A Generic and Compositional Framework for Multicore Response Time Analysis

Sebastian Altmeyer
University of Luxembourg
University of Amsterdam

Robert I. Davis
University of York
INRIA, Paris-Rocquencourt

Leandro Indrusiak
University of York

Claire Maiza
Grenoble INP Verimag

Vincent Nelis
CISTER, ISEP, Porto

Jan Reineke
Saarland University

ABSTRACT

In this paper, we introduce a *Multicore Response Time Analysis (MRTA) framework*. This framework is extensible to different multicore architectures, with various types and arrangements of local memory, and different arbitration policies for the common interconnects. 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 performance that can be obtained with different hardware configurations. 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.

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 Pre-emption Delays (CRPD) [3], or by partitioning the cache to avoid CRPD altogether. Both approaches are effective and result in tight upper bounds on task response times [4].

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 example, 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 [35]. Similarly, a 14 times slowdown has been reported [38] due to interference on the L2-cache for tasks running on Intel Core 2 Quad processors.

At the other extreme is a *fully integrated* approach. This involves considering the precise interleaving of instructions originating from different cores [22]; 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* [15]. 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, 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 [39]. 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 [35], 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; essential for application areas such as telecommunications, which have a major influence on processor design.

The final approach is the one presented in this paper, based on *explicit interference modelling*. We explore the premise that due to the strong interdependencies between timing analysis and schedulability analysis on multicore systems, they need to be considered together. In our approach, we omit the notion of WCET per se and instead directly target the calculation of task response times.

In this work, we use execution traces to model the behaviour of tasks. Traces provide a simple yet expressive way to model task behaviour. Note that relying on execution traces does not pose a fundamental limitation to our approach as all required performance quantities can also be derived using static analysis [31, 20, 1] as within the traditional context-independent timing analysis; however, traces enable a near-trivial static cache analysis and so allow us to focus on response time analysis.

The main performance metrics are the processor demand and the memory demand of each task. The latter quantity feeds into analysis of the arbitration policy used by the common interconnect, enabling us to upper bound the total memory access delays which may occur during the response time of the 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. In this paper, we instantiate the MRTA framework assuming the local memories used for instructions and data are single-level and either cache, scratchpad, or not present. Further, we assume that the memory bus arbitration policy may be TDMA, FIFO, Round-Robin, or Fixed-Priority (based on task priorities), or Processor-Priority. We also account for the effects of DRAM refresh [5, 11]. The general approach embodied in the MRTA framework is extensible to more complex, multi-level memory hierarchies, and other sources of interference. It 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 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.

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.

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 core of our framework, interference-aware Multicore Response Time Analysis (MRTA). This analysis integrates processor and memory demands accounting for cross-core interference. Extensions to the presented analysis are discussed in Section 7. Section 8 describes the results of an experimental evaluation using the MRTA framework, and Section 9 concludes with a summary and perspectives on future work.

2. RELATED WORK

In 2007, Rosen et al. [40] proposed an implementation in which TDMA slots on the bus are statically allocated to cores. This technique relies on the availability of a user-programmable table-driven bus arbiter, which is typically not available in real hardware, and on knowledge at design time, of the characteristics of the entire workload that executes on each core. Chattopadhyay et al. [17] and Kelter et al. [25] proposed an analysis which takes into account a shared bus and instruction cache, assuming separate buses and memories for both code and data (uncommon in real hardware) and TDMA bus arbitration. The method has a limited applicability as it does not address data accesses to memory.

In 2010, Schranzhofer et al. [42] developed a framework for analysing the worst-case response time of real-time tasks on a multi core with TDMA arbitration. This was followed by work on resource adaptive arbiters [43]. They proposed a task model in which tasks consist of sequences of super-blocks, themselves divided into phases that represent implicit communication (fetching or writing of data from/to memory), computation (processing the data), or both. Contrary to the technique presented here, their approach requires major program intervention and compiler assistance to prefetch data.

