UNIVERSITÄT AUGSBURG
Universitätsbibliothek

OPEN ACCESS
OPUS AUGSBURG

An extensible framework for multicore response time analysis

Robert I. Davis, Sebastian Altmeyer, Leandro S. Indrusiak, Claire Maiza, Vincent Nelis, Jan Reineke

Angaben zur Veröffentlichung / Publication details:

Davis, Robert I., Sebastian Altmeyer, Leandro S. Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. 2018. "An extensible framework for multicore response time analysis." *Real-Time Systems* 54 (3): 607–61. https://doi.org/10.1007/s11241-017-9285-4.

Nutzungsbedingungen / Terms of use:

CC BY 4.0

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
CC-BY 4.0: Creative Commons: Namensnennung
Weitere Informationen finden Sie unter: / For more information see:
https://creativecommons.org/licenses/by/4.0/deed.de

Real-Time Syst (2018) 54:607–661
https://doi.org/10.1007/s11241-017-9285-4

An extensible framework for multicore response time analysis

Robert I. Davis[1,2] · **Sebastian Altmeyer**[3] · **Leandro S. Indrusiak**[1] · **Claire Maiza**[4] · **Vincent Nelis**[5] · **Jan Reineke**[6]

Published online: 18 July 2017
© The Author(s) 2017. This article is an open access publication

Abstract In this paper, we introduce a *multicore response time analysis* (*MRTA*) *framework*, which decouples response time analysis from a reliance on context-independent WCET values. Instead, the analysis formulates response times directly

This paper extends initial research on "*A Generic and Compositional Framework for Multicore Response Time Analysis*" presented at RTNS 2015 (Altmeyer et al. 2015).

The additional material includes: Section 4: analysis for warmed-up caches and dynamic scratchpads (Sects. 4.3 and 4.2). Section 7: extensions to the task model, including RTOS and interrupts, shared software resources, and open systems and incremental verification. Section 8: presentation of a cycle-accurate multicore simulator. Section 9: evaluation of different local memory types (Sect. 9.2), a comparison between predictable and reference architectures (Sect. 9.3), and a verification of the precision of the analysis using results from the simulator (Sect. 9.4).

✉ Sebastian Altmeyer
altmeyer@uva.nl

Robert I. Davis
rob.davis@york.ac.uk

Leandro S. Indrusiak
leandro.indrusiak@york.ac.uk

Claire Maiza
claire.maiza@imag.fr

Vincent Nelis
nelis@isep.ipp.pt

Jan Reineke
reineke@cs.uni-saarland.de

1 University of York, York, UK

2 INRIA Paris, Paris, France

3 University of Amsterdam, Amsterdam, The Netherlands

from the demands placed on different hardware resources. The MRTA framework is extensible to different multicore architectures, with a variety of arbitration policies for the common interconnects, and different types and arrangements of local memory. We instantiate the framework for single level local data and instruction memories (cache or scratchpads), for a variety of memory bus arbitration policies, including: Round-Robin, FIFO, Fixed-Priority, Processor-Priority, and TDMA, and account for DRAM refreshes. The MRTA framework provides a general approach to timing verification for multicore systems that is parametric in the hardware configuration and so can be used at the architectural design stage to compare the guaranteed levels of real-time performance that can be obtained with different hardware configurations. We use the framework in this way to evaluate the performance of multicore systems with a variety of different architectural components and policies. These results are then used to compose a predictable architecture, which is compared against a reference architecture designed for good average-case behaviour. This comparison shows that the predictable architecture has substantially better guaranteed real-time performance, with the precision of the analysis verified using cycle-accurate simulation.

Keywords Multicore scheduling · Timing analysis · Verification

1 Introduction

Effective analysis of the worst-case timing behaviour of systems built on multicore architectures is essential if these high-performance platforms are to be deployed in critical real-time embedded systems used in the automotive and aerospace industries. We identify four different approaches to solving the problem of determining timing correctness.

With single-core systems, a *traditional two-step* approach is typically used. This consists of timing analysis which determines the *context-independent* worst-case execution time (WCET) of each task, followed by schedulability analysis, which uses task WCETs and information about the processor scheduling policy to determine if each task can be guaranteed to meet its deadline. When local memory (e.g. cache) is present, then this approach can be augmented by analysis of Cache Related Preemption Delays (CRPD) (Altmeyer et al. 2012), or by partitioning the cache to avoid CRPD altogether. Both approaches are effective and result in tight upper bounds on task response times (Altmeyer et al. 2014, 2016).

With a multicore system, the situation is more complex since WCETs are strongly dependent on the amount of *cross-core interference* on shared hardware resources such as main memory, L2 caches, and common interconnects, due to tasks running on other cores. The uncertainty and variability in this cross-core interference renders the traditional two-step process ineffective for many multicore processors. For exam-

[4] Universite Grenoble-Alpes, Grenoble, France

[5] CISTER, ISEP, Porto, Portugal

[6] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

ple, on the Freescale P4080, the latency of a read operation varies from 40 to 600 cycles depending on the total number of cores running and the number of competing tasks (Nowotsch et al. 2014). Similarly, a 14 times slowdown has been reported (Radojković et al. 2012) due to interference on the L2 cache for tasks running on Intel Core 2 Quad processors. Further, recent work by Valsan et al. (2016) has shown that in some multicore systems with out-of-order execution, cache partitioning is insufficient to provide effective isolation, with shared Miss Status Handling Registers (MSHR) causing up to 20 times slowdown to code allocated its own separate cache partition.

At the other extreme is a *fully integrated* approach. This involves considering the precise interleaving of instructions originating from different cores (Gustavsson et al. 2010); however, such an approach suffers from potentially insurmountable problems of combinatorial complexity, due to the proliferation of different path combinations, as well as different release times and schedules.

An alternative approach is based on *temporal isolation* (Bui et al. 2011). The idea here is to statically partition the use of shared resources, e.g. space partitioning of cache and DRAM banks, time partitioning of bus access etc., so that context-independent WCET values can be used and the traditional two-step process applied. This approach raises a further challenge, how to partition the resources to obtain schedulability (Reineke and Doerfert 2014). Techniques which seek to limit the worst-case cross-core interference, for example by using TDMA arbitration on the memory bus or by limiting the amount of contention by suspending execution on certain cores (Nowotsch et al. 2014), can have a significant detrimental effect on performance, effectively negating the performance benefits of using a multicore system altogether. We note that TDMA is rarely if ever used as a bus-arbitration policy in real multicore processors, since it is not work-conserving and so wastes significant bandwidth. This impacts both worst-case and average-case performance, which are essential for application areas such as telecommunications that have a major influence on processor design.

The final approach is the one presented in this paper, based on *explicit interference modelling*. We argue that the strong interdependencies between timing analysis and schedulability analysis on multicore systems lead to undue pessimism. Thus a more nuanced and expressive interface is needed to integrate the demands placed on different resources by each task into schedulability analysis. In our approach, we omit the notion of WCET per se and instead directly target the calculation of task response times based on the demands that tasks place on the different hardware resources.

In this work, as a proof of concept, we use execution traces to model the behaviour of tasks and their resource demands. Traces provide a simple yet expressive way to model task behaviour, that captures information about the sequence of instructions and their memory accesses and corresponding types (read/write). Note that relying on execution traces does not pose a fundamental limitation to our approach as all required parameters can also be derived using static analysis (Li and Malik 1995; Ferdinand et al. 1999; Altmeyer 2013); however, traces enable a simple analysis of resource demands and so allow us to focus on an integrated response time analysis for multicore systems.

The main parameters used are the processor demand and the memory demand of each task. The latter quantity is used in analysis of the arbitration policy for the common interconnect, enabling us to upper bound the total memory access delays which may occur during the response time of a task. By computing the overall processor demand and memory demand over a relatively long interval of time (i.e. the task response time), as opposed to summing the worst case over many short intervals (e.g. individual memory accesses), we are able to obtain much tighter response time bounds.

The *Multicore Response Time Analysis framework* (*MRTA*) that we present is extensible to different types and arrangements of local memory, and different arbitration policies for the common interconnect. We instantiate the MRTA framework assuming the local memories used for instructions and data are single-level and either cache, scratchpad, or not present. In the case of caches we consider both cold caches, i.e. execution from an empty cache, and warm caches, taking advantage of information that resides in the cache from previous execution. With scratchpads, we consider both static and dynamic behaviour. Further, we assume that the memory bus arbitration policy may be TDMA, FIFO, Round-Robin, Fixed-Priority (based on task priorities), or Processor-Priority. We also account for the effects of DRAM refresh (Atanassov and Puschner 2001; Bhat and Mueller 2011). The general approach embodied in the MRTA framework is extensible to more complex, multi-level memory hierarchies, and to other sources of interference. We outline how the framework and analysis can be adapted to cover interference due to operation of the real-time operating system (RTOS) and Interrupt Handlers, we also discuss how to take into account the effects of policies used to control access to shared software resources, and how the framework can be used with open systems and for incremental verification.

The MRTA framework provides a general timing verification framework that is parametric in the hardware configuration (common interconnect, local memories, number of cores, etc.) and so can be used at the architectural design stage to compare the guaranteed levels of real-time performance that can be obtained with different hardware configurations, and also during the development and integration stages to verify the timing behaviour of a specific system. We use the framework in this way to evaluate guaranteed performance for multicore systems with a variety of different architectural components and policies. These results are then used to compose a predictable architecture, which is compared against a reference architecture designed for good average-case behaviour. This comparison shows that the predictable architecture has substantially better guaranteed real-time performance, with the precision of the analysis verified using cycle-accurate simulation.

While the specific hardware models and their mathematical representations used in this paper cannot capture all of the interference and complexity of actual hardware, they serve as a valid starting point. They include the dominant sources of interference and represent current architectures reasonably well.

1.1 Organisation

The rest of the paper is organised as follows. Section 2 discusses the related work. Section 3 describes the system model and notation used. Sections 4 and 5 show

how the effects of a local memory and the common interconnect can be modelled. Section 6 presents the nucleus of our framework, interference-aware MRTA. This analysis integrates processor and memory demands accounting for cross-core interference. Extensions to the presented analysis are discussed in Sect. 7. Section 8 describes a cycle-accurate simulator used to verify the precision of the analysis. Section 9 provides the results of an experimental evaluation using the MRTA framework, and Sect. 10 concludes with a summary and perspectives on future work.

2 Related work

The problem of estimating the response time of software real-time tasks executed on a multicore architecture is not new. The research community has been active in this area for the past ten years and an array of solutions have been proposed. Most of these solutions are, however either single-point solutions applicable to specific software and hardware models, or they focus exclusively on one particular sub-problem and abstract the rest by making simplistic assumptions. Only a few research works aim at designing a holistic analysis technique, typically because the solutions specific to the various sub-problems are very difficult to combine.

We note that it is challenging to organize the related work, since it is rare to find papers that share the same assumptions about the application and the hardware model. We therefore make a simple classification based on the primary focus of the analysis, whether it is on interference at the shared cache, memory bus, or main memory level, or on parameterizing WCETs. Further works are also covered that assume distinct application models and hardware models.

2.1 Related work with a focus on the memory bus

Although listed in this category, some of the works cited below also consider interference at the cache(s) or main memory level as well. However, they typically make either pessimistic or simplistic assumptions (like having private caches only) at those levels in order to simplify the model and thus focus on the contention for the shared memory bus.

Rosen et al. (2007) proposed an analysis technique for systems in which the shared memory bus uses TDMA arbitration and the time slots are statically assigned to the cores. This technique relies on (i) the availability of a user-programmable table-driven bus arbiter, which is typically not available in real hardware, and (ii) on the knowledge at design time of the characteristics of the entire workload that executes on each core.

In the same vein, also relying on TDMA arbitration of the memory bus, Chattopadhyay et al. (2010) and Kelter et al. (2011) proposed a response time analysis technique that considers a shared bus and an instruction cache, assuming separate buses and memories for both code and data. Their methods have limited applicability though, as they do not address data accesses to memory.

Paolieri et al. (2009) proposed a *multicore architecture* with hardware that enforces a constant upper bound on the latency of each access to a shared L2 memory through a shared bus. This approach enables the analysis of tasks in isolation since the interference on other tasks can be conservatively accounted for using this bound on the latency of each access. Similarly, the PTARM (Liu et al. 2012) enforces constant latencies for all instructions, including loads and stores; however, both cases represent customized hardware.

Lv et al. (2010) used *timed automata* to model the memory bus and the memory request patterns. Their method handles instruction accesses only and may suffer from state-space explosion when applied to data accesses. Another method employing timed automata was proposed by Gustavsson et al. (2010) in which the WCET is obtained by proving special predicates through model checking. This approach enables detailed system modelling, but is also prone to the problem of state-space explosion.

Schliecker et al. (2010) proposed a method that employs a *general event-based model* to estimate the maximum load on a shared resource. This approach makes few assumptions about the task model and is thus quite generally applicable; however, it only supports a single unspecified work-conserving bus arbiter.

Yun et al. (2012) proposed a *software-based memory throttling mechanism* to explicitly limit the memory request rate of each core and thereby control the memory interference. They also developed analytical solutions to compute proper throttling parameters that ensure the schedulability of critical tasks while minimising the performance impact of throttling.

Kelter et al. (2014) analysed the maximum bus arbitration delays for multicore systems sharing a TDMA bus and using both (private) L1 and (shared) L2 instruction and data caches.

Dasari et al. (2016) proposed a general framework to compute the maximum interference caused by the shared memory bus and its impact on the execution time of the tasks running on the cores. This method is more complex than the one proposed in this paper, and may be more accurate when it estimates the delay due to the shared bus; however, it assumes partitioned caches and therefore does not take cache-related effects into account, which makes it less general than the framework proposed here.

Jacobs et al. (2016) proposed a formal framework for the derivation of sound WCET analyses for multi-core processors. They show how to apply their analysis to account for interference on shared buses, accounting for cumulative information about the interference from tasks on other cores.

Huang et al. (2016) presented a response-time analysis that applies to multicores with one shared resource under fixed-priority arbitration. They give a simple task-to-core allocation algorithm and show that in conjunction with their response-time analysis it has a speedup factor of 7. Unlike the work in this paper, their model neither accounts for cache-related pre-emption delays nor for DRAM refreshes. Huang et al. (2016) also provide an improved response-time analysis for the case where tasks suspend from their processing core when accessing a shared resource. This is a reasonable assumption for DMA transfers, but not for individual memory accesses.

2.2 Related work with a focus on main memory

Most of the related work on memory controllers proposes scheduling algorithms that improve the controller performance, i.e., the average time to serve a sequence of requests, by (re)ordering the incoming requests at the controller level. Typically, these techniques are aimed at reducing the number of transitions between read and write modes. They seek to get the best performance from an open page policy by exploiting data locality.

Targeting real-time systems and thus time-predictability rather than performance, Kim et al. (2014a, 2016) presented a model to upper bound the memory interference delay caused by concurrent accesses to shared DRAM. Their work differs from this paper in that they primarily focus on the contention at the DRAM controller, assuming either fully-partitioned private caches or shared caches. For shared caches, they simply assume that either task preemption does not incur cache-related preemption delays (assuming in this case that cache coloring mechanisms are employed), or that the extra number of memory requests resulting from cache line evictions at runtime is given. Any further delays from shared resources, such as the memory bus, are simply assumed to be accounted for in the tasks' WCETs.

2.3 Related work with a focus on shared caches and scratchpads

Regarding the problem of estimating the WCET of tasks running in systems with shared caches, Yan and Zhang (2008) proposed a solution assuming direct-mapped, shared L2 instruction caches on multicores. The applicability of the approach is unfortunately limited as it makes very restrictive assumptions such as (i) data caches are perfect, i.e. all accesses are hits, and (ii) data references from different threads will not interfere with each other in the shared L2 cache.

Li et al. (2009a) proposed a method to estimate the worst-case response time of concurrent programs running on multicores with shared L2 caches, assuming set-associative instruction caches using the LRU replacement policy. Their work was later extended by Chattopadhyay et al. (2010) by adding a TDMA bus analysis technique to bound the memory access delay.

Regarding flash memory, Li and Mayer (2016) proposed a post-processing analysis methodology to acquire precise information about task flash memory contentions based on non-intrusive traces. For scratchpad memory, most of the works aim at reducing the WCET by proposing optimized stack management techniques (Lu et al. 2013) or dynamic code management techniques (Kim et al. 2014b), for loading program code from the main memory to the scratchpad.

Considering shared caches, there are a plethora of works that aim at reducing the impact of task pre-emptions and hence also the cache related pre-emption delays. Solutions to that problem are various: some address the problem at the task scheduling level Davis et al. (2013, 2015) by adding restrictions on the time at which tasks may be pre-empted, or at the cache management policy level (Ward et al. 2013; Mancuso et al. 2013; Slijepcevic et al. 2014). It is outside the scope of this paper to discuss all these research works.

2.4 Related work with a focus on parameterized WCETs

Rather than computing a unique upper-bound on the tasks' context independent WCET, some works propose solutions to characterize the WCET as a function of the platform characteristics and environment.

Paolieri et al. (2011) introduced an interference-aware task allocation algorithm that considers a set of WCET estimations per task, where each WCET estimation corresponds to a different execution environment (e.g. number of contending cores in a multicore system). The sensitivity of the WCET estimates to changes in the execution environment is used to guide the task to core allocation.

Reineke and Doerfert (2014) introduced *architecture-parametric* WCET analysis, which determines a function that bounds a task's WCET in terms of the amount and speed of resources allocated to that task. If a temporal-isolation approach is taken then such an analysis is required to partition shared resources in an informed manner. This analysis can also be adapted to determine WCET bounds in terms of the amount of interference on shared resources.

2.5 Related work assuming different application models

Most of the related work assumes independent tasks and only a few approaches have been proposed so far that consider task dependencies to some extent. Among them, the approach proposed by Li et al. (2009b) analyzes the worst-case cache access scenario of parallelized applications modeled by Message Sequence Graphs. The approach suffers from a very high time-complexity and assumes that the cache access behaviors are known and finite. Choi et al. (2016) used a more general model, comprising an event stream model for resource access and a task graph model for dependent tasks, in order to support a wider range of resource access patterns and parallelized execution of an application.

Schranzhofer et al. (2010) developed a framework based on a TDMA-arbitrated bus. This was followed by work on resource adaptive arbiters (Schranzhofer et al. 2011). Their work assumes a task model where each task consists of sequences of super-blocks, themselves divided into phases that represent implicit communication (fetching or writing of data to/from memory), computation (processing the data), or both. In contrast to the techniques presented in this paper, their approach requires major program intervention and compiler assistance to prefetch data. Adopting a similar model, Pellizzoni et al. (2010) compute an upper bound on the contention delay incurred by periodic tasks, for systems comprising any number of cores and peripheral buses sharing a single main memory. Their method does not cater for sporadic tasks and does not apply to systems with shared caches. In addition it relies on accurate profiling of cache utilization, suitable assignment of the TDMA time-slots to the tasks' super-blocks, and imposes a restriction on where the tasks can be pre-empted.

