential contention scenarios, ensuring that the calculated execution budgets are robust against any actual contention that may occur. The convergence of the algorithm is thus guaranteed, as the process is bounded by the finite number of tasks and the finite number of contention possibilities, each of which is considered in the iterative budgeting.

4.5 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.

In a multicore processor, the challenge of maintaining the timing behavior of individual tasks intensifies due to the complexities introduced by inter-core dependencies and variable timing elements. It becomes imperative, therefore, to rigorously validate whether the timing assumptions made for isolated tasks continue to hold when these tasks are integrated with others that run concurrently on different cores. One stringent criterion for ensuring such reliability is **full composability**. This concept stipulates that system requirements—such as timing bounds—must remain invariant irrespective of the specific components with which the task is assembled.

To delve into the specifics of ensuring full composability in the realm of multicore processors, the following two subchapters will discuss different contention models that can be employed: the *fTC Contention Model* and the *pTC Contention Model*. These models offer different methodologies and considerations for preserving timing behavior.

4.5.1 fTC Contention Model

The concept of **full composability** is further studied in [60], under the name of *fully Time-Composable* (fTC) bounds. fTC provides a strict way to meet the timing requirements, as it 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. 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.

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 below:

$$e_i = c_i + \Delta_i = c_i + a_i \times (|\Pi| - 1) \times l^{\max} \tag{4}$$

Therefore, e_i represents the WCET of task τ_i after considering the WCD effect, as defined by fTC. Despite being safe by construction, fTC suffers from a huge amount of pessimism.

4.5.2 pTC Contention Model

Time composability is generally considered as an all-or-nothing metric [68]: 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 [60] 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 approach, as the iterative approach presented here computes pTC bounds. 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. As demonstrated, 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.

By doing so, 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 setup (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}.

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 7 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

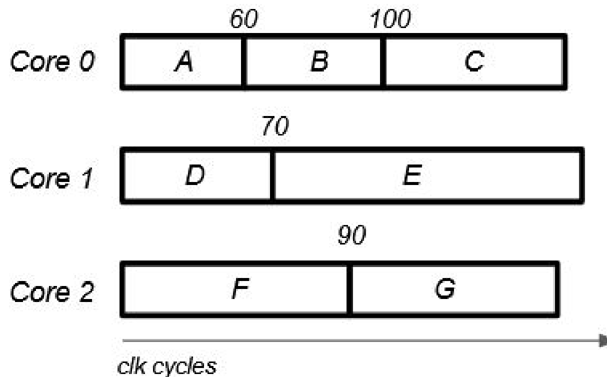


Fig. 7: Overlapping

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 3 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 3: Example input data

The WCET fTC times are calculated by applying Equation 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.

4.6 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 8 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 overlapping with the task under consideration. Among the tasks that overlap, we have to pair their accesses in the most pessimistic way (as per Equation 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 this example only one type of access is considered.

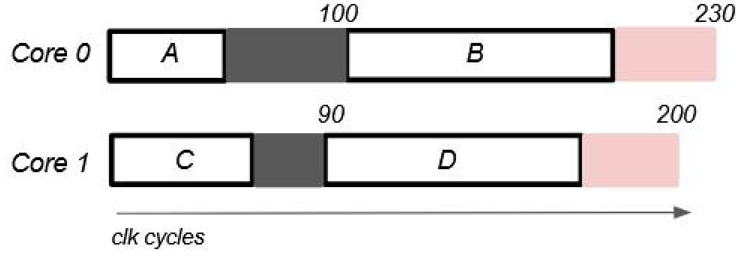


Fig. 8: pTC from fTC - Iteration 1

Referencing the presented equations in Section 4.3, 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 8, 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.

In summary:

- A → Its 4 accesses are paired with either C or D ones (as they sum 5).
- B → 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 → 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 → 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: 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.

4.7 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.

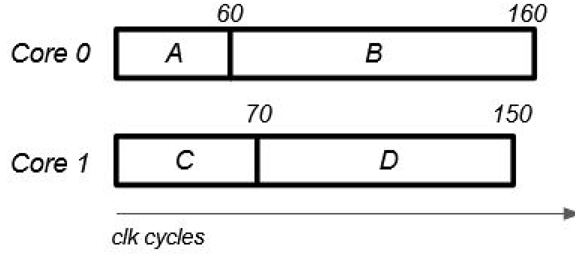


Fig. 9: pTC from isolation time - Iteration 1

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 3 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.

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 5 summarizes the new budgets and release times after one iteration. Since there has been a variation in tasks' budgets, we must proceed with another iteration, placing the tasks with their new computed budgets.

Task	pTC_1	Triggering Time_1
A	80	0
B	130	80
C	90	0
D	110	90

Table 5: pTC after iteration 1 - Isolation as starting point

Despite budgets having 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 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.

4.8 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 important

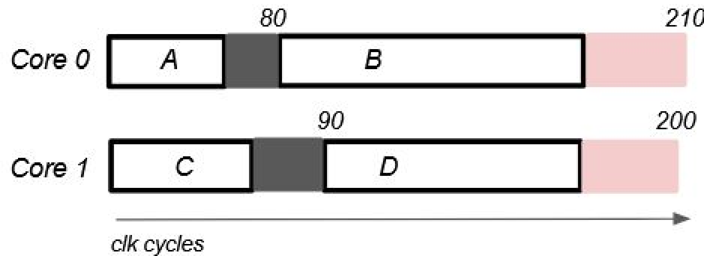


Fig. 10: pTC from isolation time - Iteration 2

Task	$\mathbf{pTC_2}$	Triggering $\mathbf{Time_2}$
A	80	0
B	130	80
C	90	0
D	110	90

Table 6: pTC after iteration 2 - Isolation as starting point

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 7 denotes.

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 7: Sample input data: safety justification

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 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.