Also in 2010, Lv et al. [33] proposed a method to model request patterns and the memory bus using timed automata. Their method handles instruction accesses only and may suffer from state-space explosion when applied to data accesses. A method employing timed automata was proposed by Gustavsson et al. [22] in which the WCET is obtained by proving special predicates through model checking. This approach allows for a detailed system modelling but is also prone to the state-space explosion problem. In 2014, Kelter et al. [26] analysed the maximum bus arbitration delays for multiprocessor systems sharing a TDMA bus and using both (private) L1 and (shared) L2 instruction and data caches.

Pellizzoni et al. [37] 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 non-periodic 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.

Schliecker et al. [41] proposed a method that employs a general event-based model to estimate the maximum load on a shared resource. This approach makes very few assumptions about the task model and is thus quite generally applicable. However, it only supports a single bus arbiter that is an unspecified work-conserving arbiter.

Paolieri et al. [36] proposed a hardware platform that enforces a constant upper bound on the latency of each access to a shared resource. 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. Similarly, the PTARM [32] enforces constant latencies for all instructions, including loads and stores. However, both cases represent customized hardware.

Kim et al. [27] presented a model to upper bound the memory interference delay caused by concurrent accesses to a shared DRAM main memory. Their work differs from this paper in that they do not assume a unique shared bus to access the main memory and they primarily focus on the contention at the DRAM controller by assuming a fully partitioned private and shared cache model. (For shared caches they simply assume that the extra number of requests generated due to cache line evictions at runtime is given).

Yun et al. [46] 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 satisfy schedulability of critical tasks while minimising the performance impact of throttling.

In 2015, Dasari et al. [18] 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. The method of computation in [18] is more complex than that proposed in this paper, and may be more accurate when it estimates the delay due to the shared bus, but it does not take cache-related effects into account (by assuming partitioned caches), which makes it less generic than the framework proposed here.

Regarding shared caches, Yan and Zhang [45] addressed the problem of computing the WCET of tasks 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 (1) data caches are perfect, i.e. all accesses are hits, and (2) data references from different threads will not interfere with each other in the shared L2 cache. Li et al. [30] 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 [17] by adding a TDMA bus analysis technique to bound the memory access delay.

Finally, one must also note that some other techniques, such as [14, 19] for instance, aim at modifying the scheduling algorithm

so that its scheduling decisions reduce the impact of the CRPD.

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. In this paper inevitably 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 $P_1, \dots P_\ell$ as depicted in Figure 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 [23]. 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 processor 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

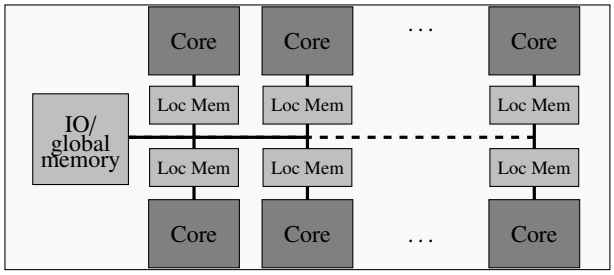

Figure 1: Multicore Platform. A set of ℓ processors with local memories connected via a common bus to a global memory.

omit a consideration of delays due to cache coherence and synchronization, and we assume write-through caches only. Write-back caches are discussed in Section 7.

3.2 Task Model

We assume a set of n sporadic tasks $\{\tau_1, \dots, \tau_n\}$, 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 $\{P_1, \dots, P_\ell\}$, and scheduled on each processor 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 hp(i) (lp(i)) to denote the set of tasks with higher (lower) priority than that of task τ_i, and we use hep(i) (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 Section 7); nevertheless, the tasks compete for hardware resources such as the processor, local memory, and the memory bus.

The execution of task τ_i is modelled using a set of traces O_i, where each trace $o = [\iota_1, \dots \iota_k]$ 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 instructions it:

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

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:

$$\iota = (m^{\text{in}}, \Delta, it) \quad (2)$$

The set of memory blocks is defined as $\mathbb{M}$. $\mathbb{M}^{\text{in}}$ denotes the instruction memory blocks and $\mathbb{M}^{\text{da}}$ the data memory blocks. We assume that data memory and instruction memory are disjoint, i.e, $\mathbb{M}^{\text{in}} \cap \mathbb{M}^{\text{da}} = \emptyset$.