Pellizzoni et al. (2011) introduced the PRedictable Execution Model (PREM) framework. This framework considers tasks as consisting of memory phases where they pre-fetch instructions and data, and execution phases where they execute with-

out the need to access memory or I/O devices. The aim is to enable more efficient operation whereby the memory phase of one task overlaps with the execution phase of another. Yao et al. (2012) presented a TDMA scheduling algorithm for PREM tasks on a multicore, and Wasly and Pellizzoni (2014) provided schedulability analysis for non-preemptable PREM tasks on a partitioned multicore. Lampka et al. (2014) proposed a formal approach for bounding the worst-case response time of concurrently executing real-time tasks under resource contention and almost arbitrarily complex resource arbitration policies, with a focus on main memory as a shared resource. Global scheduling of PREM tasks has also been considered by Alhammad and Pellizzoni (2014) and Alhammad et al. (2015).

2.6 Related work assuming COTS hardware components

COTS multicore processors are typically designed to optimize average-case performance and most of their internal mechanisms are usually not documented. For such multicore platforms, Yun et al. (2015) proposed a worst-case memory interference delay analysis under the assumptions that (i) multiple memory requests can be simultaneously outstanding and (ii) the COTS DRAM controller has a separate read and write request buffer, prioritizes reads over writes, and supports out-of-order request processing. In contrast with this work, they assume non-blocking caches (common in COTS processors) that can handle multiple simultaneous cache-misses and focus solely on non-shared LLC (last level of cache) and DRAM bank partitioned systems. Non blocking caches have been the focus a multiple studies recently; however, we do not cover those techniques here. The main problem with non-blocking caches is that the miss status holding registers (MSHRs), special hardware registers which track the status of outstanding cache-misses, can be a significant source of contention (Valsan et al. 2016).

Trilla et al. (2016) proposed a timing model to predict the performance of applications at an early design stage. Their approach is based on generating an execution profile for each application that allows contention analysis on the shared processor resources. The main difference with our approach resides in their assumption that most of the hardware arbitration mechanisms are undocumented and therefore the applications' execution profiles are obtained based on an empirical analysis.

Most work on response time analysis for multicores, including this paper, assumes *timing compositionality* (Hahn et al. 2013). Intuitively, for a timing-compositional multicore, it is safe to separately account for delays from different sources, such as computation on a given core, additional cache misses due to preemptions, and interference on a shared bus. Unfortunately, recent results by (Hahn et al. 2015) indicate that even simple commercial multicores are non-compositional, rendering most existing analyses unsound for these architectures. Hahn et al. (2016), however, introduced an extended WCET analysis that enables compositional response time analysis for arbitrary, non-compositional multicores.

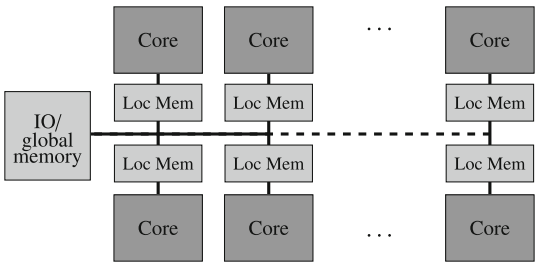

Fig. 1 Multicore platform. A set of ℓ cores with local memories connected via a common bus to a global memory

3 System model

In this paper, we provide a theoretical framework that can be instantiated for a range of different multicore architectures with different types of memory hierarchy and different arbitration policies for the common interconnect. Our aim is to create a flexible, adaptable, and generic analysis framework wherein a large number of common multicore architecture designs can be modeled and analysed. Inevitably, in this paper we can only cover a limited number of types of local memory, bus, and global memory behaviour. We select common approaches to model the different hardware components and integrate them into an extensible framework.

3.1 Multicore architectural model

We model a generic multicore platform with ℓ timing-compositional cores $P_1, \ldots P_\ell$ as depicted in Fig. 1. By timing-compositional cores we mean cores where it is safe to separately account for delays from different sources, such as computation on a given core and interference on a shared bus (Hahn et al. 2013).

The set of cores is defined as $\mathbb{P}$. Each core has a local memory which is connected via a shared bus to a global memory and IO interface. We assume constant delays d_{main} to retrieve data from global memory under the assumption of an immediate bus access, i.e., no wait-cycles or contention on the bus. We assume atomic bus transactions, i.e., no split transactions, which furthermore are not re-ordered, and non-preemptable busy waiting on the core for requests to be serviced. Further, we assume that bus access may be given to cores for one access at a time. The types of the memories and the bus policy are parameters that can be instantiated to model different multicore systems.

In this paper we assume write-through caches only and omit consideration of delays due to cache coherence and synchronization. We also consider write-through and write-back scratchpads.

3.2 Task model

We assume a set of n sporadic tasks $\{\tau_1, \ldots, \tau_n\}$; each task τ_i has a minimum period or inter-arrival time T_i and a deadline D_i. Deadlines are assumed to be constrained, hence $D_i \leq T_i$.

We assume that the tasks are statically partitioned to the set of ℓ identical cores $\{P_1, \ldots, P_\ell\}$, and scheduled on each core using fixed-priority pre-emptive scheduling. The set of tasks assigned to core P_x is denoted by Γ_x.

The index of each task is unique and thus provides a global priority order, with τ_1 having the highest priority and τ_n the lowest. The global priority of each task translates to a local priority order on each core which is used for scheduling purposes. We use $\mathrm{hp}(i)$ ($\mathrm{lp}(i)$) to denote the set of tasks with higher (lower) priority than that of task τ_i, and we use $\mathrm{hep}(i)$ ($\mathrm{lep}(i)$) to denote the set of tasks with higher or equal (lower or equal) priority to task τ_i.

We initially assume that the tasks are independent, in so far as they do not share mutually exclusive software resources (discussed in Sect. 7); nevertheless, the tasks compete for hardware resources such as the processor core, local memory, and the memory bus.

The execution of task τ_i is modelled using a set of traces O_i, where each trace $o = [\iota_1, \ldots \iota_k]$ is an ordered list of instructions. For ease of notation, we treat the ordered list of instructions as a multi-set, whenever we can abstract away from the specific order. We distinguish three types of instruction it:

$$it = \begin{cases} r[m^{\mathrm{da}}] & \text{read data from memory block } m^{\mathrm{da}} \\ w[m^{\mathrm{da}}] & \text{write data to memory block } m^{\mathrm{da}} \\ e & \text{execute} \end{cases} \tag{1}$$

An instruction ι is a triple consisting of the instruction's memory address m^{in}, its execution time Δ without memory delays, i.e., assuming a perfect local memory, and the instruction type it:

$$\iota = (m^{\mathrm{in}}, \Delta, it) \tag{2}$$

We use m to denote a memory block, and the set of memory blocks is defined as $\mathbb{M}$. $\mathbb{M}^{\mathrm{in}}$ denotes the instruction memory blocks and $\mathbb{M}^{\mathrm{da}}$ the data memory blocks. m^{in} and m^{da} are defined accordingly. We assume that data memory and instruction memory are disjoint, i.e, $\mathbb{M}^{\mathrm{in}} \cap \mathbb{M}^{\mathrm{da}} = \emptyset$.

3.2.1 Using traces to model the tasks' behaviour

The use of traces to model a task's behaviour is unusual as the number of traces is exponential in the number of control-flow branches. Despite this obvious drawback, we decided to use traces for a number of reasons:

- Traces provide a simple yet expressive way to model task behaviour. They enable a near-trivial static cache analysis and a simple multicore simulation to evaluate the accuracy of the timing verification framework.
- Traces show that the worst-case execution behaviour of a task τ_i on a multicore system is *not* uniquely defined. For example, the highest impact on a task scheduled on the same core due to task τ_i may occur when it uses that core for the longest possible time interval, whereas the highest impact on tasks scheduled on other cores may occur when task τ_i produces the largest number of bus accesses. These two cases may well correspond to different execution traces.

As a remedy for the exponential number of traces, complexity can be reduced by (i) computing a synthetic worst-case trace or by (ii) deriving the set of Pareto optimal traces that maximize the task's impact on a given performance metric or combination of different performance metrics, see (Li and Malik 1995). (The derivation of the Pareto front of traces is part of our future work.)

We note that using traces does not reduce the applicability of our approach. We use traces as a simple model which allows us to describe the computation of resource demand for various types of architectural component, and to focus on the multicore response time analysis. We can also completely resort to static analysis to derive independent upper bounds on the resource demands. For example, a static cache analysis (Ferdinand et al. 1999) can be used to bound the number of bus and also DRAM accesses. Static pre-emption cost analyses (Altmeyer 2013) are also available to bound the impact of pre-emptions, and we can use implicit path enumeration (Li and Malik 1995) to derive an upper bound on the purely computational demand of a task. These independently derived upper bounds can then be represented by a single, synthetic trace that maximizes each type of resource demand. Using static analyses in this way strongly reduces the computational complexity, but may lead to pessimism. These static analyses are, however, outside the scope of this paper, and an evaluation of the trade-off in terms of pessimism is left for future work.

3.3 Pre-emption cost model

We now extend the task model introduced above to include pre-emption costs. These costs occur when the pre-empting task evicts cache blocks of the pre-empted task that have to be reloaded after the pre-empted task resumes. To analyse the effect of pre-emption on a pre-empted task, Lee et al. (1998) introduced the concept of a useful cache block. Applying this concept to traces, a memory block m is referred to as a useful cache block (*UCB*) at a program point corresponding to instruction ι on trace o, if (i) m is cached at that program point and (ii) m is reused by a later instruction in the trace without prior eviction. In the case of pre-emption at the program point corresponding to instruction ι on trace o, only the memory blocks that (i) are cached and (ii) will be reused, may cause additional reloads. Hence, the number of UCBs at a program point gives an upper bound on the number of additional reloads due to a pre-emption at that point in the trace. A tighter definition is presented by Altmeyer and Burguière (2009); however, in this paper we need only the basic concept.

The worst-case impact of a pre-empting task is given by the number of cache blocks that the task may evict during its execution. A memory block accessed during the execution of a trace o is referred to as an evicting cache block (*ECB*). Accessing an ECB may evict a cache block of a pre-empted task. The intersection of UCBs of the pre-empted tasks with ECBs of the pre-empting task provides a tight upper bound on the cache-related pre-emption costs.

In this paper, we represent the sets of ECBs and UCBs as sets of integers with the following meaning:

$s \in \mathrm{UCB}_{\iota,o} \Leftrightarrow$ the program point ι in trace o has a useful cache block in cache-set s

$s \in \mathrm{ECB}_o \Leftrightarrow$ trace o may evict a cache block in cache-set s

We note that a separate computation of the pre-emption cost is restricted to architectures without timing anomalies (Lundqvist and Stenström 1999) but is independent of the type of cache used, i.e. data, instruction or unified cache. For examples of the use of UCBs and ECBs to compute pre-emption costs, see the work of Altmeyer et al. (2012).

3.4 Table of notation

Table 1 provides a quick reference for the notation used in this paper. Much of this notation is introduced and defined in later sections. Note we do not include in this table notation that is only used locally for the purpose of simplifying expressions.

4 Memory modelling

In this section we show how the effects of a local memory can be modelled via a MEM function which describes the number of accesses due to a task which are passed to the next level of the memory hierarchy, in this case main memory. The MEM function is instantiated for both cache and scratchpads. We model the effect of a (local) memory using a function of the form:

$$\mathrm{MEM}\colon \mathbb{O} \to \mathbb{N} \times 2^{2^{\mathbb{N}}} \times 2^{\mathbb{N}} \tag{3}$$

where $\mathbb{O}$ is the domain of traces and $\mathrm{MEM}(o) = (\mathrm{MD}_o, \overline{\mathrm{UCB}}_o, \mathrm{ECB}_o)$ computes, for a trace o, three quantities: (i) the number of bus accesses i.e., the number of memory accesses which cannot be served by the local memory alone, referred to as the memory demand MD_o; (ii) a multiset $\overline{\mathrm{UCB}}_o$ containing, for each program point ι in trace o, the set of Useful Cache Blocks (UCBs), which may need to be reloaded when trace o is pre-empted at that program point, i.e. $\overline{\mathrm{UCB}}_o = \bigcup_{\iota \in o} \{\mathrm{UCB}_{\iota\, o}\}$; (iii) the set ECB_o of Evicting Cache Blocks (ECBs) corresponding to the set of all cache blocks accessed by trace o which may evict memory blocks of other tasks from the cache. MD does not just cover cache misses, but also has to account for write accesses. In the case of write-through caches, each write access will cause a bus access, irrespective of whether or not the memory block is present in cache. (We leave integration of analysis for write-back caches (Davis et al. 2016) as future work).

MD assumes non-preemptive execution. With pre-emptive execution and caches, more than MD memory accesses can contribute to the bus contention. In this paper, we make use of the CRPD analysis for fixed-priority pre-emptive scheduling introduced by Altmeyer et al. (2012) to upper bound the additional memory accesses needed to reload cache blocks evicted due to pre-emption.

We now derive instantiations of the function $\mathrm{MEM}(o)$ for a trace $o = [\iota_1, \ldots, \iota_k]$ for instruction memories and data memories for systems (i) without cache, (ii) with

Table 1 Notation

Set of cores	$\mathbb{P} = \{P_1, \ldots, P_\ell\}$
Number of cores	ℓ
Core index	x, y
Task set	$\Gamma = \{\tau_1, \ldots, \tau_n\}$
Number of tasks	n
Task index	i, j
Tasks on core P_x	Γ_x
Task worst-case execution time with no interference	C_i
Task period	T_i
Task deadline	D_i
Task response time	R_i
Task memory demand	MD_i
Task processor demand	PD_i
Tasks with higher priority than τ_i	$\mathrm{hp}(i)$
Tasks with higher or equal priority to τ_i	$\mathrm{hep}(i)$
Tasks with lower priority than τ_i	$\mathrm{lp}(i)$
Tasks with lower or equal priority to τ_i	$\mathrm{lep}(i)$
Tasks τ_j may pre-empt within the response time of τ_i	$\mathrm{aff}(i, j)$
Bus accesses from tasks in $\mathrm{hep}(i)$ on core P_x in time t	
Where τ_i is the task under analysis	$S_i^x(t)$
Bus accesses from tasks in $\mathrm{hep}(j)$ on core P_y in time t	$A_j^y(t)$
Bus accesses from tasks in $\mathrm{lp}(j)$ on core P_y in time t	$L_j^y(t)$
Memory block	m
Set of memory blocks	$\mathbb{M}$
Instruction type	$it = \begin{cases} r[m] \\ w[m] \\ e \end{cases}$
Instruction	$\iota = (m, \Delta, it)$
Execution trace	$o = [\iota_1, \ldots \iota_k]$
Set of traces of task τ_i	O_i
Memory demand of trace o	MD_o
ECBs of trace o	ECB_o
ECBs of task τ_i	ECB_i
UCBs at instruction ι of trace o	$UCB_{\iota,o}$
Multi-set of UCBs of trace o	$\overline{\mathrm{UCB}}_o$
Memory blocks cached after executing trace o	$Cached(o)$
Cache lines that may be evicted by tasks other than τ_i	POT-EVICTED_i
Definitely cached memory blocks at the start of τ_i	DEF-CACHED_i
Cost of a pre-emption by τ_j during response time of τ_i	$\gamma_{i,j,x}$
Bus function	$\mathrm{BUS}\colon \mathbb{N} \times \mathbb{P} \times \mathbb{N} \to \mathbb{N}$
Memory function	$\mathrm{MEM}\colon \mathbb{O} \to \mathbb{N} \times \mathbb{N} \times 2^{\mathbb{N}} \times 2^{\mathbb{N}}$

Table 1 continued

Scratchpad function	$\text{SPM}: \mathbb{M} \rightarrow \{true, false\}$
Cache hit function	$\text{Hit}: \mathbb{I} \times \mathbb{M} \rightarrow \{true, false\}$
Number of bus access slots assign to each core	v
Global memory delay	d_{main}
DRAM refresh delay	d_{refresh}
DRAM refresh period	T_{refresh}
Processor interference on task τ_i on core P_x in time t	$I^{\text{PROC}}(i, x, t)$
Bus interference on task τ_i on core P_x in time t	$I^{\text{BUS}}(i, x, t)$
DRAM interference on task τ_i on core P_x in time t	$I^{\text{DRAM}}(i, x, t)$

scratchpads, and (iii) with direct-mapped or LRU caches. In the following, the superscripts indicate data (da) or instruction memory (in), and the subscripts the type of memory, i.e., uncached (nc), scratchpad (sp), or caches (ca).

4.1 Uncached

Assuming a system with no cache, considering instruction memory, the number of bus accesses MD_o for a trace o is given by the number of instructions k in the trace. The sets of UCBs and ECBs are empty, as pre-emption has no effect on the performance of the local memory, since there is none.

$$\text{MEM}^{\text{in}}_{\text{nc}}(o) = (k, \emptyset, \emptyset) \tag{4}$$

Considering data memory, we have to account for the number of data accesses, irrespective of whether they are read or write accesses. The number of accesses MD_o is thus equal to the number of data access instructions.

$$\text{MEM}^{\text{da}}_{\text{nc}}(o) = \left(\left| \left\{ \iota_i | \iota_i \in o \wedge \iota_i = (_, _, r/w[m^{\text{da}}]) \right\} \right|, \emptyset, \emptyset \right) \tag{5}$$

4.2 Scratchpads

Scratchpads are explicitly managed local memories that exhibit higher predictability and less dynamic behavior than caches. Hence, scratchpads are commonly advocated for embedded multicore systems.

Scratchpads can implement a write-back or, less commonly, a write-through policy for write accesses. Furthermore scratchpad management may be either static or dynamic. With a static scratchpad management, the scratchpad contents remain constant throughout operation, whereas with dynamic scratchpad management, scratchpad blocks can be reloaded as needed, for example on pre-emptions, which makes better use of the available scratchpad memory (Whitham et al. 2012, 2014).

4.2.1 Static scratchpads

A static scratchpad memory is defined using a function SPM: $\mathbb{M} \rightarrow \{true, false\}$, which returns $true$ for memory blocks that are stored in the scratchpad.