5 Analysis of Results

In this section, the introduced methodology is appraised through a comprehensive evaluation, contrasting it with analogous and relevant state-of-the-art works, focusing particularly on the preciseness of the computed WCD (Worst-Case Delay) bounds. The evaluation demonstrates that our approach facilitates the derivation of significantly more precise makespan estimates. This improvement is achieved by adhering to two core principles: (i) resolving the previously noted interdependence between task overlapping and WCD computation; and (ii) accounting for and monitoring multiple types of memory requests. Additionally, the benefits of partitioning tasks into phases are quantified, which highlights the versatility of the proposed solution. The practical effectiveness and industrial relevance of our technique are further confirmed by a proof-of-concept assessment on an authentic hardware board.

5.1 Hardware Platform, Access Types, and Latencies

Our iterative method is adaptable with regard to the permissible access types and their corresponding maximum latencies. For the purpose of conducting our experiments, we took into account the specific access types and linked latencies demonstrated by a real hardware platform, in harmony with the hardware assumptions delineated in Section 3.

We selected the Cobham-Gaisler 4-core LEON4 platform [69] as the reference platform for our experiments, a choice that is motivated by its widespread use in the embedded space domain. Both for informing the model and executing the proof-of-concept evaluation, we focused on an FPGA implementation of LEON4. Utilizing the provided performance monitoring counters (PMCs), we extracted valuable insights from the tasks' memory access profiles, essential for the WCD computation model. The selection of this particular platform was influenced by its high relevance to the space domain, evidenced by its broad adoption in various European Space Agency endeavors.

The LEON4 system, depicted in Figure 11, is composed of 4 homogeneous cores, each operating at a frequency of 250MHz, and equipped with private L1 caches for both data and instructions. These cores are interconnected by an AMBA bus that utilizes round-robin arbitration, leading to a shared L2 cache. This cache can be tailored to support various configurations of associativity levels (1/2/4), way sizes (1-256 KB), and AHB bus widths (64 to 128 data bits). In the specific configuration under consideration, the L2 cache is partitioned, allowing each core to rely on its dedicated L2 cache partition. This arrangement serves to mitigate excessive timing interference between tasks running on different cores.

The L1 data cache operates with a write-through, no-write-allocate policy, while the L2 cache employs a write-back strategy. During bus transactions, requests block the execution until they are fully processed. This characteristic designates the bus as the principal contention source within the architecture, an essential aspect to consider in the context of the methodology presented in this work.

Various types of memory requests can traverse the bus to reach the L2 cache (and potentially the main memory). The platform under study allows for several distinct access types, as detailed in the work by Diaz et al. [60]. These include L2 store hits (*sh*) and load hits (*lh*), as well as clean and dirty misses that are triggered by both load (*lmc*, *lmd*) and store (*smc*, *smd*) operations.

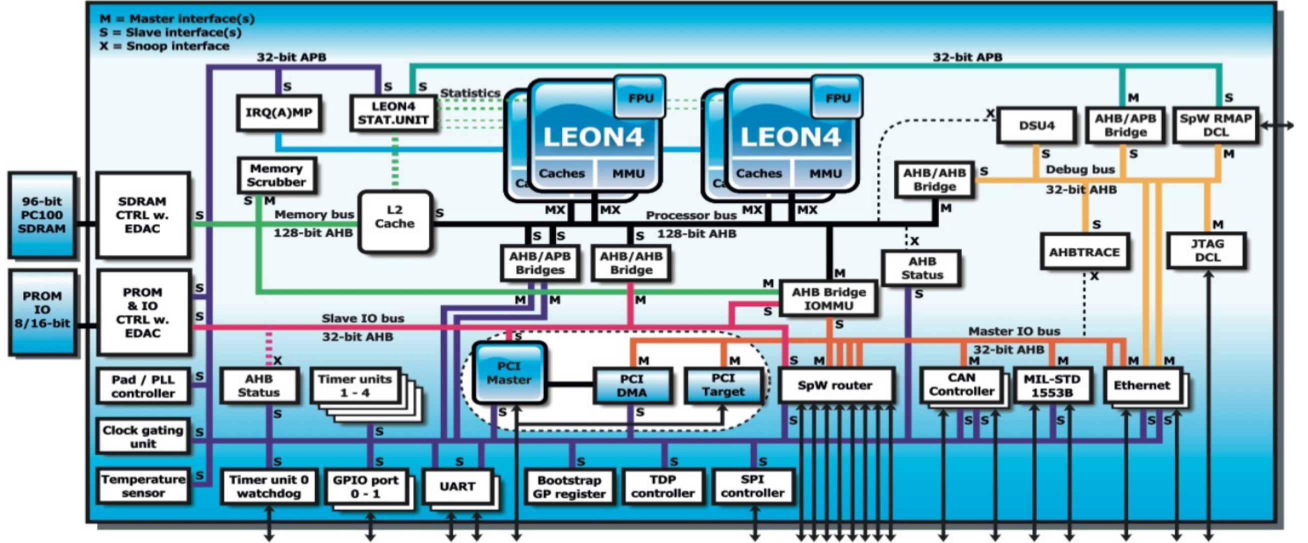


Fig. 11: LEON4 development board [70]

These access types were either directly monitored through performance monitoring counters (PMCs) or analyzed analytically, whereas the latencies for each access type were experimentally derived in the recent work by Diaz et al. [60], making use of the available PMC support on the platform. Table 8 presents a summary of the maximum latencies observed for each of these described events, providing a comprehensive insight into the memory interaction characteristics of the system.

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

Table 8: 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 9: PMCs available in the reference processor

To monitor the necessary events, four Performance Monitoring Counters (PMCs) from the LEON4 debug support unit were utilized. However, this limitation means that not all relevant events can be directly monitored. Specifically, Table 9 outlines the bus access types that can be captured with the available PMC support.

A comparison between Tables 8 and 9 reveals that not all access types can be directly tracked using the existing PMC support. Nevertheless, the counters that are missing can be analytically and conservatively derived.

Table 10 enumerates all possible events related to bus accesses. In this context, the superindexes *"l"* and *"s"* denote load and store operations respectively, while *"h"* and *"m"* correspond to hit and miss, and *"c"* and *"d"* represent clean and dirty statuses. The maximum contention delay (mcd) is expressed in clock cycles.