For an instruction scratchpad, each access to a memory block which is not stored in the scratchpad causes an additional bus access. Thus for each trace o, we have:

$$\mathrm{MEM}^{\mathrm{in}}_{\mathrm{sp}}(o) = \left(\left| \left\{ \iota_i | \iota_i \in o \wedge \iota_i = (m^{\mathrm{in}}, _, _) \wedge \neg \mathrm{SPM}(m^{\mathrm{in}}) \right\} \right|, \emptyset, \emptyset \right) \tag{6}$$

For a data scratchpad, we have to distinguish between a write-through (wt) policy, where each write-access results in a bus access:

$$\begin{aligned} \mathrm{MEM}^{\mathrm{da}}_{\text{sp-wt}}(o) = \Bigg(& \left| \left\{ \iota_i | \iota_i \in o \wedge \iota_i = (_, _, r[m^{\mathrm{da}}]) \wedge \neg \mathrm{SPM}(m^{\mathrm{da}}) \right\} \right| \\ & + \left| \left\{ \iota_i | \iota_i \in o \wedge \iota_i = (_, _, w[m^{\mathrm{da}}]) \right\} \right|, \emptyset, \emptyset \Bigg) \end{aligned} \tag{7}$$

and a write-back (wb) policy, where each accessed memory block only results in one bus access at job completion, irrespective of the number of reads and writes to it:

$$\begin{aligned} \mathrm{MEM}^{\mathrm{da}}_{\text{sp-wb}}(o) = \Bigg(& \left| \left\{ \iota_i | \iota_i \in o \wedge \iota_i = (_, _, r[m^{\mathrm{da}}]) \wedge \neg \mathrm{SPM}(m^{\mathrm{da}}) \right\} \right| \\ & + \left| \left\{ \iota_i | m^{\mathrm{da}} \in o \wedge \iota_i = (_, _, w[m^{\mathrm{da}}]) \right\} \right|, \emptyset, \emptyset \Bigg). \end{aligned} \tag{8}$$

As with other forms of local memory, the aim of a scratchpad is to reduce the number of bus accesses. An effective scratchpad configuration is obtained by storing the N most frequently used memory blocks, where N is the maximum number of memory blocks that the scratchpad can hold. With execution traces, identifying the most frequently used memory blocks is nearly trivial; however, in general more sophisticated optimization techniques have to be used (Falk and Kleinsorge 2009).

4.2.2 Dynamic scratchpads

A scratchpad can be dynamic in two respects: (i) tasks can share the scratchpad space with other tasks, and (ii) a task can reload and change the scratchpad contents during its own execution. In the first case, the scratchpad configuration has to be loaded at the beginning of a task's execution and restored after each pre-emption. In the second case, different scratchpad contents are used for different sub-traces of the task. In the following, we only present the implementation of MEM for the first case, i.e., where the scratchpad is shared among different tasks. Extension to the second case is trivial.

In a slight abuse of notation, we model the pre-emption overhead in the case of shared scratchpads using sets of UCBs and ECBs. These are used to represent the memory blocks of a task that are stored in the scratchpad. A tighter integration, which

requires dedicated hardware support is described by Whitham et al. (2012, 2014); however, analysis for it is beyond the scope of this paper. As in the static case, a scratchpad memory is defined using a function $\mathrm{SPM}\colon \mathbb{M} \to \{true, false\}$, which returns $true$ for memory blocks that are stored in the scratchpad.

For an instruction scratchpad, for each trace o we define the set of memory blocks that are accessed by the trace and are stored in the scratchpad as follows:

$$\mathrm{ECB}_o^{\mathrm{in}} = \begin{cases} \left\{ m^{\mathrm{in}} | (m^{\mathrm{in}}, _, _) \in o \wedge \mathrm{SPM}(m^{\mathrm{in}}) \right\} & \text{shared scratchpad} \\ \emptyset & \text{dedicated scratchpad} \end{cases} \tag{9}$$

Similarly for a data scratchpad:

$$\mathrm{ECB}_o^{\mathrm{da}} = \begin{cases} \left\{ m^{\mathrm{da}} | (_, _, r/w[m^{\mathrm{da}}] \in o \wedge \mathrm{SPM}(m^{\mathrm{da}}) \right\} & \text{shared scratchpad} \\ \emptyset & \text{dedicated scratchpad} \end{cases} \tag{10}$$

The set of UCBs is then defined using the set of ECBs, i.e, $\overline{\mathrm{UCB}}_o^{\mathrm{in/da}} = \{\mathrm{ECB}_o^{\mathrm{in/da}}\}$

In the case of an instruction scratchpad, each memory access to a memory block which is not stored in the scratchpad causes an additional bus access:

$$\mathrm{MEM}_{\mathrm{sp}}^{\mathrm{in}}(o) = \left(\left|\mathrm{ECB}_o^{\mathrm{in}}\right| + \left| \left\{ \iota_i | \iota_i \in o \wedge \iota_i = (m^{\mathrm{in}}, _, _) \wedge \neg \mathrm{SPM}(m^{\mathrm{in}}) \right\} \right|, \overline{\mathrm{UCB}}_o^{\mathrm{in}}, \mathrm{ECB}_o^{\mathrm{in}} \right) \tag{11}$$

The term $\left|\mathrm{ECB}_o^{\mathrm{in}}\right|$ in (11) accounts for the initialization of the scratchpad at the start of the task in the case of a shared scratchpad. (The set $\mathrm{ECB}_o^{\mathrm{in}}$ is empty in case of a dedicated scratchpad memory).

In the case of a data scratchpad, we again have to distinguish between a write-through (wt) policy and a write-back (wb) policy. Assuming a write-through policy, each write-access results in a bus access:

$$\begin{aligned} \mathrm{MEM}_{\text{sp-wt}}^{\mathrm{da}}(o) = \Bigg(& \left|\mathrm{ECB}_o^{\mathrm{da}}\right| + \left| \left\{ \iota_i | \iota_i \in o \wedge \iota_i = (_, _, r[m^{\mathrm{da}}]) \wedge \neg \mathrm{SPM}(m^{\mathrm{da}}) \right\} \right| \\ & + \left| \left\{ \iota_i | \iota_i \in o \wedge \iota_i = (_, _, w[m^{\mathrm{da}}]) \right\} \right|, \overline{\mathrm{UCB}}_o^{\mathrm{da}}, \mathrm{ECB}_o^{\mathrm{da}} \Bigg) \end{aligned} \tag{12}$$

Whereas assuming a write-back policy, each memory block that is written to causes only one bus access at job completion:

$$\begin{aligned} \mathrm{MEM}_{\text{sp-wb}}^{\mathrm{da}}(o) = \Bigg(& \left|\mathrm{ECB}_o^{\mathrm{da}}\right| + \left| \left\{ \iota_i | \iota_i \in o \wedge \iota_i = (_, _, r[m^{\mathrm{da}}]) \wedge \neg \mathrm{SPM}(m^{\mathrm{da}}) \right\} \right| \\ & + \left| \left\{ m^{\mathrm{da}} | (_, _, w[m^{\mathrm{da}}]) \in o \right\} \right|, \overline{\mathrm{UCB}}_o^{\mathrm{da}}, \mathrm{ECB}_o^{\mathrm{da}} \Bigg). \end{aligned} \tag{13}$$

4.3 Caches

Caches are commonly used in multicore systems to bridge the performance gap between processor and main memory speeds. Unlike scratchpads they require no explicit management, rather the eviction of cache blocks is determined by the cache replacement policy. In this section, we consider both *cold* caches, representing the pessimistic case where the cache is empty or contains no useful blocks when a job of task starts to execute, and *warm* caches, where some useful blocks may persist from the execution of previous jobs of the same task.

4.3.1 Cold caches

We assume a function Hit: $\mathbb{I} \times \mathbb{M} \to \{true, false\}$, which classifies each memory access at each instruction as a cache hit or a cache miss. This function can be derived using cache simulation of the access trace starting with an empty cache or by using traditional cache analysis (Ferdinand et al. 1999), where each unclassified memory access is considered a cache miss. This allows us to upper bound the number of cache misses. For each possible program point ι on trace o (i.e. for each possible preemtion point), the set of UCBs is derived using the corresponding analysis described in the thesis of Altmeyer (2013, Chap. 5, Sect. 4): a forward cache analysis derives for each program point the set of cached memory blocks, and a backward cache analysis provides the set of memory blocks that will be reused before they may be evicted. The set of UCBs per program point ι is then given by the intersection of the result of the forward and the backward cache analyses at ι. For the purpose of our analysis, it is sufficient to store only the cache sets that useful memory blocks map to. The multiset $\overline{\mathrm{UCB}}_o$ contains, for each program point ι in trace o, the set of UCBs for that program point, i.e, $\overline{\mathrm{UCB}}_o = \bigcup_{\iota \in o} \{\mathrm{UCB}_{\iota,o}\}$. The set of ECBs is the set of cache sets that memory blocks accessed in trace o map to. Finally, the memory demand of trace o is given by the number of instructions in the trace that are not cache hits.

In the case of an instruction cache, we have:

$$\mathrm{MEM}^{\mathrm{in}}_{\mathrm{ca}}(o) = \left(\left| \left\{ \iota_i \,|\, \iota_i \in o \wedge \iota_i = (m^{\mathrm{in}}, _, _) \wedge \neg \mathrm{Hit}(m^{\mathrm{in}}, \iota_i) \right\} \right|, \overline{\mathrm{UCB}}^{\mathrm{in}}_o, \mathrm{ECB}^{\mathrm{in}}_o \right) \tag{14}$$

For a data cache, since we assume a write-through policy, each write access contributes a bus access, thus:

$$\begin{aligned} \mathrm{MEM}^{\mathrm{da}}_{\mathrm{ca}}(o) = \Bigg(& \left| \left\{ \iota_i \,|\, \iota_i \in o \wedge \iota_i = (_, _, r[m^{\mathrm{da}}]) \wedge \neg \mathrm{Hit}(m^{\mathrm{da}}, \iota_i) \right\} \right| \\ & + \left| \left\{ \iota_i \,|\, \iota_i \in o \wedge \iota_i = (_, _, w[m^{\mathrm{da}}]) \right\} \right|, \overline{\mathrm{UCB}}^{\mathrm{da}}_o, \mathrm{ECB}^{\mathrm{da}}_o \Bigg) \end{aligned} \tag{15}$$

4.3.2 Warmed-up caches

Previously, we pessimistically assumed that each job of each task starts its execution with an empty cache. In reality, starting from the second job of a task, some of its instructions and data may still be cached when the job starts, reducing its execution time and memory demand. In order to capture this phenomenon, we propose an analysis that can be used to bound the response time of jobs after each task has run at least once, for example during a separate *start-up* phase.

We now consider which memory blocks can safely be assumed to be cached at the start of a job of task τ_i, when another job of that task has run before. This set of memory blocks is determined by what *must* be cached at the end of the job's execution when it runs in isolation, minus those memory blocks that *may* be evicted by a job of any task τ_j that can run *between* the two jobs of task τ_i. Since task τ_i is not active during this time, evicting jobs can belong to any task τ_j with higher or lower priority than τ_i.

Let, $Cached(o)$ be the set of memory blocks that are cached after executing trace o of task τ_i starting from an empty cache. For a given trace o, this set can be determined by simulation. The set of blocks that are definitely cached at the end of one run of the task in isolation is given by $\bigcap_{o \in O_i} Cached(o)$, where O_i is the set of traces representing the task. The set of memory blocks that are guaranteed to be cached after executing one job of a task could similarly be approximated by must-cache analysis (Ferdinand and Wilhelm 1999).

Let P_x be the core that task τ_i executes on. The set of cache lines that may be evicted by tasks other than τ_i is determined as follows:

$$\text{POT-EVICTED}_i = \bigcup_{\tau_j \in \Gamma_x \setminus \{\tau_i\}} \bigcup_{o \in O_j} \text{ECB}_o^{\text{in}} \cup \text{ECB}_o^{\text{da}}. \tag{16}$$

Given the sets defined above, we can compute the set of definitely cached memory blocks DEF-CACHED_i of task τ_i as follows:

$$\text{DEF-CACHED}_i = \left\{ b \in \bigcap_{o \in O_i} Cached(o) \middle| line(b) \notin \text{POT-EVICTED}_i \right\}, \tag{17}$$

where $line(b)$ determines the cache line that memory block b maps to.

To take information about definitely cached memory blocks into account, we assume the function Hit from Sect. 4.3.1 is extended to take into account which blocks are guaranteed to be cached initially. The MEM functions for warmed-up instruction and data caches are hence:

$$\text{MEM}^{\text{in}}_{\text{ca,w}}(o) = \left(\left| \left\{ \iota_i \,|\, \iota_i \in o \wedge \iota_i = (m^{\text{in}}, _, _) \wedge \neg \text{Hit}(m^{\text{in}}, \iota_i, \text{DEF-CACHED}_i) \right\} \right|, \right.$$
$$\left. \overline{\text{UCB}}_o^{\text{in}}, \text{ECB}_o^{\text{in}} \right) \tag{18}$$

$$\mathrm{MEM}^{\mathrm{da}}_{\mathrm{ca,w}}(o) = \left(\left| \left\{ \iota_i \,|\, \iota_i \in o \wedge \iota_i = (_, _, r[m^{\mathrm{da}}]) \wedge \neg \mathrm{Hit}(m^{\mathrm{da}}, \iota_i, \text{DEF-CACHED}_i) \right\} \right| + \left| \left\{ \iota_i \,|\, \iota_i \in o \wedge \iota_i = (_, _, w[m^{\mathrm{da}}]) \right\} \right|, \overline{\mathrm{UCB}}^{\mathrm{da}}_o, \mathrm{ECB}^{\mathrm{da}}_o \right) \tag{19}$$

Note, in order to avoid any interference effects from the start-up or warm-up phase into regular operation, we assume the following protocol:

1. In the warm-up phase, a single job of each task is run non-preemptively on its allocated core. The order of task execution is arbitrary.
2. When the final job of the warm-up phase has finished, regular operation commences, and tasks are scheduled by fixed-priority preemptive scheduling.

Apart from the above protocol no further changes to the scheduling policy or analysis are needed to account for warmed-up caches.

4.4 Memory combinations

To allow different combinations of local memories, for example scratchpad memory for instructions and an LRU cache for data, we define the combination of instruction memory $\mathrm{MEM}^{\mathrm{in}}$ and data memory $\mathrm{MEM}^{\mathrm{da}}$ as follows

$$\mathrm{MEM}(o) = \left(\mathrm{MD}^{\mathrm{in}}_o + \mathrm{MD}^{\mathrm{da}}_o, \overline{\mathrm{UCB}}^{\mathrm{in}}_o \cup \overline{\mathrm{UCB}}^{\mathrm{da}}_o, \mathrm{ECB}^{\mathrm{in}}_o \cup \mathrm{ECB}^{\mathrm{da}}_o \right) \tag{20}$$

with $\mathrm{MEM}^{\mathrm{in}}(o) = \left(\mathrm{MD}^{\mathrm{in}}_o, \overline{\mathrm{UCB}}^{\mathrm{in}}_o, \mathrm{ECB}^{\mathrm{in}}_o \right)$ being the result for the instruction memory and $\mathrm{MEM}^{\mathrm{da}}(o) = \left(\mathrm{MD}^{\mathrm{da}}_o, \overline{\mathrm{UCB}}^{\mathrm{da}}_o, \mathrm{ECB}^{\mathrm{da}}_o \right)$ the result for the data memory.

5 Bus modelling

In this section we show how the memory bus delays experienced by a task can be modelled via a BUS function of the form:

$$\mathrm{BUS} : \mathbb{N} \times \mathbb{P} \times \mathbb{N} \to \mathbb{N} \tag{21}$$

where $\mathrm{BUS}(i, x, t)$ denotes an upper bound on the number of bus accesses that can delay completion of task τ_i on core P_x during a time interval of length t. This abstraction covers a variety of bus arbitration policies, including Round-Robin, FIFO, Fixed-Priority, and Processor-Priority, all of which are *work-conserving*, and also TDMA which is not work-conserving.

We now introduce the mathematical representations of the delays incurred under these arbitration policies. We note that the framework is extensible to a wide variety of different policies. The only constraints we place on instantiations of the $\mathrm{BUS}(i, x, t)$ function is that they are monotonically non-decreasing in t.

Let τ_i be the task of interest, and x the index of the core P_x on which it executes. Other task indices are represented by j, k etc. while y, z are used for core indices.

Let $S_i^x(t)$ denote an upper bound on the total number of bus accesses due to τ_i and all higher priority tasks that run on the same core P_x during an interval of length t, while one job of task τ_i is active, i.e. within its response time. Let $A_j^y(t)$ be an upper bound on the total number of bus accesses due to all tasks of priority j or higher executing on a different core $P_y \neq P_x$ during an interval of length t. (Note, j may not necessarily be the priority of a task allocated to core P_y). In Sect. 6.3 we show how the values of $S_i^x(t)$, $A_j^y(t)$ and $L_j^y(t)$, defined below, are computed and explain why $S_i^x(t)$ and $A_j^y(t)$ are subtly different and hence require distinct notation.

As memory bus requests are typically non-preemptive, one lower priority[1] memory request may block a higher priority one, since the global shared memory may have just received a lower priority request before the higher priority one arrives. To account for these blocking accesses, we use $L_j^y(t)$ which denotes an upper bound on the total number of bus accesses due to all tasks of priority lower than j executing on some other core $P_y \neq P_x$ during an interval of length t.

In the following equations for the $\text{BUS}(i, x, t)$ function, we account for blocking due to one non-preemptive access from lower priority tasks running on the same core P_x as task τ_i (this is the $+1$ in the equations). This holds because such blocking can only occur at the start of the priority level-i (processor) busy period.

For a fixed-priority bus with memory accesses inheriting the priority of the task that generates them, we have:

$$\text{BUS}(i, x, t) = S_i^x(t) + \sum_{\forall y \neq x} A_i^y(t) + \min\left(S_i^x(t), \sum_{\forall y \neq x} L_i^y(t)\right) + 1 \tag{22}$$

Here, the term $S_i^x(t)$ covers the accesses from task τ_i and higher priority tasks running on core P_x. The term $\sum_{\forall y \neq x} A_i^y(t)$ is the interference due to accesses from higher priority tasks running on other cores. The term $\min\left(S_i^x(t), \sum_{\forall y \neq x} L_i^y(t)\right)$ upper bounds the blocking due to tasks of lower priority than τ_i running on other cores. (The number of blocking accesses is limited to one per access made by task τ_i and higher priority tasks running on core P_x during time t i.e. $S_i^x(t)$, and also restricted to the maximum number of accesses made by tasks of lower priority than τ_i running on other cores i.e. $\sum_{y \neq x} L_i^y(t)$). Finally, the $+1$ accounts for a single blocking access from a task of priority lower than that of τ_i on core P_x.

For a Processor-Priority bus with memory accesses inheriting the priority of the core rather than the task, we have:

$$\text{BUS}(i, x, t) = S_i^x(t) + \sum_{y \in HP(x)} A_n^y(t) + \min\left(S_i^x(t), \sum_{y \in LP(x)} A_n^y(t)\right) + 1 \tag{23}$$

[1] Here we mean priorities on the bus, which are not necessarily the same as task priorities.

where $HP(x)$ ($LP(x)$) is the set of cores with higher (lower) priority than that of P_x, and n is the index of the task with the lowest priority. The summation term $\sum_{y \in HP(x)} A_n^y(t)$ captures the interference from all tasks (independent of their task priority) running on cores which have a higher processor priority than P_x. The term $\min\left(S_i^x(t), \sum_{y \in LP(x)} A_n^y(t)\right)$ upper bounds the blocking due to tasks running on cores which have a processor priority lower than that of P_x.

A Round-Robin bus and a TDMA bus both make use of a pre-determined cycle of slots, where a slot is a time interval during which a single access can be serviced. Slots in the cycle are assigned to cores. With TDMA, the bus arbiter iterates over the cycle, taking the same time for each slot regardless of whether there is an access pending from the associated core, or not (i.e. the slot is empty). In contrast, a Round-Robin bus skips any empty slots, immediately moving to the next one with a pending access.

For a Round-Robin bus with v adjacent slots per core in a cycle of length $\ell \cdot v$, we have:

$$\text{BUS}(i, x, t) = S_i^x(t) + \sum_{\forall y \neq x} \min\left(A_n^y(t), v \cdot S_i^x(t)\right) + 1 \tag{24}$$

The worst-case situation for each access in $S_i^x(t)$ from core P_x occurs when it just misses its last slot in the cycle and is therefore delayed by v accesses by each core $P_y \neq P_x$. This leads to interference of at most $v \cdot S_i^x(t)$ from each core $P_y \neq P_x$; however, the overall interference from each core P_y is also bounded by the number of accesses $A_n^y(t)$, hence the $\min\left(A_n^y(t), v \cdot S_i^x(t)\right)$ term. Again, as we already account for all possible accesses from all other cores, there is no separate contribution to blocking.

For a TDMA bus with v adjacent slots per core in a cycle of length $\ell \cdot v$, we have:

$$\text{BUS}(i, x, t) = S_i^x(t) + ((\ell - 1) \cdot v) \cdot S_i^x(t) + 1 \tag{25}$$

Since TDMA is not work-conserving, the worst case corresponds to each access in $S_i^x(t)$ just missing the last slot in the cycle for core P_x and hence having to wait $((\ell - 1) \cdot v + 1)$ slots to be serviced. Effectively, there is additional interference from the $(\ell - 1) \cdot v$ slots reserved for other cores on each access, irrespective of whether these slots are used or not (in contrast to Round-Robin). Note that when $v = 1$, Eq. (25) simplifies to $\text{BUS}(i, x, t) = \ell \cdot S_i^x(t) + 1$.

It is interesting to note that while TDMA provides more predictable behaviour, this is at a cost of significantly worse guaranteed performance over long time intervals (e.g. the response time of a task) due to the fact that it is not work-conserving. Effectively, this means that the memory accesses of a task may suffer additional interference due to empty slots on the bus. Nevertheless, Round-Robin behaves like TDMA when all other cores create a large number of competing memory accesses.

We note that the equal number of slots per core for Round-Robin and TDMA, and the grouping of slots per core are simplifying assumptions to exemplify how TDMA and Round-Robin buses can be analysed. An analysis for more complex configurations (patterns of slots) is reserved for future work.

For a FIFO bus, we assume that all accesses generated on the other cores may be serviced ahead of the last access of τ_i, hence we have:

$$\text{BUS}(i, x, t) = S_i^x(t) + \sum_{\forall y \neq x} A_n^y(t) + 1 \tag{26}$$

Note that accesses from other cores do not contribute to blocking since we already pessimistically account for all these accesses in the summation term.

We note that the above analysis for a FIFO bus is potentially very pessimistic. If we assume that tasks busy wait on accesses and therefore only one access request per core can be in the FIFO queue at any given time then the worst-case situation for each access in $S_i^x(t)$ from core P_x occurs when it finds the FIFO queue already contains one access request from each of the other cores. This case can be analysed using (24) for a Round-Robin bus assuming that $v = 1$ i.e. one slot per core in the cycle. We note that some architectures may permit multiple requests to be queued by a single core (e.g. 8 in the case of the Kalray MPPA). Again, the bound on the number of requests in the queue means that the worst-case analysis equates to that for a Round-Robin bus assuming that $v = 8$ i.e. 8 slots per core in the cycle. We include pure FIFO behaviour here to illustrate the degraded performance in this case.

6 Response time analysis

In this section, we present the nucleus of our timing verification framework: interference-aware MRTA. This analysis integrates the processor and memory demands of the task of interest and higher priority tasks running on the same core, including CRPD. It also accounts for the cross-core interference on the memory bus due to tasks running on the other cores.

A task set is deemed schedulable, if for each task τ_i, its response time R_i is less than or equal to its deadline D_i:

$$\forall_i : R_i \leq D_i \Rightarrow \text{ schedulable}$$

The traditional response time calculation (Audsley et al. 1993; Joseph and Pandya 1986) for fixed-priority pre-emptive scheduling on a uniprocessor is based on an upper bound on the WCET of each task τ_i, denoted by C_i. By contrast, our MRTA framework dissects the individual components (processor and memory demands) that contribute to the WCET bound and re-assembles them at the level of the worst-case response time. It thus avoids the over-approximation inherent in using context-independent WCET bounds.

In the following, we assume that τ_i is the task of interest whose schedulability we are checking, and P_x is the core on which it runs. Recall that there is a unique global ordering of task priorities even though the scheduling is partitioned with a fixed-priority pre-emptive scheduler on each core.

6.1 Interference on the core

We compute the maximal processor demand PD_i for each task τ_i as follows:

$$\mathrm{PD}_i = \max_{o \in O_i} \sum_{(_,\Delta,_) \in o} \Delta \tag{27}$$

where Δ is the execution time of an instruction without memory delays. Task τ_i suffers interference $I^{\mathrm{PROC}}(i, x, t)$ on its core P_x due to tasks of higher priority running on the same core within a time interval of length t starting from the critical instant:

$$I^{\mathrm{PROC}}(i, x, t) = \sum_{j \in \Gamma_x \wedge j \in \mathrm{hp}(i)} \left\lceil \frac{t}{T_j} \right\rceil \mathrm{PD}_j \tag{28}$$

6.2 Interference on the local memory

Local memory improves a task's execution time by reducing the number of accesses to main memory. The memory demand of a trace gives the number of accesses that go to main memory and hence the bus, despite the presence of the local memory. The maximal memory demand MD_i of a task τ_i is defined by the maximum number of bus accesses of any of its traces:

$$\mathrm{MD}_i = \max_{o \in O_i} \left\{ \mathrm{MD} \middle| \mathrm{MEM}(o) = (\mathrm{MD}, _, _) \right\} \tag{29}$$

Note that the maximal memory demand refers to the demand of the combined instruction and data memory as defined in Eq. (20).

The memory demand MD_i is derived assuming non-preemptive execution, i.e. that the task runs to completion without interference on the local memory. The sets of UCBs and ECBs are used to compute the additional overhead due to pre-emption. In the computation of this overhead, we use the sets of UCBs per trace o to preserve precision,

$$\overline{\mathrm{UCB}}_o = \overline{\mathrm{UCB}} \; \textit{with} \; \mathrm{MEM}(o) = (_, \overline{\mathrm{UCB}}, _) \tag{30}$$

and derive the maximal set of ECBs per task τ_i as the union of the ECBs on all traces.

$$\mathrm{ECB}_i = \bigcup_{o \in O_i} \left\{ \mathrm{ECB} \middle| \mathrm{MEM}(o) = (_, _, \mathrm{ECB}) \right\} \tag{31}$$

We use $\gamma_{i,j,x}$ (with $j \in \mathrm{hp}(i)$) to denote the overhead (additional accesses) due to a pre-emption of task τ_i by task τ_j on core P_x.

We use the ECB-Union (Altmeyer et al. 2011, 2012) approach as an exemplar of CRPD analysis, as it provides a reasonably precise bound on the pre-emption overhead with low complexity. (Other CRPD analysis techniques (Altmeyer et al. 2012; Lee et al. 2001) could also be integrated into this framework). The ECB-Union

approach first computes the union of all ECBs that may affect a pre-empted task. The intuition here is that direct pre-emption by task τ_j is represented by the pessimistic assumption that task τ_j has itself already been pre-empted by all of the tasks of higher priority and hence may result in evictions due to the set $\bigcup_{h \in \text{hep}(j) \wedge h \in \Gamma_x} \text{ECB}_h$. Note that a CRPD analysis has to correctly account for all pre-emption scenarios including nested pre-emption, as it otherwise may compute optimistic bounds on the CRPD (Altmeyer et al. 2011). The ECB-Union approach considers the maximum impact of these evicting cache blocks on any job of a task τ_k that could be running during the response time of task τ_i, where τ_i is the task of interest in the response time analysis. Such a task τ_k must have a priority equal to or higher than that of task τ_i (otherwise it could not run in the busy period), but lower than that of the pre-empting task τ_j (otherwise it could not be pre-empted). These are referred to as the set of *affected* tasks $\text{aff}(i, j) = \text{hep}(i) \cap \text{lp}(j)$. Further, task τ_k must also be on core P_x. Thus the set of potentially pre-empted tasks is therefore indicated by $k \in \text{aff}(i, j) \wedge k \in \Gamma_x$. Further, pre-emption may take place at any program point in any trace of task τ_k. Putting all this together, the follow expression upper bounds the pre-emption cost by determining the maximum intersection between the evicting cache blocks of task τ_j and higher priority tasks and the useful cache blocks at any program point in any trace of any task that can be pre-empted by task τ_j during the response time of task τ_i.

$$\gamma_{i,j,x} = \max_{k \in \text{aff}(i,j) \wedge k \in \Gamma_x} \left(\max_{o \in O_k} \left(\max_{\overline{\text{UCB}_i} \in \overline{\text{UCB}_o}} \left| \left\{ \text{UCB}_\iota \cap \left(\bigcup_{h \in \text{hep}(j) \wedge h \in \Gamma_x} \text{ECB}_h \right) \right\} \right| \right) \right) \tag{32}$$

For dynamic shared scratchpads, our slight abuse of notation in defining ECBs and UCBs enables the above analysis of pre-emption costs to be used. Recall that for dynamic shared scratchpads, (9) and (10) define the ECBs for a trace as all of the memory blocks that the trace stores in the scratchpad. The UCBs are then defined as equal to the ECBs. With these definitions, the pre-emption cost analysis effectively assumes that at any program point in any trace in any pre-empted task, all of the UCBs (i.e. memory blocks stored in the scratchpad) for that trace are useful and will need reloading if they are evicted by the ECBs (i.e. memory blocks transferred to the scratchpad) by a pre-empting task.

6.3 Interference on the bus

In this section, we instantiate the functions $S_i^x(t)$, $A_j^y(t)$, and $L_j^y(t)$ that count the number of accesses from the cores and are used as input to the BUS function (see Sect. 5), which we use to derive the maximum bus delay that task τ_i on core P_x can experience during a time interval of length t:

$$I^{\text{BUS}}(i, x, t) = \text{BUS}(i, x, t) \cdot d_{\text{main}} \tag{33}$$

where d_{main} is the bus access latency to the global memory.

We first compute $S_i^x(t)$, an upper bound on the total number of bus accesses that can occur due to tasks of priority i or higher running on core P_x during an interval of length t, while one job of task τ_i is active, i.e. within its response time. Since lower priority tasks cannot execute on P_x during the response time of task τ_i (a priority level-i processor busy period), the only contribution from those tasks is a single blocking access as discussed in Sect. 5. The maximum number of accesses is computed assuming task τ_i is released simultaneously with all higher priority tasks that run on P_x, and subsequent releases of those tasks occur as soon as possible, while also assuming that the maximum possible number of pre-emptions occur.

$$S_i^x(t) = \sum_{k \in \Gamma_x \wedge k \in \text{hep}(i)} \left\lceil \frac{t}{T_k} \right\rceil \left(\text{MD}_k + \gamma_{i,k,x}\right) \tag{34}$$

MD_k denotes the memory demand of task τ_k and $\gamma_{i,k,x}$ accounts for the pre-emption costs on core P_x due to jobs of task τ_k.

In the appendix, we show that in the context of the response time analysis given in this paper, it is correct to compute $S_i^x(t)$ assuming synchronous release with higher priority tasks on the same core.

Recall that we use $A_j^y(t)$ to denote an upper bound on the total number of bus accesses due to all tasks of priority j or higher executing on core $P_y \neq P_x$ during an interval of length t. A special case is $A_n^y(t)$: since τ_n is the lowest priority task, this term includes accesses due to *all* tasks running on core P_y. In contrast to the derivation of $S_i^x(t)$, for $A_j^y(t)$ we can make no assumptions about the synchronisation or otherwise of tasks on core P_y with respect to the release of task τ_i on core P_x. The value of $A_j^y(t)$ is therefore obtained by upper bounding, for each task τ_k running on some other core P_y, the number of memory accesses that it could produce in an interval of length t, considering only the time constraints on when jobs of that task can execute. The worst-case scenario is illustrated in Fig. 2. The first job of task τ_k executes as late as possible, i.e. just prior to its worst-case response time, while the next and subsequent jobs execute as early as possible. Further, we assume that the first job of task τ_k has all of its memory accesses in a region as late as possible during its execution, while for subsequent jobs we assume the opposite is true, with execution and a region of memory accesses occurring as early as possible after release of the job. This scenario maximizes the number of memory accesses from task τ_k in an interval of length t, as shown in Fig. 2. This is similar to the concept of carry-in interference used in the analysis of global multiprocessor fixed-priority scheduling (Bertogna and Cirinei 2007; Davis and Burns 2010). Effectively due to local interference from higher priority tasks on its core, the memory accesses of the first job are *carried-in* to the interval of interest leading to increased interference on the bus during the interval.

In the following, the number of memory accesses in each region, shown in dark grey in Fig. 2, is given by $MD_k + \gamma_{j,k,y}$ and thus the length of each region is $(MD_k + \gamma_{j,k,y}) \cdot d_{\text{main}}$. We now upper bound the largest number of memory accesses in an interval of length t due to task τ_k executing on core P_k along with its cache related pre-emption effects. The next job release of task τ_k after the *carried-in* memory accesses (first dark grey region) occurs at a time $(MD_k + \gamma_{j,k,y}) \cdot d_{\text{main}} + T_k - R_k$ after the start of the

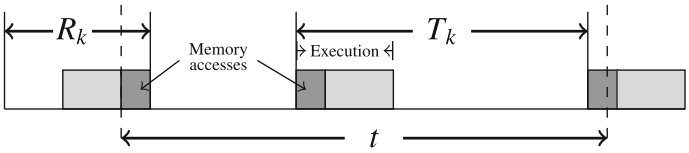

Fig. 2 Illustration of the carry-in interference analysis

interval of length t. Subsequent jobs of τ_k are then released periodically every T_k. The number of complete periods (by this we mean from the start of one memory access region to the start of the next one) that contribute memory accesses in the interval of length t is given by:

$$N^y_{j,k}(t) = \left\lfloor \frac{t - ((MD_k + \gamma_{j,k,y}) \cdot d_{\text{main}} + T_k - R_k)}{T_k} \right\rfloor + 1 = \left\lfloor \frac{t + R_k - (MD_k + \gamma_{j,k,y}) \cdot d_{\text{main}}}{T_k} \right\rfloor \tag{35}$$

The remaining incomplete period in which further memory accesses can occur is therefore of length $t + R_k - (MD_k + \gamma_{j,k,y}) \cdot d_{\text{main}} - N^y_{j,k}(t) \cdot T_k$. In this time at most one memory access can occur every d_{main}, up to $MD_k + \gamma_{j,k,y}$ accesses in total. Putting all this together, the total number of accesses possible in an interval of length t due to task τ_k (on core P_y) and its cache related pre-emption effects is given by:

$$W^y_{j,k}(t) = N^y_{j,k}(t) \cdot (MD_k + \gamma_{j,k,y}) + \min\left(MD_k + \gamma_{j,k,y}, \left\lceil \frac{t + R_k - (MD_k + \gamma_{j,k,y}) \cdot d_{\text{main}} - N^y_{j,k}(t) \cdot T_k}{d_{\text{main}}} \right\rceil\right) \tag{36}$$

We note that $W^y_{j,k}(t)$ is *sustainable* (Baruah and Burns 2006) with respect to a reduction in the number of memory accesses per job, and also to the spreading out of memory accesses within a job. Neither can cause the value of $W^y_{j,k}(t)$ to increase. This can be seen graphically by considering, in Fig. 2, what happens if the number of memory accesses is reduced i.e. the size of the grey regions is reduced, or if the memory accesses are spread out within the jobs. $W^y_{j,k}(t)$ is also monotonically non-decreasing with respect to t.

Using $W^y_{j,k}(t)$, we obtain an expression for $A^y_j(t)$, an upper bound on the total number of bus accesses due to all tasks of priority j or higher executing on core $P_y \neq P_x$ during an interval of length t, as follows:

$$A^y_j(t) = \sum_{k \in \Gamma_y \wedge k \in \text{hep}(j)} W^y_{j,k}(t) \tag{37}$$

The value of $L^y_j(t)$ is obtained in a similar way to A^y_j, but considering accesses with lower priority than j:

$$L_j^y(t) = \sum_{k \in \Gamma_y \wedge k \in \mathrm{lp}(j)} W_{n,k}^y(t) \tag{38}$$

We note that the carry-in interference was not accounted for in the analysis given by Kim et al. (2014a) (Eqs. (5) and (6) in that paper), resulting in potentially optimistic bounds on the number of competing memory requests.

6.4 Global memory—DRAM

Global memory is usually realized based on dynamic random-access memory (DRAM), which needs to be refreshed periodically. During a refresh, memory accesses cannot be serviced by the DRAM, and hence DRAM refreshes cause interference on tasks. Now, we show how to take into account delays imposed by refreshes. We assume a DRAM controller with a First Come First Served (FCFS) scheduling policy so that memory accesses cannot be reordered within the controller. Further, we assume a closed-page policy to minimize the effect of the memory access history on access latencies. We consider two refresh strategies (Micron Technologies, Inc. 1999): *distributed refresh* where the controller refreshes each row at a different time, at regular intervals, and *burst refresh* where all rows are refreshed immediately one after another.