As previously noted, the current PMC support does not allow for tracking all of the above events. However, in this study, we demonstrate how we can analytically derive or upper-bound the untraceable events from the available PMCs, utilizing a straightforward set of formulas. This method ensures a comprehensive understanding of the system's behavior while maintaining an acceptable level of conservatism.

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^s)	n^{sh}	n^{smd}	n^{smc}

Table 10: L2 cache events

First, it is important to recognize that the events monitored by the L2 miss counter (pmc^m) can be categorized into four distinct types of misses, each depending on the operation type. These four types are further delineated based on whether the cache miss was triggered in response to a load or store operation, and whether it involved the write-back of dirty cache content (a dirty miss) or not (a clean miss).

The relationship between these four types of misses is captured by Equation 5, which breaks down the total number of L2 cache misses into these specific categories. This equation illustrates how the different types of misses contribute to the overall behavior of the cache, allowing for more nuanced analysis and understanding of the system's performance. By considering the combination of load and store operations with clean and dirty misses, a more comprehensive picture of the system's operation can be developed, enabling more precise modeling and control.

$$pmc^m = n^{lmc} + n^{smc} + n^{lmd} + n^{smd} \tag{5}$$

Given the current limitations of support, it's not possible to derive the exact number of dirty misses that occurred. Dirty misses are particularly significant since they result in a longer delay compared to clean misses. Therefore, the ability to accurately and safely upper-bound their number becomes vital for a reliable assessment of the system.

To achieve this, we can take advantage of a particular characteristic of dirty misses: a dirty miss can only occur if it has been preceded by a store operation, and furthermore, there cannot be more dirty misses than misses in total. Based on this, Equation 6 provides a secure way to upper-bound the number of dirty misses. This is done by considering the minimum between the total number of store operations (available as pmc^{st}) and the overall number of L2 misses (pmc^m).

By introducing this bound, the model is enabled to make safe predictions about the system's behavior without having to precisely measure every dirty miss. This approach manages to incorporate an understanding of the structural dependencies between different types of cache behavior into the analysis, enhancing the robustness and accuracy of the results.

$$\hat{n}^{md} = min(pmc^m, pmc^{st}) \tag{6}$$

Consequently, Equation 7 uses the bound on dirty misses to derive a lower bound on the number of clean misses (n^{mc}).

$$\check{n}^{mc} = pmc^m - \hat{n}^{md} \tag{7}$$

In the specific platform under consideration, the latency associated with a cache miss is invariant with respect to the type of operation being executed (i.e., load or store). Both dirty miss latency and clean miss latency remain constant, with values of 31 cycles and 28 cycles respectively, regardless of whether the operation is a load or store. Therefore, it is not necessary to differentiate between these two types.

However, a distinction does arise in the case of L2 load hits, which create a worst-case delay of 8 cycles, in contrast to store hits, which only result in a 1-cycle delay. Given this disparity, it becomes necessary to create a safe upper bound for the number of load hits.

By employing the same logical reasoning used to bound the number of misses, we can observe that the number of load hits cannot exceed the number of hits or loads themselves. Accordingly, Equation 8 provides an upper bound for the number of load hits by considering the minimum value between the overall number of L2 hits (n^h) and the number of loads (n^l). It's important to note that the former can be calculated by subtracting the number of L2 misses from the total number of bus accesses, and the latter can be derived by adding the L1 instruction and data cache miss counters.

This approach allows for a more refined and accurate characterization of the system's behavior, taking into consideration the specific latencies associated with different types of cache hits. By integrating these insights into the analysis, the model can better reflect the true performance constraints and operational intricacies of the multicore system.

$$\begin{aligned} n^h &= pmc^{icm} + pmc^{dcm} + pmc^{st} - pmc^m \\ n^l &= pmc^{icm} + pmc^{dcm} \\ \hat{n}^{lh} &= min(n^h, n^l) \end{aligned} \tag{8}$$

The remaining store hits are thus modeled by Equation 9 below:

$$\check{n}^{sh} = n^h - \hat{n}^{lh} \tag{9}$$

By leveraging the equations previously defined, we are able to derive the four critical events that significantly impact the latency of requests within the bus. This precise information serves as the foundation for evaluating the approaches we have presented.

5.2 Experimental Framework and Setup

Our experimental approach primarily focused on contrasting various techniques to compute the WCD for a given set of tasks under specific contention scenarios. The primary objective of this comparison was to assess how the total makespan—a critical measure of system performance—could fluctuate depending on the technique employed. Unless explicitly stated otherwise, the analysis was constrained to the context of a singular scheduling slot (Minor Interference Frame, MIF), since it is at these boundaries that timing constraints must be strictly enforced. Nonetheless, this analysis can be readily adapted to accommodate hyperperiod (Major Activation Frame, MAF) boundaries by individually applying it to each MIF.

The experimental evaluations were carried out across a comprehensive range of task sets. A fixed scheduling slot duration of 100 milliseconds per MIF was assumed, a representative value for statically scheduled systems [71], facilitating the definition of experiments that span varying overall MIF-level utilizations (calculated based on isolated task timing requirements). Importantly, this approach is not constrained to the specified slot size, as the same methodological framework can be applied to alternative configurations.

Task sets were stochastically generated employing the UUniFast algorithm [72], an established approach for the generation of task sets in real-time systems. Specifically, a total of 4,000 task sets were produced for each utilization threshold within the range of [0.1, 1], at intervals of 0.05,

culminating in an aggregate of 76,000 distinct task sets. For each utilization threshold, up to 8 tasks were generated to fit the specified requirements, showcasing the robustness and adaptability of the experimental framework. As an illustrative example, consider the goal of generating an experiment with a MIF utilization of 50%. In this scenario, the UUniFast algorithm would be employed to randomly generate a set of tasks, ensuring that the cumulative sum of their isolated execution times corresponds to the desired utilization of the MIF (in this specific case, 50ms).