Under distributed refresh, an upper bound on the maximum number of refreshes within an interval of length t in which m memory accesses occur is given by:

$$\mathrm{DRAM}_{\mathrm{dist}}(t, m) = \min\left(m, \left\lceil \frac{t \cdot \#\mathrm{rows}}{T_{\mathrm{refresh}}} \right\rceil\right) \tag{39}$$

where #rows is the number of rows in the DRAM module, and T_{refresh} is the interval at which each row needs to be refreshed. T_{refresh} is usually 64 ms for DDR2 and DDR3 modules. This formula holds, since at most one memory access can be delayed by each of the refreshes, whereas under burst refresh, a single memory access can be delayed by #rows many refreshes. Under burst refresh, the upper bound is given by:

$$\mathrm{DRAM}_{\mathrm{burst}}(t, m) = \left\lceil \frac{t}{T_{\mathrm{refresh}}} \right\rceil \cdot \#\mathrm{rows} \tag{40}$$

Note that the parameter m is redundant in (40); however, we keep it to retain the same signature for the function.

As the number of memory accesses within t is equal to the number of BUS accesses, we can bound the interference due to DRAM refreshes on task τ_i on core P_x as follows:

$$I^{\mathrm{DRAM}}(i, x, t) = \mathrm{DRAM}(t, \mathrm{BUS}(i, x, t)) \cdot d_{\mathrm{refresh}} \tag{41}$$

where d_{refresh} is the refresh latency.

6.5 Multicore response time analysis

The response time R_i of task τ_i is given by the smallest solution to the following recurrence relation:

$$R_i = \mathrm{PD}_i + I^{\mathrm{PROC}}(i, x, R_i) + I^{\mathrm{BUS}}(i, x, R_i) + I^{\mathrm{DRAM}}(i, x, R_i) \tag{42}$$

where PD_i is the processor demand for task τ_i given by (27), $I^{\mathrm{PROC}}(i, x, R_i)$ is the interference due to processor demand from higher priority tasks running on the same core assuming no misses on the local memory, given by (28), $I^{\mathrm{BUS}}(i, x, R_i)$ is the delay due to bus accesses from tasks running on all cores and includes MD_i, given by (33), and $I^{\mathrm{DRAM}}(i, x, R_i)$ is the delay due to DRAM refreshes, given by (41).

Since the response time of each task can depend on the response times of other tasks via the functions (37) and (38) describing memory accesses $A_j^y(t)$ and $L_j^y(t)$, we use an outer loop around a set of fixed-point iterations to compute the response times of all the tasks, and so deal with the apparent circular dependency. Iteration starts with $\forall_i : R_i = \mathrm{PD}_i + \mathrm{MD}_i \cdot d_{\mathrm{main}}$ and ends when all the response times have converged (i.e. no response time changes w.r.t. the previous iteration), or the response time of a task exceeds its deadline in which case that task is unschedulable. See Algorithm 1 for the pseudo-code of the response time calculation. Since the response time R_i of a task τ_i is monotonically increasing w.r.t. increases in the response time of any other task, convergence or exceeding a deadline is guaranteed in a bounded number of iterations.

Algorithm 1 Response Time Calculation

function MULTICORERTA
  $\forall_i : R_i^0 = 0$
  $\forall_i : R_i^1 = \mathrm{PD}_i + \mathrm{MD}_i \cdot d_{\mathrm{main}}$
  $l = 1$
  while $\exists_i : R_i^l \neq R_i^{l-1} \wedge \forall_i : R_i^l \leq D_i$ **do**
    for all i **do**
      $R_i^{l,0} = R_i^{l-1}$
      $R_i^{l,1} = R_i^l$
      $k = 1$
      while : $R_i^{l,k} \neq R_i^{l,k-1} \wedge R_i^{l,k} \leq D_i$ **do**
        $R_i^{l,k+1} = \mathrm{PD}_i + I^{\mathrm{PROC}}(i, x, R_i^{l,k}) + I^{\mathrm{BUS}}(i, x, R_i^{l,k}) + I^{\mathrm{DRAM}}(i, x, R_i^{l,k})$
        $k = k + 1$
      end while
    end for
    $\forall i : R_i^{l+1} = R_i^{l,k}$
    $l = l + 1$
  end while
  if $\forall_i : R_i^l \leq D_i$ **then**
    return schedulable
  else
    return not schedulable
  end if
end function

We note that the analysis is sustainable (Baruah and Burns 2006) with respect to the processor PD_j and memory demands MD_j of each task, since values that are smaller than the upper bounds used in the analysis cannot result in a larger response time. This sustainability extends to traces; if any trace of task execution results in practice in a lower processor or memory demand than that considered by the analysis, then this also cannot result in an increase in the response time. Similarly, a decrease in the set of UCBs or ECBs such that they are a subset of those considered by the analysis cannot increase the worst-case response time.

Note that the definitions of MD_i, PD_i and ECB_i completely decouple the traces from the response time analysis. This comes at the cost of possible pessimism, but strongly reduces the complexity of the analysis. Different traces may maximize different parameters, meaning that the combination of the parameters in this way may represent a synthetic worst-case that cannot occur in practice. An alternative solution is to define a multicore response time analysis that is parametric in the execution traces. In the extreme, completely expanding the analysis to explore every combination of traces from different tasks would be intractable. However, as a first step in this direction, response times could be computed for each individual trace of the task of interest τ_i, using combined traces for all other tasks. The maximum such response time would then provide an improved upper bound.

We note that the presented analysis framework is not fine-tuned to specific hardware features or execution scenarios such as burst accesses, since this counteracts its extensibility and generality.

7 Extensions to the task model

In the previous section we instantiated the multicore response time analysis (MRTA) framework for a relatively simple task model. In the section, we briefly discuss extensions including: RTOS and interrupts, sharing software resources, and open systems and incremental verification.

7.1 RTOS and interrupts

The analysis presented so far only considers tasks and their execution, as represented by traces. We now outline how the MRTA framework can be extended to cover RTOS and interrupt handler behaviour.

We assume that task release is triggered via interrupts from a timer/counter or other interrupt sources. When an interrupt is raised, the appropriate handler is dispatched and may pre-empt the currently executing task.[2] When the interrupt handler returns, then if a higher priority task has been released, the scheduler will run and dispatch that task, otherwise control returns to the previously running task. When a task completes, then the scheduler again runs and chooses the next highest priority task to execute.

[2] Or interrupt handler if multiple interrupt priority levels are supported.

The behaviour of each interrupt handler is represented by a set of execution traces similar to those for tasks. Thus interrupt handlers can be included in the MRTA framework in a similar way to tasks, but at higher priorities. (We note that there may be some differences if all interrupts share the same interrupt priority level; however due to the wide variety of possible arrangements of interrupt priorities, we do not go into details here). In some cases, interrupt handlers may be prohibited from using the cache, have their own cache partition, or have their code permanently locked into a scratchpad. All of these possibilities can be covered using variants of the analysis described in Sect. 6.

The RTOS is different from interrupt handlers and tasks in that it is not a schedulable entity in itself, rather RTOS code is run as part of each task, typically before and after the actual task code, and interleaved with it in the form of system calls. Similarly with interrupt handlers that release tasks, RTOS code is typically called as the handler returns. With our representation of tasks and interrupt handlers as sets of traces, execution of the RTOS can be fully accounted for by a concatenation of the appropriate sub-traces for the RTOS onto the start and end of the traces for tasks and interrupt handlers.

7.2 Sharing software resources

The analysis presented in Sect. 6 assumes that tasks are independent in the sense that they do not share software resources that must be accessed in mutual exclusion, rather the only contention is over hardware resources. We now consider how that restriction can be lifted.

We assume that tasks executing on the same core may share software resources that are accessed in mutual exclusion according to the stack resource protocol (SRP) (Baker 1991). Under SRP, a task τ_i may be blocked from executing by at most a single critical section where a task of priority lower than i locks a resource shared with task τ_i or a task of higher priority. Further, under SRP, blocking only occurs before a task starts to execute, thus SRP introduces no extra context switches. We assume a set of traces O_i^B for all of the critical sections that may block task τ_i.

In the MRTA framework, the impact of blocking needs to be considered in terms of both processor and memory demand. This can be achieved by considering the traces O_i^B as belonging to a single virtual task with higher priority than τ_i. Thus we obtain a contribution PD_i^B to the processor demand which is added into $I_i(i, x, t)$ and a contribution MD_i^B to the memory demand which contributes to $S_i^x(t)$.

Accounting for the CRPD effects due to blocking are more complex, here we make use of the approach given by Altmeyer et al. (2012) to extend the ECB-Union approach to accounting for pre-emption costs to take account of blocking.

Specifically, we extend the formula for the pre-emption cost (32) to include the UCBs of tasks in the set $b(i, j)$, where $b(i, j)$ is defined as the set of tasks with priorities lower than that of task τ_i that lock a resource with a ceiling priority higher than or equal to the priority of task τ_i but lower than that of task τ_j. These tasks can block task τ_i, but can also be pre-empted by task τ_j. Hence they need to be included in the set of tasks aff(i, j) whose UCBs are considered when determining the pre-emption cost $\gamma_{i,j,x}$ due to task τ_j:

$$\text{aff}(i, j) = (\text{hep}(i) \cap \text{lp}(j)) \cup b(i, j) \tag{43}$$

Tasks in $b(i, j)$ have lower priorities than task τ_i and so cannot pre-empt during the response time of task τ_i, hence their ECBs do not need to be considered when computing $\gamma_{i,j,x}$. Using (43) extends the ECB-Union approach (32) to correctly account for pre-emption costs when tasks share resources according to the SRP. We note that the simplest form of resource locking, sometimes called *critical sections* has the resource accesses at the highest priority. In that case, there is no additional pre-emption cost, since no tasks can pre-empt the critical sections, and so $b(i, j)$ is empty.

Blocking due to software resources accessed by tasks on other cores does not affect the term $A_n^y(t)$ since SRP introduces no additional context switches, and at the lowest priority level n, there are no extra tasks to include in the CRPD computation ($b(n, j)$ is empty, since there are no tasks of priority lower than n). The value of $A_j^y(t)$ used in the analysis of a Fixed-Priority bus is also unchanged due to resource accesses, since we assume that the bus access priority reflects only a task's base priority, rather than any raised priority as a result of SRP.

We note that accounting for resources that are shared between tasks on different cores using for example the MSRP (Gai et al. 2001) or MrsP (Burns and Wellings 2013) protocols is beyond the scope of this paper.

7.3 Open systems and incremental verification

The basic analysis for the MRTA framework given in the paper assumes that we have information (i.e. traces etc.) for all of the tasks in the system. There are a number of reasons why this may not be the case: (i) the system may be open, with tasks on one or more cores loadable post deployment, (ii) the system may be under development and the tasks on another core not yet known, (iii) incremental verification may be required, so no assumption can be made about the tasks executing on another core, (iv) the system may be mixed criticality and tasks on another core may not be developed to the same criticality level, and hence cannot be assumed to be well behaved. Instead we must assume they may exhibit the worst possible behaviour.

For a core P_y where we have no information, or need to assume the worst, we may replace $A_j^y(t)$ and $A_n^y(t)$ with a function that represents continual generation of memory accesses at the maximum possible rate. In practice, this may be equivalent to simply setting $A_j^y(t) = A_n^y(t) = \infty$. We note that analysis for TDMA and Round-Robin bus arbitration still results in bounded response times in this case, while the analysis for FIFO and Fixed-Priority arbitration will result in unbounded response times. With arbitration based on processor priority, then bounded response times can only be obtained if P_y is a lower priority processor than P_x. We note that the use of memory throttling mechanisms may result in a bounded number of memory accesses from a core P_y in any given time interval, independent of the tasks running on that core. Such a bound can be used to define an appropriate interference function $A_j^y(t)$.

8 Multicore simulator

In this section, we present a Discrete Event (DE) multicore simulator that implements the system model presented in Sect. 3. The simulator serves two purposes. Firstly, it validates the soundness of the response time analysis from Sect. 6: all task sets deemed schedulable by the analysis must not incur a deadline miss under simulation. Secondly, it enables us to investigate the precision of the analysis. The precision of timing analyses in general, and in particular timing analyses for multicore systems, is often unknown. Due to the inherent complexity and the large state-space, determining the ground-truth, i.e., a task's worst-case execution time or the exact determination of the schedulability of a task set, is computationally infeasible. Simulation cannot provide precise schedulability results; however, it can be used as a *necessary* test or over-approximation of task set schedulability. It can also provide an under-approximation of the amount, and thus the impact, of the interference on shared resources.

In this paper, we are only concerned with the timing behaviour of the multicore system, so a fully-functional simulation is not needed, hence we do not need to simulate the contents of registers or the precise arithmetic and logic operators of the cores. Therefore, we opted for a so-called transaction-level modelling (TLM) approach, as proposed by (Cai and Gajski 2003). With this approach, the interactions between different components are modelled separately from the computation that is performed by them, and each component can be modelled at a different level of detail and timing accuracy if required. For our simulator, we modelled the complete multicore system in a cycle-accurate way, implementing the exact task and system model assumed by the analysis.

The simulation needs to be capable of detecting any possible deadline miss in a given simulation run, therefore the model includes the detailed timing behaviour of bus arbitration, cache policies, core scheduling and memory accesses, but it abstracts away the individual functional behaviour of the cores (i.e. the computation they do), the bus (i.e. the data it carries) and the memory (i.e. the data it stores). According to Cai and Gajski's TLM taxonomy, this is known as a TLM implementation model.

Hardware, and in particular a multicore system is inherently parallel, whereas a DE simulator executes sequentially. To model the parallelism of the hardware, we follow the simultaneous event handling approach used in most DE simulation languages (Muliadi 1999), assuming that all actions within one cycle happen instantaneously, but ordered by infinitesimally small delays; and that signals also propagate within the same cycle to all receiving components, but ordered from upstream to downstream components. Once all signals have been propagated and processed, the simulator proceeds with the next cycle. To eliminate cyclic dependencies between the hardware components, the behaviour of the cores and the bus are each modeled within two functions, a pre-computation function and a post-computation function. The DRAM function is implemented using a single process. The dependencies of the processes and the process order of the simulator are depicted in Fig. 3.

In the following, we sketch the simulation procedure for a single execution cycle. Each core stores the set of currently active jobs that are ready to execute, and for each of these jobs a trace index. The trace index indicates how much of the trace has already been executed. The execution of a trace, and thus of a job, finishes once the

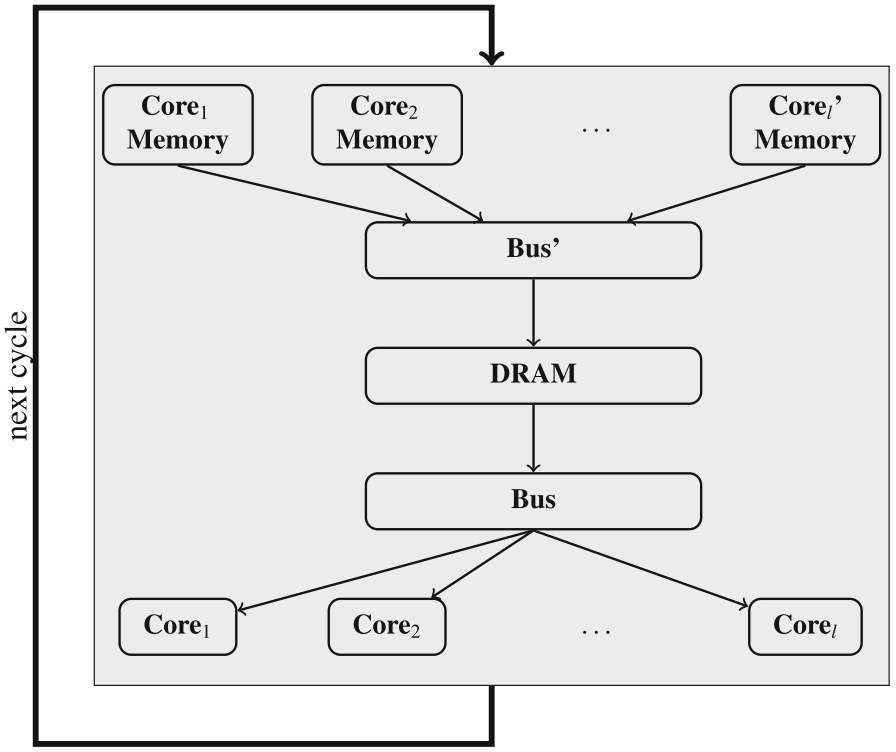

Fig. 3 Illustration of hardware dependencies and process order of the multicore simulator

final instruction of the trace is reached. The job is then set to completed. Further, each core has a boolean flag indicating whether or not the core is currently stalled waiting for a memory access to be serviced.

1. The local scheduler on each core is simulated and scheduler events such as job releases or a deadline miss are handled. If the core is not stalled, the execution of the next instruction of the currently active job with the highest priority is simulated. If the execution of this instruction incurs any memory accesses to memory blocks that are not stored in the local memory then a bus request to that memory block is issued and added to a global bus request queue, and the boolean flag is set to stalled. This step is repeated for each core.
2. The simulation proceeds with the bus controller. The bus also features a boolean flag indicating if the bus is busy or idle. If the bus is busy, the simulation immediately proceeds to the next step. If the bus is idle, the next bus access from the access queue is taken and the internal bus counter is set to the global memory latency d_{main}. The access which is taken from the queue depends on the bus arbitration policy.
3. The DRAM process counts the number of cycles and served bus accesses until a DRAM refresh is required. When a refresh is needed, the DRAM is stalled for d_{refresh} cycles.

4. Next, the simulator executes the bus post-processing, which decrements the bus counter. If the bus counter is equal to zero, a memory request is served and the post-processing for the corresponding core is invoked.
5. The post-processing for the core associated with a served memory request sets the boolean flag of the core from stalled to ready.
6. Goto step 1.

The simulation stops either after a pre-defined number of cycles, or when a deadline miss is detected. In the first case, the task set executed on the multicore system is deemed schedulable, in the second case, it is deemed unschedulable, thus providing a *necessary* test of schedulability.

Note that we assume sporadic task releases. Consequently, no initial release offsets can be assumed and all possible task interleavings are permissible. Due to the large number of release patterns full coverage of all possibilities cannot be achieved. Furthermore, the precision decreases with increasingly dynamic behaviour of the multicore, similarly, increasing the number of cores, tasks, or the size of the cache, increases the size of the state space and hence reduces the proportion of it which can be covered by simulation.

We note that one drawback of our simulator is that it does not account for any overlapping between the execution of processor instructions and memory accesses. Effectively each core always stalls waiting for its memory accesses to be serviced. Overlapping of execution and memory accesses is an area which we aim to explore in future work, both in terms of analysis and simulation.

9 Experimental evaluation

In this section we describe the results of an experimental evaluation using the MRTA framework.[3] We define a reference architecture, which serves as the baseline for the experimental evaluation. In the first set of experiments, we vary the bus architecture, and in the second set of experiments, we vary the type of local memory used. In the third experiment, we compare three distinct multicore architectures: a predictable architecture tailored towards optimizing the guaranteed real-time performance, an architecture that implements full temporal and spatial isolation, and the reference architecture. Finally, we evaluate the precision of the MRTA framework using the multicore simulator presented in Sect. 8.