Shifting the focus to inter-core interference, it becomes apparent that bus activity represents a crucial variable in the problem space. The relative tightness of the results and the comparative improvements achieved through different techniques can fluctuate based on the memory intensity of the task sets under investigation. To facilitate a balanced and unbiased evaluation, we derived specific access profiles, utilizing the memory access and cache statistics gleaned from benchmark suites in the EEMBC [73] and mediabench [74]. The access characteristics are considered in terms of *Accesses Per Kilo (thousand) Instructions* (APKI) and *Misses Per Kilo Instructions* (MPKI), providing a standardized measure for analysis.

With each generated task set, four distinct variants were crafted, each characterized by one of the access profiles itemized in Table 11. These profiles span a spectrum of behaviors, ranging from the CPU-bound profile (CPU), which exhibits relative resilience against contention, to the MEM- and BUS-bound profiles (B+M), which are more susceptible to significant contention effects.

The bus access characteristics of each task are determined in direct proportion to the quantity of instructions constituting the task. This methodology ensures a realistic and meaningful evaluation of the proposed techniques, reflecting the complex interplay between CPU instructions and memory access patterns within real-world applications.

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

Table 11: Categories created from per-access profiles

Our model is populated with data drawn from both the task specification and the defined access profiles. Experiments are conducted in an analogous manner across all access profiles. In a given utilization threshold, each experiment involves selecting 4 task sets (one per core in the LEON4 system): one task set is singled out for focused analysis, while the other three sets are presumed to be mapped to the contending cores. The original task set's makespan is juxtaposed with the one attained when considering the WCD for each task in the set. Specifically, what captures interest here is the proportion of task sets that stay within the slot boundaries even after factoring in the WCD. As the WCD bounds approach, the makespan increase becomes smaller and the probability of overruns decreases.

The examined solution is contrasted with similar approaches found in contemporary research, details of which will be provided in subsequent descriptions of individual experiments. Each experiment is meticulously designed to highlight a feature that sets the proposed method apart from existing state-of-the-art techniques.

It is imperative to underscore that the necessary inputs for executing the experiments are fairly minimal. The only inputs required are: (i) the number of cycles that tasks consume for execution when run in isolation, denoted as n^c; and (ii) the count of each monitored bus access event, retrievable through Performance Monitoring Counters (PMCs). All inputs can be extrapolated from tasks operating independently, devoid of contention effects.

The model assumes that the quantity and types of accesses can be ascertained either through static analysis [9] or by garnering data from the performance monitoring counters, akin to the method described in [59]. The latter approach, outlined in the previous section, has been employed

in our experiments. Each access type is typified by a worst-case latency. In the forthcoming experiments, the per-type latencies, fed into the model as input, are those discerned and obtained from the LEON4 board. Analogous experiments could be formulated for other access types and latency profiles.

All the experiments were conducted on a laptop system, comprising a 4x Intel i7-5600U CPU running at 2.60GHz, with 16GB of RAM memory. The approach is implemented through a Python script, which efficiently carries out the computations and produces the result within a negligible time frame.

5.3 Multiple Access Types Distinction

The first scenario we aimed to scrutinize is the advantages that might be reaped by differentiating multiple types of access in the WCD computation. The approach posited in [16] presupposes a singular request type, equivalent to assuming that all requests incur the lengthiest (worst) latency, to safely provide a WCD bound. The degree of pessimism originating from this constraint is contingent on the specific platform. In the context of our chosen case and platform, this methodology forces the assumption that all requests take 31 cycles (corresponding to dirty misses), as detailed in Table 8.

This assumption inherently introduces an over-estimation in the computed WCD, especially in scenarios where the dirty misses form only a fraction of the total access types. By recognizing and accounting for the various access types and their corresponding latencies, the analysis can be refined, leading to more accurate and less conservative results.

In these experiments, the evaluation focuses on the ratio of task sets whose timing requirements, when augmented by the WCD, surpass the scheduling slot. This comparison serves to highlight the effectiveness and precision of the iterative approach over the one-type model.

Figure 12 illustrates the success ratio achieved at various utilization thresholds and for all the specified workload types (CPU, BUS, MEM, and B+M). Each data point in the figure represents the application of the WCD analysis to 1,000 randomly generated task sets, as delineated in Section 5.2.

Within the graph, the results obtained through the iterative approach are labeled as *"Iter"*, whereas those obtained through the iterative method considering only a single request type of bus access are denoted as *"Iter-1RT"*.

Observations: The comparison evidently manifests that distinguishing between different types of accesses in the WCD computation reduces the overruns, leading to a more accurate representation of system behavior. This finding underlines the importance of considering the distinctions between access types rather than generalizing them into one.

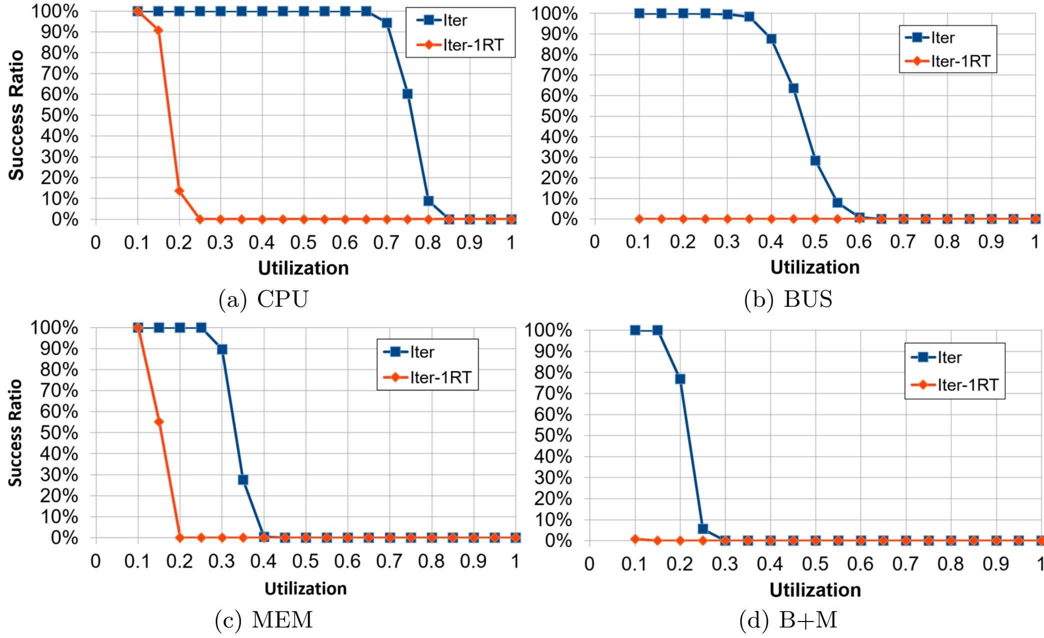
"Iter" consistently outperforms *"Iter-1RT"* across various utilization thresholds and workload types, reinforcing the premise that considering multiple access types adds more granularity and realism to the WCD analysis.

These results not only underscore the enhanced accuracy of the iterative approach but also elucidate the inherent conservatism in the one-type model. The extension to multiple types offers a valuable refinement in modeling the system's responsiveness and could potentially lead to more efficient system design and resource allocation.

Overall, this comparative analysis provides strong evidence that taking into account the specific characteristics of different access types in WCD computation has a significant impact on the accuracy of timing predictions, thereby offering a more robust basis for system design and evaluation.

Figure 12 elucidates the advantage of recognizing and tracking distinct access types. The *Iter* method consistently prevails over the *Iter-1RT* method across all access profiles. Particularly, in the context of memory-intensive access profiles, the failure to differentiate between request types leads to an inability to allocate tasks even for a meager 10% of MIF utilization. This emphasizes the necessity of considering bus access types to evade undue pessimism in WCD computation.

It's noteworthy to highlight that the utilization threshold pertains to the ratio between the slot size and tasks' WCET in isolation, meaning it does not incorporate the influence of multicore



(a) CPU (b) BUS (c) MEM (d) B+M

Fig. 12: Iterative vs Iterative-1RT

contention. Consequently, as the pressure on shared resources escalates – predominantly occurring in MEM and B+M workloads, as dirty misses (and subsequent memory access) are the events incurring the highest latency – the resulting multicore CPU utilization can exceed 100% at the core level. This, in turn, renders the task set unschedulable, showcasing the importance of accurately representing different access types in the analysis.

In summary, these findings reinforce the crucial role of accurate modeling of access types in WCD computation. Not only does this enhance the precision of the system's analysis, but it also helps in averting overly conservative estimations that could limit the system's operational flexibility and efficiency. This opens avenues for more robust and refined system design and resource allocation, particularly in environments with memory-intensive workloads.

5.4 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 within tasks. Several studies [10, 21, 22, 38, 39] 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.

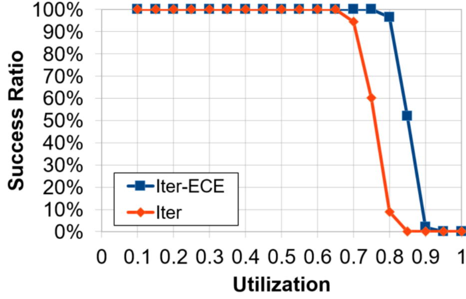


Fig. 13: Iterative vs Iterative-ECE CPU profile

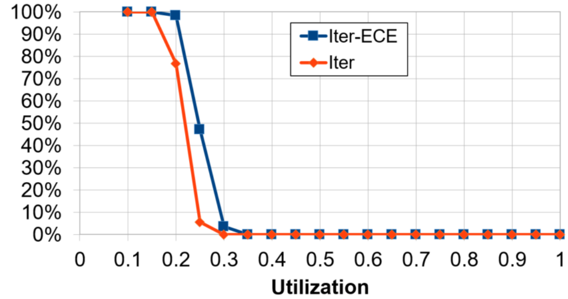


Fig. 14: Iterative vs Iterative-ECE B+M profile

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 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.

5.5 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 method is 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, the hereby proposed model ends 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:

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., $c_i \approx 50$ms) each. 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. In accordance with cyclic executive scheduling, each task's release time r_i is scheduled to occur immediately at the end of the period of the preceding task, denoted as P_{i-1}, ensuring tasks do not start until their designated time slots even if earlier tasks complete ahead of schedule.
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 the model, in the form of a CSV.

4. By having this information, the Python script was executed to calculate the WCD bounds. Additionally, the fTC budgets were computed as a reference. As said, 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.

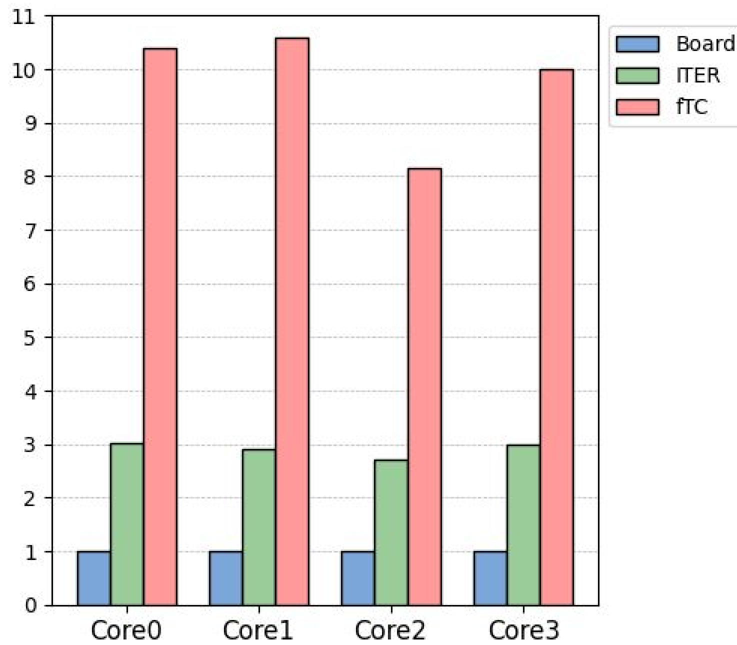


Fig. 15: Summary of the results

In Figure 15, 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. The ITER approach 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.