Reference Architecture We model a multicore system based on an ARM Cortex A5 multicore[4] and use this as a reference architecture. As this work is intended to provide an overview of the MRTA framework, we do not model all the details of the specific multicore architecture; however, the ARM Cortex A5 provides the cache configuration, and memory and bus latencies. The reference architecture depicted in Fig. 4 is configured as follows: It has 4 ARMv7 cores connected to the global memory and I/O over a shared bus assuming a Round-Robin arbitration policy and a core

[3] The software is available on demand by contacting the first author.

[4] http://www.arm.com/products/processors/cortex-a/cortex-a5.php.

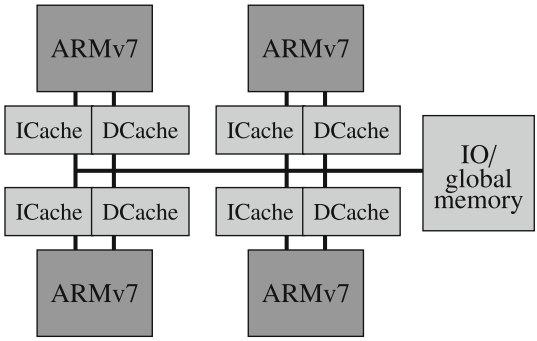

Fig. 4 Multicore architecture case study: $m = 4$ cores with local caches connected via a common bus to a global memory

frequency of 200 MHz. Each core has separate instruction and data caches, both direct-mapped with 512 cache sets each and a block size of 32 Bytes. The global memory latency d_{main} and the DRAM refresh latency d_{refresh} are both 5 cycles. The DRAM refresh period T_{refresh} is 64 ms. We assume the DRAM implements the distributed refresh strategy (see Sect. 6.4).

Trace Generation For the evaluation, we use the Mälardalen benchmark suite (Gustafsson et al. 2010) to provide traces. The traces for the benchmarks were generated using the gem5 instruction set simulator (Binkert et al. 2011) and contain statically linked library calls. As the benchmark code corresponds to independent tasks, no data is shared between the tasks. Table 2 shows information for all 39 benchmark programs used to provide traces including the total number of instructions (which is equal to the processor demand), the number of read/write operations, the memory demand, and the maximum number of UCBs and ECBs on the reference multicore architecture. Each benchmark is assigned only one trace, which is sufficient due to the simple structure of the benchmark suite: The benchmarks are either single-path by design or the input is provided as part of the benchmark suite. Despite the rather simple structure of the benchmarks, the tasks show a strong variation in processor and memory demand. As all benchmarks exhibit only one trace, the worst-case processor and memory demand coincide. Evaluation for multi-path benchmarks is left for future work, and will require a more realistic set of benchmarks than those currently available. (The Mälardalen benchmark suite contains mostly single-path benchmarks, or benchmarks with little variation between the execution paths).

Task Set Generation We evaluated the guaranteed performance of various architectural configurations as computed using the MRTA framework on a large number of randomly generated task sets. The task set parameters were as follows:

- The default task set size was 32, with 8 tasks per core.
- Each task was randomly assigned a trace from Table 2.

Table 2 Benchmark traces

Name	# Instr. (PD)	Read/Write	MD	UCB	ECB
adpcm_dec	627, 553	123, 641	38, 575	144	332
adpcm_enc	628, 795	124, 168	38, 729	155	346
binarysearch	678	293	229	20	118
bsort100	272, 715	1, 305, 613	25, 464	31	135
bs	658	201	226	19	117
cnt	7765	1573	573	33	150
compressdata	3166	1040	494	22	134
compress	8793	3358	993	74	174
countnegative	34, 860	7861	2240	74	181
cover	3661	1495	696	19	231
crc	67, 359	20, 452	6656	44	162
duff	3121	1484	553	24	130
edn	164, 596	73, 857	15, 383	104	306
expint	8058	2221	716	27	118
fac	1096	411	274	17	108
fdct	5923	3098	1088	67	193
fft1	92, 289	11, 229	4766	133	231
fibcall	1194	571	319	19	106
fir	6938	3585	1207	39	140
insertsort	2218	1317	415	18	121
janne_complex	1038	390	254	18	113
jfdctint	7771	2987	1086	63	198
lcdnum	794	326	240	22	116
lms	3, 023, 813	373, 874	120, 821	150	276
loop3	10, 539	4412	1820	16	351
ludcmp	8278	3004	768	59	189
matmult	384, 140	78, 058	11, 923	123	272
minver	16, 256	3627	1437	121	284
ndes	107, 957	50, 632	13, 186	96	252
nsichneu	8648	4841	1582	397	589
ns	25, 494	7238	1219	23	186
petrinet	2272	1206	438	160	250
qsort-exam	535	219	202	18	109
qurt	8663	1351	735	75	182
recursion	5564	1949	907	19	113
select	7211	2183	986	58	173
sqrt	26, 167	3185	1438	62	151
statemate	62, 188	51, 792	13, 360	117	235
st	1, 498, 482	125, 946	31, 969	341	429

– The base WCET per task τ_i (needed solely to set the task periods and deadline), was defined as

$$C_i = \mathrm{PD}_i + \mathrm{MD}_i \cdot d_{\mathrm{main}} + \mathrm{DRAM}(\mathrm{PD}_i + \mathrm{MD}_i \cdot d_{\mathrm{main}}, \mathrm{MD}_i) \cdot d_{\mathrm{refresh}}$$

C_i denotes the execution time of the task without any interference from any other task.
– The task utilizations were generated using UUnifast (Bini and Buttazzo 2005) with an equal utilization assumed for each core.
– Task periods were set based on task utilization and base WCET, i.e., $T_i = C_i / U_i$.
– Task deadlines were implicit.
– Priorities were assigned in deadline monotonic order.
– Tasks were assumed to be independent, i.e. no shared software resources.
– The tasks were assumed to be located in memory sequentially in priority order. (We note that improvements in layout can reduce CRPD (Lunniss et al. 2012); however, such optimisations were not considered here).

The utilization per core was varied from 0.025 to 0.975 in steps of 0.025. For each utilization value, 1000 task sets were generated and the schedulability was determined for each architectural configuration.

We note that the processor utilization is often not the limiting factor on a multicore system, but rather the memory utilization, defined as follows is:

$$U^{\mathrm{BUS}} = \sum_i \frac{\mathrm{MD}_i \cdot d_{\mathrm{main}}}{T_i} \tag{44}$$

Note the task set utilization is determined for the reference architecture and we use the same task sets throughout all experiments. Since we adapt the reference architecture and assume varying types of local memories, task set utilizations that are notionally larger than 1 can be deemed schedulable.

9.1 Bus arbitration policies

In our first set of experiments, we examine derivatives of the reference architecture assuming the different bus arbitration policies presented in Sect. 5 and also a hypothetical **perfect bus** which eliminates all bus interference if the bus utilization is ≤ 1.

Figure 5 shows the number of schedulable task sets plotted against the core utilization (computed using the base WCETs on the reference architecture) and Fig. 6 against the bus utilization U^{BUS}.

Most traces from Table 2 have a high memory demand, which results in a large number of bus accesses even at low core utilizations. Consequently, many task sets are not schedulable even with a perfect bus. The Fixed-Priority bus (green line) where the memory accesses inherit the task priority shows the best performance, followed by Round-Robin (black line) and then TDMA (purple line). Note for TDMA and Round-Robin, we assume a cycle with 2 slots per core.

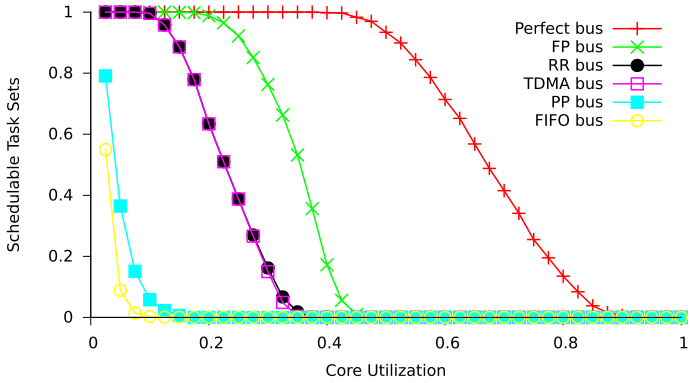

Fig. 5 Number of schedulable task sets versus core utilization: varying bus arbitration policy

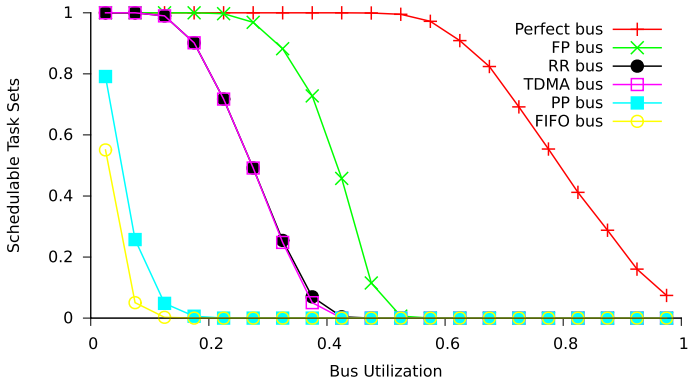

Fig. 6 Number of schedulable task sets versus bus utilization: varying bus arbitration policy

The FIFO bus shows the lowest performance, closely followed by the Processor-Priority bus (PP). The worst-case arrival pattern for a FIFO bus (yellow line) assumes that each potentially co-running task has issued bus requests just before the release of the task of interest, which results in a very pessimistic bus contention and response times. The analysis for the Processor-Priority bus (light blue line) assumes that only accesses due co-running tasks assigned to a core of higher priority cause interference, which explains the improved performance compared to the FIFO bus. We note that the task set generation does not optimize the task assignment with respect to the Processor-Priority bus. Such an optimization could greatly improve the relative performance of this policy by assigning tasks with shorter deadlines to a core with higher priority.

The difference between the Fixed-Priority and Round-Robin/TDMA policies shows the MRTA framework is able to guarantee good real-time performance even if the bus policy does not provide a tightly bounded bus latency for single accesses, as is the case with TDMA and Round-Robin.

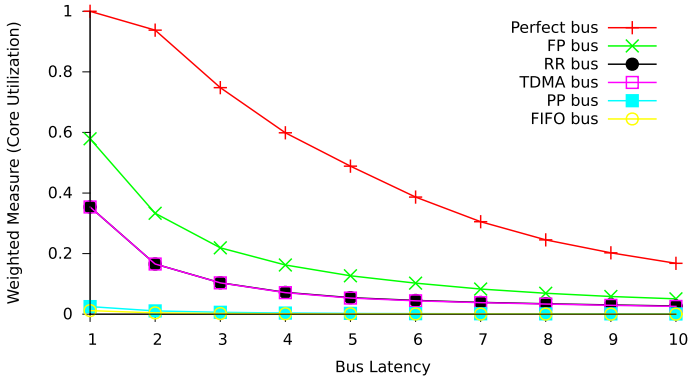

Fig. 7 Weighted schedulability: varying bus latency

9.1.1 Weighted schedulability measure

Figures 5 and 6 provide results for different bus policies, showing how guaranteed performance varies with the core and bus utilization. In our second set of experiments, we examine how other parameters including: the main memory latency, the number of cores, and the DRAM refresh latency impact schedulability. We use the weighted schedulability measure (Bastoni et al. 2010) $W_\phi(p)$ for schedulability test ϕ to show how schedulability varies across a range of values for each of these parameters p. For each value of p, this measure combines results for the task sets τ generated for all of a set of equally spaced utilization levels (0.025–0.975 in steps of 0.025).

Let $\Psi_\phi(\tau, p)$ be the binary result (1 or 0) of schedulability test ϕ for a task set τ with parameter value p:

$$W_\phi(p) = \frac{\sum_{\forall\tau}(u(\tau) \cdot \Psi_\phi(\tau, p))}{\sum_{\forall\tau} u(\tau)} \tag{45}$$

where $u(\tau)$ is the utilization of task set τ.

As the memory demand of the benchmark traces is high, the bus latency d_{main} has a tremendous impact on overall schedulability (see Fig. 7). The bus latency affects all bus arbitration policies similarly. By increasing the number of cores, the number of tasks also increases, assuming a fixed number of tasks per core, and so does the bus utilization. The performance of all configurations therefore decreases with more cores, as shown in Fig. 8, since fewer task sets are deemed schedulable irrespective of the bus policy used. As might be expected, longer DRAM refresh latencies also have a significant detrimental effect on schedulability for all policies, as shown in Fig. 9.

9.2 Local memory types

In our third set of experiments, we examine derivatives of the reference architecture assuming the different types of local memory as presented in Sect. 4, a hypothetical

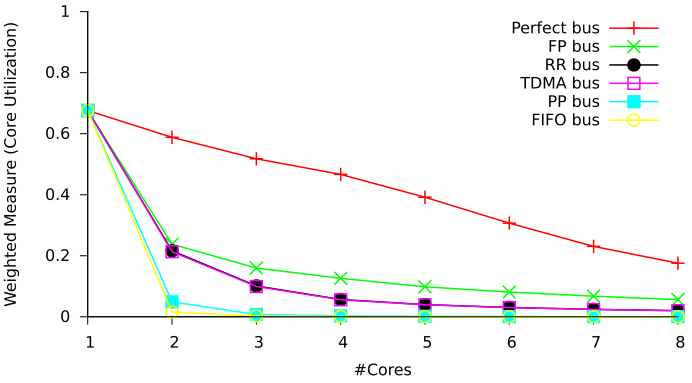

Fig. 8 Weighted schedulability: varying number of cores

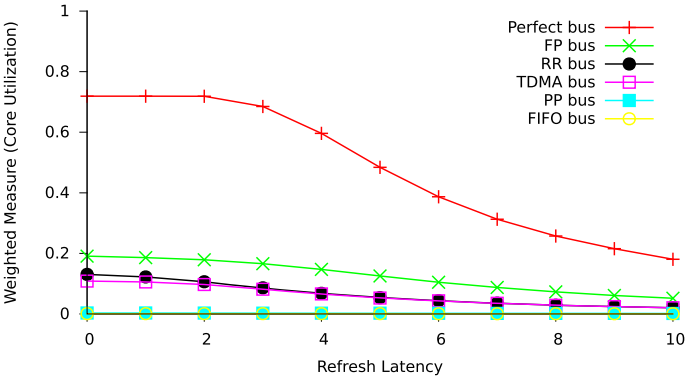

Fig. 9 Weighted schedulability: varying DRAM refresh latency

perfect cache which eliminates all memory accesses, and a configuration without any caches. Local caches are either partitioned per task, with a uniform partition size, or unconstrained, meaning that all tasks on the same core share the same local cache and can potentially use all of it. The scratchpads are partitioned per task and are configured statically to store the most frequently used memory blocks. We present results for the partitioned and unconstrained cases assuming either a cold cache, or a warmed-up cache where memory blocks have been loaded into the cache by running all tasks once during a warm-up phase (see Sect. 4). Irrespective of the type of local memory, its size remains constant at 16kB.

Figure 10 shows the number of schedulable task sets plotted against the core utilization (computed using the base WCETs on the reference architecture), and Fig. 11 the number of schedulable task sets plotted against the bus utilization U^{BUS}.

Except for uncached architectures, traditional cache architectures, i.e. both partitioned (yellow line) and unconstrained (black line), exhibit the lowest performance: at a core utilization of 0.275 and a bus utilization of 0.3 less than half of the task sets are schedulable. Accounting for a warm-up phase has an insignificant impact for unconstrained caches (light blue line), and a visible, but still somewhat limited impact on

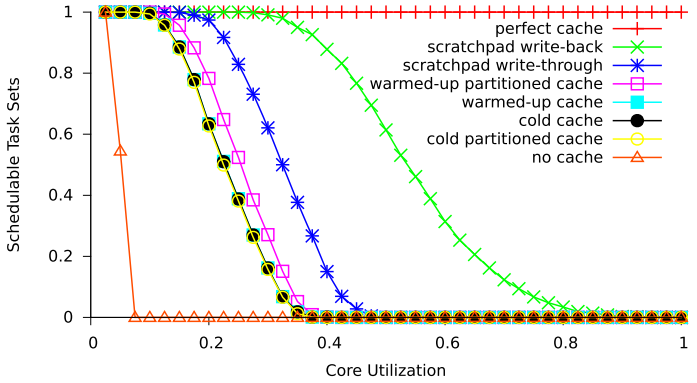

Fig. 10 Number of schedulable task sets versus core utilization: varying types of local memory

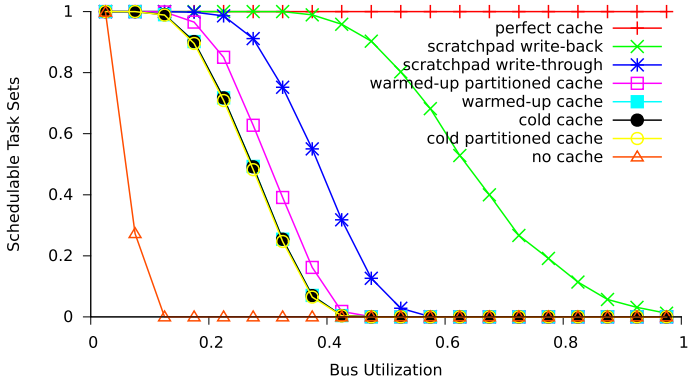

Fig. 11 Number of schedulable task sets versus bus utilization: varying types of local memory

partitioned caches (purple line). With 8 tasks per core, the combined set of ECBs often exceeds the cache size, thus the simple analysis accounting for warmed-up caches is not able to guarantee the presence of cache blocks from a previous job of the same task. Like partitioned caches, static scratchpads reduce the size of the available local memory per task. The evaluation shows, however, that the simple policy of storing the N most frequently used memory blocks in the scratchpad reduces the overall number of bus accesses significantly compared to partitioned caches. The scratchpad architecture with write-back caches (green line) offers the best performance: more than half of all task sets at a core utilization of 0.55 and at a bus utilization of 0.65 are still schedulable. The difference in performance between using a scratchpad with a write-through policy (dark blue line) compared to a write-back policy (green line) shows the substantial impact of write-accesses on the overall system performance.

9.2.1 Weighted schedulability measure

Figures 10 and 11 show the results for different types of local memory against the core and bus utilization. Using the weighted schedulability measure defined in (45), we also

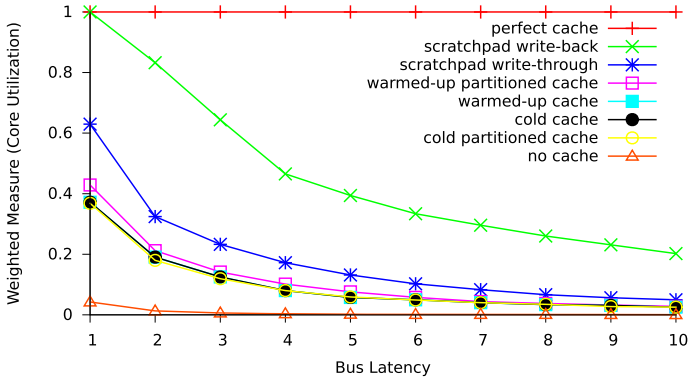

Fig. 12 Weighted schedulability: varying bus latency

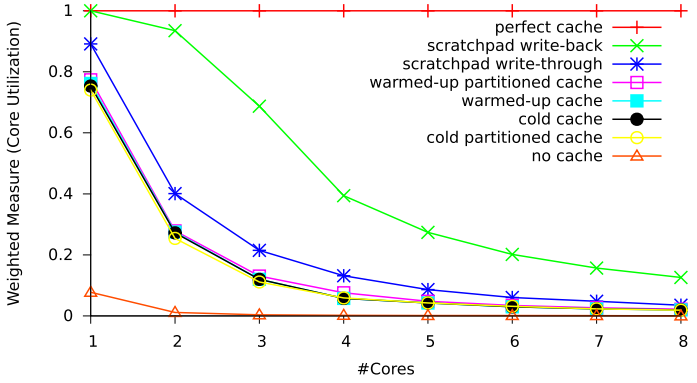

Fig. 13 Weighted schedulability: varying number of cores

examined how the main memory latency (Fig. 12), the number of cores (Fig. 13), and the DRAM refresh latency (Fig. 14) impact schedulability.

In all these cases, the performance benefit of using scratchpad memory with a write-back policy is clear, retaining its relative advantages as overall schedulability decreases.

9.3 Predictable multicore architecture

In our third set of experiments, we compare the reference architecture with two alternatives. The first, referred to as the **full-isolation architecture** is a derivative of the reference architecture that implements complete spatial and temporal isolation. The local caches are partitioned with an equal partition size for each task and the bus uses a TDMA arbitration policy. All other parameters remain the same as for the reference architecture. Performance on the isolation architecture corresponds to the traditional two-step approach to timing verification with context-independent WCETs. In addition, we note that the isolation architecture can be considered as a software

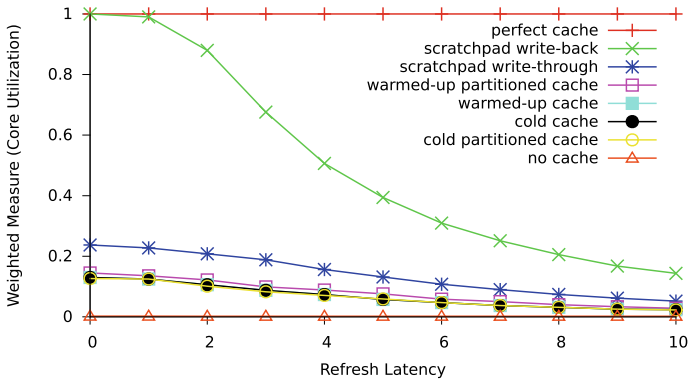

Fig. 14 Weighted schedulability: varying DRAM refresh latency

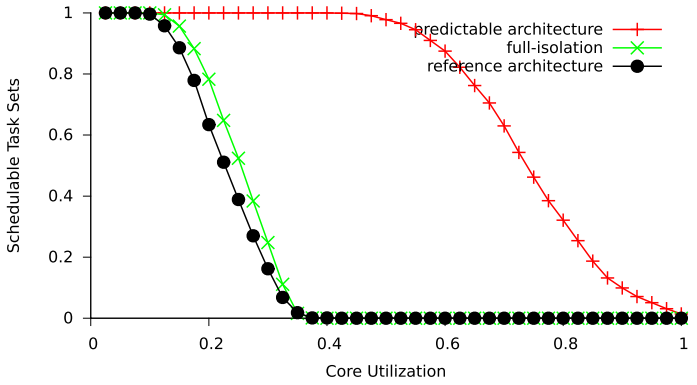

Fig. 15 Number of schedulable task sets versus core utilization

configuration of the reference architecture. The second alternative, referred to as the **predictable architecture**, has been designed to maximize the guaranteed real-time performance. From the previous evaluation, we found that using scratchpad memory with a write-back policy and a Fixed-Priority bus outperforms all other configurations in terms of guaranteed real-time performance. These components and policies were therefore chosen for the predictable architecture.

Figure 15 shows the number of schedulable task sets plotted against the core utilization (computed using the base WCETs on the reference architecture).

For the reference architecture and the full-isolation architecture, we assumed warmed-up caches as detailed in Sect. 4.

We observe that the predictable architecture, tailored towards guaranteed real-time performance, accumulates the performance benefits of both scratchpads and a Fixed-Priority bus; its performance far exceeds that of the full-isolation architecture, which uses partitioned caches and a TDMA bus, and that of the reference architecture, which uses shared caches and a Round-Robin bus.

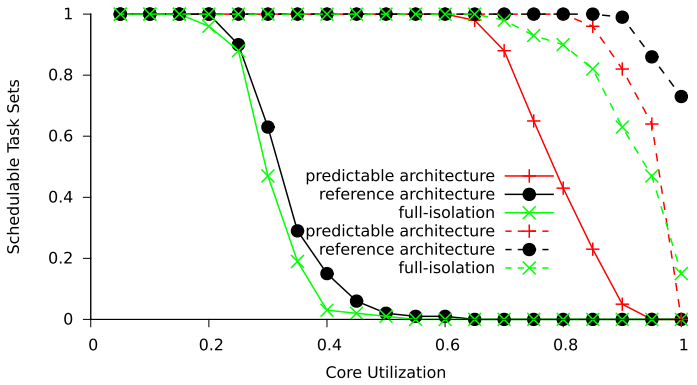

Fig. 16 Number of schedulable task sets versus core utilization: MRTA framework (*solid lines*) and multicore simulator (*dashed lines*)

9.4 Precision

Based on our experiments, we were able to identify three main sources of over-approximation (pessimism) in the MRTA framework: The number of memory accesses on the same core cannot be precisely estimated due to imprecision in the pre-emption cost analysis. The interference due to bus accesses may be pessimistic as not all tasks running on another core can simultaneously access the bus. The DRAM refreshes are assumed to occur too frequently if the number of main memory accesses are over-approximated. In this section, we examine the precision of the analysis for the different architectures using the simulator described in Sect 8.

In the previous experiments, we investigated architectures and configurations assuming a total of 32 tasks distributed over 4 cores. Since all tasks are assumed to be sporadic with no relationship between their release times, we cannot reduce the state space of possible scenarios that could be simulated. Consequently, the multicore simulator is only able to cover a negligibly small fraction of the total state space. In order to achieve a meaningful level of coverage, we reduced the number of tasks in the system to 8. The simulation runs for 10^9 cycles or until it detects a deadline miss. Further, we use the WCET in isolation on the predictable architecture, instead of on the reference architecture for the task-set generation.[5]

The results are shown in Fig. 16. The dashed lines indicate results from the simulator and solid lines results from the MRTA analysis.

The predictable architecture not only provides the highest level of guaranteed real-time performance, but also the tightest results. Both lines, the solid line for the results of the analysis and the dashed line for results from the simulator are close, which indicates a high precision in the analysis. For the other two architectures, i.e., the reference architecture and the full-isolation architecture, we cannot draw the same conclusion.

[5] We did this since using WCETs based on the reference architecture would mean that the predictable architecture could schedule some task sets with utilization >1 which makes the differences more difficult to comprehend.

In these cases, there are large differences between the lower and upper bounds on the number of schedulable task sets. We note that this could be due to imprecision in the necessary test formed by the simulation, imprecision in the sufficient test given by the MRTA framework, or both.

The reference architecture is optimized towards achieving good average case performance, whereas the predictable architecture is optimized towards guaranteed real-time performance. From the results of the simulation, we observe that the optimization targets may have been achieved. The upper bound on the number of schedulable task sets for the reference architecture is well above the upper bound for the predictable architecture. On the other hand, the predictable architecture provides significantly better guaranteed performance. The results provide an indication that multicore architectures should be tailored towards their intended use.

10 Conclusions

In this paper, we introduced the *Multicore Response Time Analysis framework (MRTA)*. This framework is extensible to different multicore architectures, with various types and arrangements of local memory, and different arbitration policies for the common interconnects. In this paper, we instantiated the MRTA framework assuming single level local data and instruction memories (cache or scratchpads), and for a variety of memory bus arbitration policies, including: Round-Robin, FIFO, Fixed-Priority, Processor-Priority, and TDMA.

The MRTA framework decouples response time analysis from a reliance on context-independent WCET values. Instead, the analysis formulates response times directly from the demands on different hardware resources. Such a separation of concerns trades different sources of pessimism. The simplifications used to make the analysis tractable are unable to take advantage of overlaps between processing and memory demands; however, this compromise is set against substantial gains acquired by considering the worst-case behaviour of resources, such as the memory bus, over long durations equating to task response times, rather than summing the worst case over short durations such as a single accesses, as is the case with the traditional two-step approach using context-independent WCETs.

While the initial instantiation of the MRTA framework given in this paper cannot capture every source of interference or delay exhibited in actual multicore processors, it captures the most significant effects. Importantly, the framework can be: (i) extended to incorporate effects due to other hardware resources, and different scheduling/resource access policies, (ii) refined to provide tighter analysis for those elements instantiated in this paper, (iii) tailored to better model the implementation of actual multicore processors.

The MRTA framework provides a general timing verification framework that is parametric in the hardware configuration (common interconnect, local memories, number of cores, etc.) and so can be used at the architectural design stage to compare the guaranteed levels of real-time performance that can be obtained with different hardware configurations, and also during the development and integration stages to verify the timing behaviour of a specific system.

We used the framework to first model, analyse and evaluate the effectiveness of different local memory components (cache and scratchpads) and bus arbitration policies with respect to a reference architecture based on a 4-core ARM Cortex A5. The evaluation utilised software from the Mälardalen benchmark suite as code for the tasks in this case study. These results were then used to compose a predictable architecture using scratchpads with a write-back policy and a Fixed-Priority bus arbitration policy, which was compared against a reference architecture designed for good average-case behaviour, and also a full isolation architecture. This comparison showed that the predictable architecture has substantially better guaranteed real-time performance than either the reference or the full isolation architecture, with the precision of the analysis verified using cycle-accurate simulation.

Our results show that while a full-isolation architecture may be preferable with the traditional two-step approach to timing verification, the MRTA framework can leverage the substantial performance improvements that can be obtained by using dynamic bus arbitration policies and components such as scratchpads designed with worst-case performance in mind.

In future, we aim to extend our work by instantiating the analysis for more complex behaviours and architectures. Examples include: (i) covering processor clusters and multi-level bus/network-on-chip (NoC) arbitration policies. Initial work in this area is reported in (Rihani et al. 2016). (ii) covering processor scheduling policies where analysis of cache-related pre-emption delays already exists, such as fixed-priority scheduling with pre-emption thresholds (Bril et al. 2014, 2017), and EDF (Lunniss et al. 2013). (iii) covering write-back caches, following recent work in this area (Davis et al. 2016; Blass et al. 2017). Further, we aim to evaluate the impact of multi-path benchmarks on the precision of the framework, and explore the optimal selection of a Pareto front of traces. We also aim to integrate the approach with the M/C task model (Melani et al. 2015, 2016) to leverage the improvements in performance that can be obtained by accounting for the overlapping of memory accesses and execution. We would also like to analyse a more complex case study with an application made up of multiple tasks running on different cores, with locally shared software resources and a simple RTOS. Our analysis framework could also be used as a means of optimising the allocation of tasks to cores.

Acknowledgements This work was supported in part by the COST Action IC1202 TACLe, by the NWO Veni Project 'The time is now: Timing Verification for Safety-Critical Multi-Cores' by the Deutsche Forschungsgemeinschaft as part of the project PEP, by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER Research Centre), by FCT/MEC and the EU ARTEMIS JU within project ARTEMIS/0001/2013—JU Grant No. 621429 (EMC2), by the INRIA International Chair program, and by the EPSRC projects MCC (EP/K011626/1) and MCCps (EP/P003664/1). EPSRC Research Data Management: no new primary data was created during this study. This collaboration was partly due to the Dagstuhl Seminar on Mixed Criticality http://www.dagstuhl.de/15121.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

Appendix: priority level-i busy periods and the validity of $S_i^x(t)$

In this paper, $S_i^x(t)$ is used to denote an upper bound on the total number of bus accesses that can occur due to tasks running on core P_x during the worst-case response time of the task of interest τ_i. In this appendix, we show that it is sound to compute $S_i^x(t)$ assuming that the job of τ_i that exhibits the largest upper bound response time (according to our analysis) is released simultaneously with all higher priority tasks on core P_x, and that these tasks are re-released as soon as possible. This is referred to as a *synchronous arrival sequence*.

In the following, we use the concept of a *busy period* defined with respect to the method used to compute an upper bound response time for task τ_i. A priority level-i busy period is defined as an interval of time during which the notional system is busy with activities (processor demand, memory accesses, DRAM refreshes etc.) which are required by the analysis to be finished before a job of task τ_i can complete its execution. (In other words, all interference is assumed to be serialised). Let $\Phi_i(t)$ be the total load in a time interval of length t that can delay the completion of τ_i, including any execution (processor demand) and memory accesses of τ_i itself. The system is considered busy with respect to τ_i (i.e. a priority level-i busy period) starting at some arbitrary time (for simplicity redefined as time zero) until time e iff $\forall t < e\ \Phi_i(t) > t$. It follows from this definition that release and completion of any specific job of task τ_i must occur within a single priority level-i busy period. Further, since all tasks have constrained deadlines, only at most one job of τ_i can execute in a single priority level-i busy period, which ends with the completion of that job. (Note, the next priority level-i busy period may start at the next discrete time instant that follows the completion of the job of τ_i).

We consider contributions to $\Phi_i(t)$ from two sources (i) external to core P_x on which τ_i executes, (ii) internal to core P_x.

In case (i), we assume the maximum possible load due to memory accesses that can be generated in an arbitrary interval of length t. This leads to the definition of $A_i^y(t)$ given in (37). We do this because for tasks on another core P_y only their memory accesses can directly impact τ_i. Their processor demand does not directly interfere with τ_i, but may do so indirectly by delaying memory accesses emanating from tasks on core P_y.

In case (ii) we assume the maximum possible load due to memory accesses in a specific priority level-i busy period in which a single job of τ_i exhibits the worst-case response time for the task. We now show that this worst-case response time occurs assuming a synchronous arrival sequence for tasks executing on the same core. This leads to the definition of $S_i^x(t)$ given in (34).

1. As τ_i is preemptable in terms of its processor demand (execution), some of which can be considered as occurring after all memory accesses and other activities which delay its completion, then since all interference on τ_i is considered serialised, there can be no interference emanating from or caused by a previous job of τ_i which impacts the job of interest. This is because, due to constrained deadlines, no two jobs of τ_i can occur within the same priority level-i busy period.

2. By definition of a priority level-i busy period, there must exist some such busy period and pattern of task releases that results in the job of task τ_i in that busy period assuming the worst-case response time for the task. Consider moving the release of the job of τ_i back to the start of this busy period. This has no effect on the length of the busy period, or on the completion time of the job, hence its computed response time cannot decrease.
3. For any task $\tau_k \in hp(i)$ executing on core P_x, then the computed response time of τ_i for the above busy period cannot be decreased by moving the release of each job of τ_k as early as possible within the busy period. Further, it cannot be impacted by any release of a job of τ_k that takes place prior to the start of the busy period. (This is the case because both the processor demand and memory accesses of τ_k interfere with τ_i and therefore form part of any priority level-i busy period).

From the three points above, it follows that in the context of our analysis, for fixed-priority preemptive (partitioned) scheduling of tasks with constrained deadlines, the computed worst-case interference (due to both processor demand and memory accesses) from tasks on the same core occurs assuming a synchronous arrival sequence, hence the definition of $S_i^x(t)$ given in (34).

The difference between $S_i^x(t)$ and $A_i^y(t)$ comes from the fact that the total interference, both processor demand and memory accesses, from tasks on the same core is maximised by synchronous arrival; whereas for tasks on a different core (which do not cause direct interference due to their processor demand) it is not.

We note that the contribution of CRPD in $S_i^x(t)$ is over-approximated similar to existing analysis (2012) by counting each release in the synchronous arrival sequence as a potential pre-emption.