Evaluating our solution, we can observe that it provides an overall budget of slightly less than 3 times the observed one in the board.

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 our approach is. The benefits of ITER with regard to task-level timing guarantees have already been observed. 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 3x 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.

6 Conclusions

In this final section, we consolidate the work that has been conducted as part of this paper. We outline the ways in which the proposed approach enhances the current state of the art concerning the computation of the worst-case contention delay. Furthermore, we present some promising avenues for further refining the outcomes obtained.

One significant achievement of our work is a method to calculate worst-case contention delay (WCD) bounds with task-level guarantees. We demonstrated how an iterative approach enables the detection of safe and reasonably tight time budgets for each individual task, with the tightness approximately three times the observed execution time. The contributions of this paper can be summarized as follows:

A comprehensive review of the state-of-the-art of real-time multicore timing analysis, particularly concerning inter-core contention modeling. An iterative method to compute WCD bounds with task-level guarantees. A fully automated framework for the implementation and execution of the proposed approach. A proof-of-concept application of the proposed methodology on real hardware, which evaluates their performance and industrial applicability.

6.1 Tightening the Bounds on WCD Computation

In multicore systems, the potential contention that multiple tasks on different cores might suffer at runtime, when accessing shared hardware resources, is largely determined by how these tasks overlap in time. Conversely, the delay resulting from this contention modifies tasks' worst-case execution time (WCET) and response time, which in turn affects tasks overlapping.

Throughout this paper, we've highlighted that while multicore and multi-level cache systems can deliver significant timing and performance benefits, timing analysis models must be updated to provide safe and tight bounds on contention effects to avoid undermining these potential advantages.

This paper proposes an iterative methodology for computing a bound to the WCD endured by tasks in statically scheduled, bus-based, multicore systems. The method prioritizes WCD bounds for each task in the system, sacrificing some pessimism (i.e., conservative assumptions) in exchange for task-level guarantees instead of system-level ones. We particularly focus on capturing the circularity of contention effects on task overlapping.

The iterative methodology has proven effective in finding a safe task schedule by calculating appropriate tasks' release times. Our evaluation indicates that the iterative approach outperforms previous work in terms of WCD tightness. This advantage stems from the iterative approach's ability to exploit access information from both the task under analysis and potential contenders, and to differentiate between various access types.

Additionally, we showcased the flexibility of our evaluation. We demonstrated how it could be successfully adapted to model different WCD assumptions and scenarios, such as approaches that distribute shared resource accesses across different task phases. We demonstrated how easily our model can be adapted to account for any additional information that might be available on the system and task set.

We also implemented a highly automated experimental framework that significantly reduces the time needed to conduct the experiments and analyze the results. The framework is platform-independent, easily configurable, and does not rely on any real-time operating system (RTOS) support, requiring only minimal information about applications running in isolation as inputs.

Finally, we demonstrated the performance, safety, and industrial applicability of the presented approach on a real hardware platform, detailing all the necessary steps for its proper application from scratch.

6.2 Possible Extensions and Future Directions

The results and insights obtained from this study can be enhanced and expanded in multiple ways. Here, we discuss a few promising directions for future research:

Using actual overlapping time to limit the number of conflicts: The current work could be further refined by quantifying task alignment. Presently, a pair of tasks are modeled to either overlap completely or not at all. Depending on this, their accesses are permitted to interfere with each other or not. A potential enhancement could involve quantifying the extent of overlapping, taking into account not just whether tasks overlap, but also for how long. This could limit the number and type of possible conflicting accesses to the bus. For instance, if two tasks overlap for only 100 cycles even in the most conservative scenario, we can restrict our model to not pair ten accesses each incurring a delay of 28 cycles. This information could be encoded in the logic of the iterative approach. The safety of the approach would still be guaranteed, as we would only be eliminating unrealistic scenarios.

Events distribution and ordering: Another potential enhancement could involve obtaining more detailed information on the order in which tasks' requests are triggered. This added knowledge could help refine the potential collision scenarios among tasks by keeping track of the already paired and yet-to-be-paired types of access. For example, if we knew that the first L2 cache miss does not occur until after ninety load hits, we could prevent this miss from being paired against other accesses before having paired these ninety load hits. This order could be incorporated into the models, adding precedence constraints.

A similar approach could be taken if we had more information about access distribution. To do this, tasks run in isolation would need to be analyzed in segments rather than as a whole. We could divide a task into several parts and use the performance monitoring counters (PMC) values at the level of these parts to inform our model. This is similar to what we did in our experiments when we divided tasks into phases, which as we explained, can potentially rule out certain cases leading to worse contention delays.

Improved classification of accesses: As discussed in Section 5.1, not all events can be directly monitored, and a conservative upper bound had to be used for certain types of access. If more PMCs were available, we would be able to accurately determine the number of all relevant events (types of access). This could significantly reduce the computed WCD in most cases, as the number of events causing the highest latencies are likely to decrease.

Ethical Approval
Not applicable.

Funding
This work has been partially funded by the EU Horizon program under grant agreements No.825184 and No.101092644, by the Spanish Government through project PID2019-106774RB-C22, by the Government of Catalonia as Consolidated Research Group 2021-SGR-109, and by the Ministry of Economic Affairs and Digital Transformation, in conjunction with the European Union-NextGenerationEU (within the framework of the PRTR and the MRR), through the CLOUDLESS UNICO I+D CLOUD 2022.

Availability of data and materials
Not applicable.

Bibliography

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

[2] RTCA and EUROCAE: DO-178C / ED-12C, Software Considerations in Airborne Systems and Equipment Certification. (2011)

[3] International Organization for Standardization: ISO/DIS 26262. Road Vehicles – Functional Safety. (2009)

[4] 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)

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

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

[7] 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

[8] Nowotsch, J., Paulitsch, M., Bühler, D., Theiling, H., Wegener, S., Schmidt, M.: Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In: 26th Euromicro Conference on Real-Time Systems, pp. 109–118 (2014). https://doi.org/10.1109/ECRTS.2014.20

[9] 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 (2008)

[10] Pellizzoni, R., Bui, B.D., Caccamo, M., Sha, L.: Coscheduling of CPU and I/O Transactions in COTS-Based Embedded Systems. In: Real-Time Systems Symposium, pp. 221–231 (2008). https://doi.org/10.1109/RTSS.2008.42

[11] Jalle, J., Fernandez, M., Abella, J., Andersson, J., Patte, M., Fossati, L., Zulianello, M., Cazorla, F.J.: Contention-aware performance monitoring counter support for real-time mpsocs. In: 11th IEEE Symposium on Industrial Embedded Systems (SIES), pp. 1–10 (2016). https://doi.org/10.1109/SIES.2016.7509440

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

[13] Valsan, P.K., Yun, H., Farshchi, F.: 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, pp. 1–12

[14] Blagodurov, S., Zhuravlev, S., Fedorova, A.: Contention-aware scheduling on multicore systems. ACM Trans. Comput. Syst. **28**(4), 8–1845 (2010) https://doi.org/10.1145/1880018.1880019

[15] Behnam, M., Inam, R., Nolte, T., Sjödin, M.: Multi-core composability in the face of memory-bus contention. ACM SIGBED Review **10**(3), 35–42 (2013)

[16] Dasari, D., Nelis, V.: 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, pp. 1450–1457 (2012). https://doi.org/10.1109/HPCC.2012.212 . http://dx.doi.org/10.1109/HPCC.2012.212

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

[18] Burns, A., Wellings, A.J.: A Schedulability Compatible Multiprocessor Resource Sharing Protocol – MrsP. In: 25th Euromicro Conference on Real-Time Systems, pp. 282–291 (2013)

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

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

[21] Pellizzoni, R., Betti, E., Bak, S., Yao, G., Criswell, J., Caccamo, M., Kegley, R.: A predictable execution model for cots-based embedded systems. In: 17th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 269–279 (2011). https://doi.org/10.1109/RTAS.2011.33

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

[23] Kim, H., De Niz, D., Andersson, B., Klein, M., Mutlu, O., Rajkumar, R.: Bounding and reducing memory interference in cots-based multi-core systems. Real-Time Syst. **52**(3), 356–395 (2016) https://doi.org/10.1007/s11241-016-9248-1

[24] Melani, A., Mancuso, R., Caccamo, M., Buttazzo, G., Freitag, J., Uhrig, S.: 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), pp. 1–10 (2017). https://doi.org/10.1109/RTCSA.2017.8046314

[25] Martinez, S., Hardy, D., Puaut, I.: 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 (2017). https://doi.org/10.475/123_4

[26] Kim, H., Niz, D., Andersson, B., Klein, M., Mutlu, O., Rajkumar, R.: Bounding memory interference delay in cots-based multi-core systems. In: 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 145–154. https://doi.org/10.1109/RTAS.2014.6925998

[27] Serrano-Cases, A., Reina, J.M., Abella, J., Mezzetti, E., Cazorla, F.J.: Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC. In: Brandenburg, B.B. (ed.) 33rd Euromicro Conference on Real-Time Systems (ECRTS 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 196, pp. 3–1326. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ECRTS.2021.3 . https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2021.3

[28] Altmeyer, S., Davis, R.I., Indrusiak, L., Maiza, C., Nelis, V., Reineke, J.: A generic and compositional framework for multicore response time analysis. In: Proceedings of the 23rd International Conference on Real Time and Networks Systems. RTNS '15, pp. 129–138. Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2834848.2834862 . https://doi.org/10.1145/2834848.2834862

[29] Rihani, H., Moy, M., Maiza, C., Davis, R.I., Altmeyer, S.: Response time analysis of synchronous data flow programs on a many-core processor. In: 24th International Conference on Real-Time Networks and Systems. RTNS, pp. 67–76 (2016)

[30] Dinechin, M., Schuh, M., Moy, M., Maiza, C.: Scaling up the memory interference analysis for hard real-time many-core systems. In: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 330–333 (2020). https://doi.org/10.23919/DATE48585.2020.9116460

[31] Schliecker, S., Negrean, M., Ernst, R.: Bounding the shared resource load for the performance analysis of multiprocessor systems. In: Conference on Design, Automation and Test in Europe. DATE, pp. 759–764 (2010). http://dl.acm.org/citation.cfm?id=1870926.1871108

[32] Schliecker, S., Negrean, M., Ernst, R.: Bounding the shared resource load for the performance analysis of multiprocessor systems. In: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 759–764 (2010). European Design and Automation Association

[33] Schranzhofer, A., Pellizzoni, R., Chen, J.J., Thiele, L., Caccamo, M.: Worst-case response time analysis of resource access models in multi-core systems. In: Design Automation Conference, pp. 332–337 (2010). https://doi.org/10.1145/1837274.1837359

[34] Dasari, D., Andersson, B., Nelis, V., Petters, S.M., Easwaran, A., Lee, J.: Response time analysis of cots-based multicores considering the contention on the shared memory bus. In: Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference On, pp. 1068–1075. IEEE

[35] Grebant, S., Ballabriga, C., Forget, J., Lipari, G.: WCET Analysis with Procedure Arguments as Parameters. In: In Proceedings of the 31st International Conference on Real-Time Networks and Systems, pp. 11–22 (2023)

[36] Dharishini, P.P.P., Murthy, P.R.: Static Analyzer for Computing WCET of Multithreaded Programs using Hoare's CSP. In: In Proceedings of the 15th Innovations in Software Engineering Conference, pp. 1–12 (2022)

[37] Dharishini, P.P.P., Murthy, P.R.: Design and Implementation of a Simulator for Precise WCET Estimation of Multithreaded Program. In: International Journal of Electrical and Computer Engineering Systems, vol. 14 (2023)