References

Alhammad A, Pellizzoni R (2014) Schedulability analysis of global memory-predictable scheduling. In: Proceedings of the international conference on embedded software (EMSOFT), pp 20:1–20:10

Alhammad A, Wasly S, Pellizzoni R (2015) Memory efficient global scheduling of real-time tasks. In: Proceedings of the real-time and embedded technology and applications symposium (RTAS), pp 285–296

Altmeyer S (2013) Analysis of preemptively scheduled hard real-time systems. epubli GmbH. http://www.ebay.de/itm/Analysis-of-Preemptively-Scheduled-Hard-Real-time-Systems-Sebastian-Altmeyer-/142397905969

Altmeyer S, Burguière C (2009) A new notion of useful cache block to improve the bounds of cache-related preemption delay. In: Proceedings of the euromicro conference on real-time systems (ECRTS), pp 109–118

Altmeyer S, Davis RI, Maiza C (2011) Cache related pre-emption aware response time analysis for fixed priority pre-emptive systems. In: Proceedings of the real-time systems symposium (RTSS), pp 261–271

Altmeyer S, Davis RI, Maiza C (2012) Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. Real-Time Syst 48(5):499–526

Altmeyer S, Douma R, Lunniss W, Davis RI (2014) Evaluation of cache partitioning for hard real-time systems. In: Proceedings of the euromicro conference on real-time systems (ECRTS), pp 15–26

Altmeyer S, Davis RI, Indrusiak L, Maiza C, Nelis V, Reineke J (2015) A generic and compositional framework for multicore response time analysis. In: Proceedings of the international conference on real time networks and systems (RTNS), pp 129–138

Altmeyer S, Douma R, Lunniss W, Davis RI (2016) On the effectiveness of cache partitioning in hard real-time systems. Real-Time Syst. doi:10.1007/s11241-015-9246-8

Atanassov P, Puschner P (2001) Impact of DRAM refresh on the execution time of real-time tasks. In: IEEE international workshop on application of reliable computing and communication, pp 29–34
Audsley N, Burns A, Richardson M, Tindell K, Wellings AJ (1993) Applying new scheduling theory to static priority pre-emptive scheduling. Softw Eng J 8:284–292
Baker TP (1991) Stack-based scheduling for realtime processes. Real-Time Syst 3:67–99
Baruah S, Burns A (2006) Sustainable scheduling analysis. In: Proceedings of the real-time systems symposium (RTSS), pp 159–168
Bastoni A, Brandenburg B, Anderson J (2010) Cache-related preemption and migration delays: empirical approximation and impact on schedulability. In: Proceedings of the workshop on operating systems platforms for embedded real-time applications (OSPERT), pp 33–44
Bertogna M, Cirinei M (2007) Response-time analysis for globally scheduled symmetric multiprocessor platforms. In: Proceedings of the Real-Time Systems Symposium (RTSS), pp 149–160
Bhat B, Mueller F (2011) Making DRAM refresh predictable. Real-Time Syst 47(5):430–453
Bini E, Buttazzo G (2005) Measuring the performance of schedulability tests. Real-Time Syst 30:129–154
Binkert N et al (2011) The gem5 simulator. SIGARCH Comput Archit News 39(2):1–7
Blass T, Hahn S, Reineke J (2017) Write-back caches in WCET analysis. In: Proceedings of the euromicro conference on real-time systems (ECRTS)
Bril RJ, Altmeyer S, van den Heuvel MMHP, Davis RI, Behnam M (2014) Integrating cache-related pre-emption delays into analysis of fixed priority scheduling with pre-emption thresholds. In: Proceedings of the real-time systems symposium (RTSS), pp 161–172
Bril RJ, Altmeyer S, van den Heuvel MM, Davis RI, Behnam M (2017) Fixed priority scheduling with pre-emption thresholds and cache-related pre-emption delays: integrated analysis and evaluation. Real-Time Syst. doi:10.1007/s11241-016-9266-z
Bui D, Lee E, Liu I, Patel H, Reineke J (2011) Temporal isolation on multiprocessing architectures. In: Proceedings of the design automation conference (DAC), pp 274–279
Burns A, Wellings AJ (2013) A schedulability compatible multiprocessor resource sharing protocol—MRSP. In: Proceedings of the euromicro conference on real-time systems (ECRTS), pp 282–291
Cai L, Gajski D (2003) Transaction level modeling: an overview. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES), pp 19–24
Chattopadhyay S, Roychoudhury A, Mitra T (2010) Modeling shared cache and bus in multi-cores for timing analysis. In: Proceedings of the international workshop on software and compilers for embedded systems (SCOPES), pp 6:1–6:10
Choi J, Kang D, Ha S (2016) Conservative modeling of shared resource contention for dependent tasks in partitioned multi-core systems. In: Proceedings of design, automation, and test in Europe (DATE), pp 181–186
Dasari D, Nelis V, Akesson B (2016) A framework for memory contention analysis in multi-core platforms. Real-Time Syst 52(3):272–322
Davis RI, Burns A (2010) Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. Real-Time Syst 47(1):1–40
Davis RI, Burns A, Marinho J, Nelis V, Petters SM, Bertogna M (2013) Global fixed priority scheduling with deferred pre-emption. In: Proceedings of the international conference on embedded and real-time computing systems and applications (RTCSA), pp 1–11
Davis RI, Burns A, Marinho J, Nelis V, Petters SM, Bertogna M (2015) Global and partitioned multiprocessor fixed priority scheduling with deferred preemption. ACM TECS 14(3):47:1–47:28
Davis RI, Altmeyer S, Reineke J (2016) Analysis of write-back caches under fixed-priority preemptive and non-preemptive scheduling. In: Proceedings of the international conference on real-time networks and systems (RTNS), pp 309–318
Falk H, Kleinsorge J (2009) Optimal static WCET-aware scratchpad allocation of program code. In: Proceedings of the design automation conference (DAC), pp 732–737
Ferdinand C, Wilhelm R (1999) Efficient and precise cache behavior prediction for real-time systems. Real-Time Syst 17(2–3):131–181
Ferdinand C, Martin F, Wilhelm R, Alt M (1999) Cache behavior prediction by abstract interpretation. Sci Comput Program 35(2–3):163–189
Gai P, Lipari G, Natale MD (2001) Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In: Proceedings of the real-time systems symposium (RTSS), pp 73–83

Gustafsson J, Betts A, Ermedahl A, Lisper B (2010) The Mälardalen WCET benchmarks—past, present and future. In: Proceedings of the international workshop on worst-case execution time analysis (WCET), pp 137–147

Gustavsson A, Ermedahl A, Lisper B, Pettersson P (2010) Towards WCET analysis of multicore architectures using UPPAAL. In: Proceedings of the international workshop on worst-case execution time analysis (WCET), pp 101–112

Hahn S, Reineke J, Wilhelm R (2013) Towards compositionality in execution time analysis—definition and challenges. In: Proceedings of the international workshop on compositional theory and technology for real-time embedded systems (CRTS)

Hahn S, Reineke J, Wilhelm R (2015) Toward compact abstractions for processor pipelines. In: Proceedings of the correct system design—symposium in honor of Ernst-Rüdiger Olderog on the occasion of his 60th birthday, pp 205–220

Hahn S, Jacobs M, Reineke J (2016) Enabling compositionality for multicore timing analysis. In: Proceedings of the international conference on real time and networks systems (RTNS)

Huang WH, Chen JJ, Reineke J (2016) MIRROR: symmetric timing analysis for real-time tasks on multicore platforms with shared resources. In: Proceedings of the design automation conference (DAC), pp 1–6

Jacobs M, Hahn S, Hack S (2016) A framework for the derivation of WCET analyses for multi-core processors. In: Proceedings of the euromicro conference on real-time systems (ECRTS), pp 141–151

Joseph M, Pandya P (1986) Finding response times in a real-time system. Comput J 29(5):390–395

Kelter T, Falk H, Marwedel P, Chattopadhyay S, Roychoudhury A (2011) Bus-aware multicore WCET analysis through TDMA offset bounds. In: Proceedings of the euromicro conference on real-time systems (ECRTS), pp 3–12

Kelter T, Falk H, Marwedel P, Chattopadhyay S, Roychoudhury A (2014) Static analysis of multi-core TDMA resource arbitration delays. Real-Time Syst J 50(2):185–229

Kim H, de Niz D, Andersson B, Klein M, Mutlu O, Rajkumar R (2014a) Bounding memory interference delay in COTS-based multi-core systems. In: Proceedings of the real-time and embedded technology and applications symposium (RTAS), pp 145–154

Kim Y, Broman D, Cai J, Shrivastaval A (2014b) Wcet-aware dynamic code management on scratchpads for software-managed multicores. In: Proceedings of the real-time and embedded technology and applications symposium (RTAS), pp 179–188

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

Lampka K, Giannopoulou G, Pellizzoni R, Wu Z, Stoimenov N (2014) A formal approach to the WCRT analysis of multicore systems with memory contention under phase-structured task sets. Real-Time Syst 50(5–6):736–773

Lee CG, Hahn J, Seo YM, Min S, Ha R, Hong S, Park CY, Lee M, Kim CS (1998) Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. IEEE Trans Comput 47(6):700–713

Lee C, Lee K, Hahn J, Seo YM, Min SL, Ha R, Hong S, Park CY, Lee M, Kim CS (2001) Bounding cache-related preemption delay for real-time systems. IEEE Trans Softw Eng 27(9):805–826

Li YTS, Malik S (1995) Performance analysis of embedded software using implicit path enumeration. In: Proceedings of the design automation conference (DAC), pp 456–461

Li L, Mayer A (2016) Trace-based analysis methodology of program flash contention in embedded multicore systems. In: Proceedings of design, automation, and test in Europe (DATE), pp 199–204

Li Y, Suhendra V, Liang Y, Mitra T, Roychoudhury A (2009a) Timing analysis of concurrent programs running on shared cache multi-cores. In: Proceedings of the real-time systems symposium (RTSS), pp 57–67

Li Y, Suhendra V, Liang Y, Mitra T, Roychoudhury A (2009b) Timing analysis of concurrent programs running on shared cache multi-cores. In: Proceedings of the real-time systems symposium, pp 57–67

Liu I, Reineke J, Broman D, Zimmer M, Lee EA (2012) A PRET microarchitecture implementation with repeatable timing and competitive performance. In: Proceeedings of the international conference on computer design (ICCD), pp 87–93

Lu J, Bai K, Shrivastava A (2013) Ssdm: Smart stack data management for software managed multicores (SMMS). In: Proceedings of the design automation conference (DAC), pp 1–8

Lundqvist T, Stenström P (1999) Timing anomalies in dynamically scheduled microprocessors. In: Proceedings of the real-time systems symposium (RTSS), pp 12–21

Lunniss W, Altmeyer S, Davis RI (2012) Optimising task layout to increase schedulability via reduced cache related pre-emption delays. In: Proceedings of the international conference on real-time networks and systems (RTNS), pp 161–170

Lunniss W, Altmeyer S, Maiza C, Davis R (2013) Integrating cache related pre-emption delay analysis into EDF scheduling. In: Proceedings of the real-time and embedded technology and applications symposium (RTAS), pp 75–84

Lv M, Yi W, Guan N, Yu G (2010) Combining abstract interpretation with model checking for timing analysis of multicore software. In: Proceedings of the real-time systems symposium (RTSS), pp 339–349

Mancuso R, Dudko R, Betti E, Cesati M, Caccamo M, Pellizzoni R (2013) Real-time cache management framework for multi-core architectures. In: Proceedings of the real-time and embedded technology and applications symposium (RTAS), pp 45–54

Melani A, Bertogna M, Bonifaci V, Marchetti-Spaccamela A, Buttazzo G (2015) Memory-processor co-scheduling in fixed priority systems. In: Proceedings of the international conference on real-time networks and systems (RTNS), pp 87–96

Melani A, Bertogna M, Davis RI, Bonifaci V, Marchetti-Spaccamela A, Buttazzo G (2016) Exact response time analysis for fixed priority memory-processor co-scheduling. IEEE Trans Comput. doi:10.1109/TC.2016.2614819

Micron Technologies, Inc (1999) Various methods of DRAM refresh. Tech. rep

Muliadi L (1999) Discrete event modeling in ptolemy II. Master's report, University of California, Berkeley. http://ptolemy.eecs.berkeley.edu/publications/papers/99/deModeling/

Nowotsch J, Paulitsch M, Buhler D, Theiling H, Wegener S, Schmidt M (2014) Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In: Proceedings of the euromicro conference on real-time systems (ECRTS), pp 109–118

Paolieri M, Quiñones E, Cazorla FJ, Bernat G, Valero M (2009) Hardware support for WCET analysis of hard real-time multicore systems. SIGARCH Comput Archit News 37(3):57–68

Paolieri M, Quiñones E, Cazorla FJ, Davis RI, Valero M (2011) Ia^3: An interference aware allocation algorithm for multicore hard real-time systems. In: Proceedings of the real-time and embedded technology and applications symposium (RTAS), pp 280–290

Pellizzoni R, Schranzhofer A, Chen JJ, Caccamo M, Thiele L (2010) Worst case delay analysis for memory interference in multicore systems. In: Proceedings of design automation and test in Europe (DATE), pp 741–746

Pellizzoni R, Betti E, Bak S, Criswell J, Caccamo M, Kegley R (2011) A predictable execution model for COTS-based embedded systems. In: Proceedings of the real-time and embedded technology and applications symposium (RTAS), pp 269–279

Radojković P, Girbal S, Grasset A, Quiñones E, Yehia S, Cazorla FJ (2012) On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. ACM TACO 8(4):34

Reineke J, Doerfert J (2014) Architecture-parametric timing analysis. In: Proceedings of the real-time and embedded technology and applications symposium (RTAS), pp 189–200. doi:10.1109/RTAS.2014.6926002

Rihani H, Moy M, Maiza C, Davis RI, Altmeyer S (2016) Response time analysis of synchronous data flow programs on a many-core processor. In: Proceedings of the international conference on real-time networks and systems (RTNS), ACM, pp 67–76

Rosen J, Andrei A, Eles P, Peng Z (2007) Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In: proceedings of the real-time systems symposium (RTSS), pp 49–60

Schliecker S, Negrean M, Ernst R (2010) Bounding the shared resource load for the performance analysis of multiprocessor systems. In: Proceedings of the design automation conference (DAC), pp 759–764

Schranzhofer A, Chen JJ, Thiele L (2010) Timing analysis for TDMA arbitration in resource sharing systems. In: Proceedings of the real-time and embedded technology and applications symposium (RTAS), pp 215–224

Schranzhofer A, Pellizzoni R, Chen JJ, Thiele L, Caccamo M (2011) Timing analysis for resource access interference on adaptive resource arbiters. In: Proceedings of the real-time and embedded technology and applications symposium (RTAS), pp 213–222

Slijepcevic M, Kosmidis L, Abella J, Quiones E, Cazorla FJ (2014) Time-analysable non-partitioned shared caches for real-time multicore systems. In: Proceedings of the design automation conference (DAC), pp 1–6

Trilla D, Jalle J, Fernandez M, Abella J, Cazorla FJ (2016) Improving early design stage timing modeling in multicore based real-time systems. In: Proceedings of the real-time and embedded technology and applications symposium (RTAS), pp 1–12

Valsan PK, Yun H, Farshchi F (2016) Taming non-blocking caches to improve isolation in multicore real-time systems. In: Proceedings of the real-time and embedded technology and applications symposium (RTAS), pp 1–12

Ward BC, Herman JL, Kenna CJ, Anderson JH (2013) Making shared caches more predictable on multicore platforms. In: Proceedings of the euromicro conference on real-time systems, pp 157–167

Wasly S, Pellizzoni R (2014) Hiding memory latency using fixed priority scheduling. In: Proceedings of the IEEE real-time and embedded technology and applications symposium (RTAS), pp 75–86

Whitham J, Davis RI, Audsley N, Altmeyer S, Maiza C (2012) Investigation of scratchpad memory for preemptive multitasking. In: Proceedings of the real-time systems symposium (RTSS), pp 3–13

Whitham J, Audsley N, Davis RI (2014) Explicit reservation of cache memory in a predictable, preemptive multitasking real-time system. ACM Trans Embed Comput Syst 13(4s):120:1–120:25

Yan J, Zhang W (2008) WCET analysis for multi-core processors with shared L2 instruction caches. In: Proceedings of the real-time and embedded technology and applications symposium (RTAS), pp 80–89

Yao G, Pellizzoni R, Bak S, Betti E, Caccamo M (2012) Memory-centric scheduling for multicore hard real-time systems. Real-Time Syst J 48(6):681–715

Yun H, Yao G, Pellizzoni R, Caccamo M, Sha L (2012) Memory access control in multiprocessor for real-time systems with mixed criticality. In: Proceedings of the euromicro conference on real-time systems (ECRTS), pp 299–308

Yun H, Pellizzoni R, Valsan PK (2015) Parallelism-aware memory interference delay analysis for COTS multicore systems. In: Proceedings of the euromicro conference on real-time Systems, pp 184–195

Robert I. Davis is a Senior Research Fellow in the Real-Time Systems Research Group at the University of York, UK, and an Inria International Chair with Inria, Paris, France. Robert received his DPhil in Computer Science from the University of York in 1995. Since then he has founded three start-up companies, all of which have succeeded in transferring real-time systems research into commercial products. Robert's research interests include the following aspects of real-time systems: scheduling algorithms and analysis for single processor, multiprocessor and networked systems; analysis of cache related pre-emption delays, mixed criticality systems, and probabilistic hard real-time systems.

Sebastian Altmeyer is a researcher at the University of Amsterdam, where he has received a 2015 NWO Veni grant on the timing verification of real-time multicore systems. Prior to this he has been a postdoctoral researcher at the University of Luxembourg and the University of Amsterdam. He has received his PhD in Computer Science in 2012 from Saarland University, Germany with a thesis on the analysis of pre-emptively scheduled hard real-time systems. His research focuses on various aspects on the design, analysis and verification of hard real-time systems, with a particular focus timing verification.

Leandro S. Indrusiak graduated in Electrical Engineering from the Federal University of Santa Maria (UFSM, Brazil) and obtained a MSc in Computer Science from the Federal University of Rio Grande do Sul (UFRGS, Brazil) in 1995 and 1998, respectively. He held a tenured assistant professorship at the Informatics department of the Catholic University of Rio Grande do Sul (Brazil) from 1998 to 2000. From 2001 to 2008 he worked as a researcher at the Technische Universitaet Darmstadt (Germany) where he worked towards a PhD and then led a research team on the area of System-on-Chip design. His binational doctoral degree was jointly awarded by UFRGS and TU Darmstadt in 2003. Since 2008, he is a permanent faculty member of University of York's Computer Science department (Lecturer 2008, Senior Lecturer 2013, Reader 2016), and a member of the Real-Time Systems (RTS) research group. His current research interests include on-chip multiprocessor systems, distributed embedded systems, resource allocation, cloud computing, and real-time networks, having published more than 120 peer-reviewed papers in the main international conferences and journals covering those topics (seven of them received best paper awards). He has graduated seven doctoral students, currently supervises three doctoral students and three post-doc research associates. He is a principal investigator of EU-funded SAFIRE project, and a co-investigator in a number of other funded projects. He serves as the department's Internationalisation coordinator, and has held visiting faculty positions in five different countries. He is a member of the EPSRC College, a member of the HiPEAC European Network of Excellence, and a senior member of the IEEE.

Claire Maiza is an assistant professor of Computer Science at Grenoble INP (France) since 2010. She received her PhD degree in computer science from the university of Toulouse, France in 2008. Her research activities are with the VERIMAG laboratory in the "synchrone" group. Her research interests include timing analysis, abstract interpretation, cache analysis, predictable multi-core architecture of real-time systems.

Vincent Nelis received his PhD degree in 2010 at the Computer Science Department of the Université Libre de Bruxelles, Belgium. Since then, Vincent has been working at the CISTER Research Center of Porto, Portugal. Throughout his career, he has published 65+ articles in international journals, conferences, and workshops, received 7 international awards for his work, contributed to 3 R&D projects, led a Work Package in a European FP7 STREP project, chaired 3 international workshops, and he has been member of the program committee of more than 30 international journals, conferences and workshops. His research work has been focused mainly on developing resource allocation techniques (mapping, scheduling, partitioning, and sharing algorithms) for embedded real-time systems and guaranteeing their expected temporal behavior through extensive simulations and timing analyses.

Jan Reineke is an assistant professor of computer science at Saarland University. Previously, he was a postdoctoral scholar at UC Berkeley in the Ptolemy group from 2009 to 2011. He completed his PhD in Computer Science at Saarland University in 2008. His research interests include static analysis by abstract interpretation with applications to the verification of cyber-physical systems, static timing analysis, shape analysis, and side-channel analysis. He is also interested in automatic methods to obtain faithful models of microarchitectures and in the design of timing-predictable microarchitectures for use in hard real-time systems. In 2012, he was selected as an Intel Early Career Faculty Honor Program awardee. He was the program committee co-chair of EMSOFT, the International Conference on Embedded Software, in 2014.