[38] Alhammad, A., Wasly, S., Pellizzoni, R.: Memory efficient global scheduling of real-time tasks. In: 21st IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 285–296 (2015). https://doi.org/10.1109/RTAS.2015.7108452

[39] Becker, M., Dasari, D., Nicolic, B., Akesson, B., Nelis, V., Nolte, T.: Contention-free execution of automotive applications on a clustered many-core platform. In: 28th Euromicro Conference on Real-Time Systems (ECRTS), pp. 14–24 (2016). https://doi.org/10.1109/ECRTS.2016.14

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

[41] Andrade, T.N.C., Lima, G., Cadena Lima, V.M., Bem-Amor, S., Hawila, I., Cucu-Grosjean, L.: On the Impact of Hardware-Related Events on the Execution of Real-Time Programs. In: Design Automation for Embedded Systems, vol. 27, pp. 275–302 (2023)

[42] Amalou, A.N., Puaut, I., Muller, G.: WE-HML: Hybrid WCET Estimation Using Machine Learning for Architectures with Caches. In: IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 31–40 (2021)

[43] Kumar, V.: Estimation of an Early WCET Using Different Machine Learning Approaches. In: International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, pp. 297–307 (2022)

[44] Vasconcelos, J., Lima, G., El Khazen, M.W., Gogonel, A., Cucu-Grosjean, L.: On Vulnerabilities in EVT-Based Timing Analysis: An Experimental Investigation on a Multi-Core Architecture. In: Design Automation for Embedded Systems (2023)

[45] Reghenzani, F., Santinelli, L., Fornaciari, W.: Dealing with Uncertainty in pWCET Estimations. In: ACM Transactions on Embedded Computing Systems, vol. 19 (2020)

[46] Inam, R., Mahmud, N., Behnam, M., Nolte, T., Sjödin, M.: The multi-resource server for predictable execution on multi-core platforms. In: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th, pp. 1–12

[47] Rouxel, B., Derrien, S., Puaut, I.: Tightening Contention Delays While Scheduling Parallel Applications on Multi-core Architectures. ACM Transactions on Embedded Computing Systems (TECS) **16**(5s), 1–20 (2017) https://doi.org/10.1145/3126496

[48] Kopetz, H.: Event-triggered versus time-triggered real-time systems. In: Operating Systems of the 90s and Beyond, pp. 86–101. Springer, ??? (1991)

[49] Negrean, M., Schliecker, S., Ernst, R.: 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, pp. 524–529 (2009). European Design and Automation Association

[50] Schliecker, S., Negrean, M., Ernst, R.: Response time analysis on multicore ecus with shared resources. IEEE Transactions on Industrial Informatics **5**(4), 402–413 (2009)

[51] Kopetz, H.: The time-triggered model of computation. In: RTSS, p. 168 (1998). IEEE

[52] Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., Sha, L.: 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, pp. 55–64 (2013)

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

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

[55] Zhuravlev, S., Blagodurov, S., Fedorova, A.: Addressing shared resource contention in multicore processors via scheduling. In: ACM Sigplan Notices, vol. 45, pp. 129–142 (2010)

[56] Diaz, E., Mezzetti, E., Kosmidis, L., Abella, J., Cazorla, F.J.: Modelling Multicore Contention on the AURIX^{TM} TC27x. In: Design& Automation Conference (DAC) (2018)

[57] Jalle, J., Fernandez, M., Abella, J., Andersson, J., Patte, M., Fossati, L., Zulianello, M.,

Cazorla, F.J.: Bounding Resource Contention Interference in the Next-Generation Microprocessor (NGMP). In: 8th European Congress on Embedded Real Time Software and Systems (ERTS) (2016). https://hal.archives-ouvertes.fr/hal-01259133

[58] Palomo, X., Mezzetti, E., Abella, J., Bril, R.J., Cazorla, F.J.: Accurate ilp-based contention modeling on statically scheduled multicore systems. In: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 15–28 (2019). IEEE

[59] Dasari, D., Andersson, B., Nelis, V., Petters, S.M., Easwaran, A., Lee, J.: 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, pp. 1068–1075 (2011). https://doi.org/10.1109/TrustCom.2011.146

[60] Díaz, E., Fernández, M., Kosmidis, L., Mezzetti, E., Hernandez, C., Abella, J., Cazorla, F.J.: MC2: Multicore and Cache Analysis via Deterministic and Probabilistic Jitter Bounding. In: Ada-Europe International Conference on Reliable Software Technologies, pp. 102–118 (2017)

[61] Kopetz, H.: Real-Time Systems: Design Principles for Distributed Embedded Applications, 1st edn. (1997)

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

[63] ARINC: Specification 651: Design Guide for Integrated Modular Avionics. Aeronautical Radio, Inc, (1997). Aeronautical Radio, Inc

[64] SYSGO PikeOS RTOS. http://www.sysgo.com/ (2018)

[65] WIND RIVER VxWorks MILS Platform 3.0. https://www.windriver.com/ (2018)

[66] Parkinson, P.J.: Multicore mils. In: 9th IET International Conference on System Safety and Cyber Security, pp. 1–8 (2014)

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

[68] Fernandez, G., Abella, J., Quiñones, E., Vardanega, T., Fossati, L., Zulianello, M., Cazorla, F.J.: Introduction to partial time composability for COTS multicores. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, pp. 1955–1956 (2015)

[69] LEON4 Datasheet. https://www.gaisler.com/doc/LEON4-N2X-DS.pdf

[70] LEON4 Summary. http://www.sjalander.com/research/pdf/sjalander-dasia2009.pdf

[71] Locke, D., R. Vogel, D., Lucas, L., Goodenough, J.: Generic avionics software specification. Technical report, Software Engineering Institute, Carnegie Mellon University (1990)

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

[73] Poovey, J.: Characterization of the EEMBC Benchmark Suite. North Carolina State University, (2007). North Carolina State University

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