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, 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. However, most importantly, traces show that the worst-case execution behaviour of a task τ_i on a multicore system is *not* uniquely defined. From the viewpoint of a task scheduled on the same core, τ_i may have the highest impact when it uses the core for the longest possible time interval, whereas the impact on tasks scheduled on any other core may be maximized when 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, the complexity can be reduced by (i) computing a synthetic worst-case trace or (ii) by deriving the set of Pareto optimal traces that maximize the task's impact according to a pre-defined cost function (see [31]). We can also completely resort to static analysis to derive upper bounds on the performance metrics. Static analyses provide independent upper bounds on the different performance quantities. This strongly reduces the computational complexity, but may lead to pessimism. An evaluation of this trade-off is future work.

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:

$$\text{MEM}\colon \mathbb{O} \to \mathbb{N} \times 2^{2^{\mathbb{N}}} \times 2^{\mathbb{N}} \quad (3)$$

where $\text{MEM}(o) = (\text{MD}_o, \overline{\text{UCB}}_o, \text{ECB}_o)$ computes, for a trace o, the number of bus accesses i.e., the number of memory accesses which cannot be served by the local memory alone (denoted as memory demand MD), $\overline{\text{UCB}}_o$ which denotes a multiset containing, for each program point in trace o, the set of Useful Cache Blocks (UCBs) [28], which may need to be reloaded when trace o is pre-empted at that program point, and the set of Evicting Cache Blocks (ECBs) which is the set of all cache blocks accessed by trace o which may evict memory blocks of other tasks. The

value 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.

The number of bus accesses MD assumes non-preemptive execution. With pre-emptive execution and caches, more than MD memory accesses can contribute to the bus contention due to cache eviction. In this paper, we make use of the CRPD analysis for fixed priority pre-emptive scheduling introduced by Altmeyer et al. [3].

We now derive instantiations of the function MEM(o) for a trace $o = [\iota_1, \ldots, \iota_k]$ for instruction memories and data memories for systems (i) without cache, (ii) with scratchpads, and (iii) with direct-mapped or LRU caches. In the following, the superscripts indicate data (da) or instruction memory (in), the subscripts the type of memory, i.e., uncached (nc), scratchpad (sp), or caches (ca).

4.1 Uncached

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

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

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

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

4.2 Scratchpads

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 ease of presentation, we assume a static a write-through scratchpad configuration, which does not change at runtime. An extension to dynamic scratchpads and the write-back policy is straight-forward, but beyond the scope of this paper.

Each memory access to a memory block which is not stored in the scratchpad causes an additional bus access.

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

Further, in the case of write accesses, even if a memory block is stored in the scratchpad, that access also contributes to the bus contention as we assume a write-through policy.

$$\mathrm{MEM}^{\mathrm{da}}_{\mathrm{sp}}(o) = \left(\left|\left\{m^{\mathrm{da}} | ((\_, \_, r(m^{\mathrm{da}})) \in o \wedge \neg \mathrm{SPM}(m^{\mathrm{da}})) \vee (\_, \_, w(m^{\mathrm{da}})) \in o\right\}\right|, \emptyset, \emptyset\right) \tag{7}$$

The sets of UCBs and ECBs are empty as no pre-emption overhead is assumed with static scratchpad memory. Dynamic scratchpad management is discussed in Section 7.

4.3 Caches

We assume a function $\mathrm{Hit}\colon \mathbb{I} \times \mathbb{M} \to \{true, false\}$, 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 [20], where each unclassified memory access is considered a cache miss. This means that we upper bound the number of cache misses. For each possible pre-emption point ι on trace o, the set of UCBs is derived using the corresponding analysis described in Altmeyer's thesis [1], Chapter 5, Section 4. It is sufficient to only store the cache sets a useful memory blocks maps to, instead of the useful memory blocks. The multiset $\overline{\mathrm{UCB}}_o$ then contains, for each program point ι in trace o, the set of UCBs at that program point, i.e, $\overline{\mathrm{UCB}}_o = \bigcup_{\iota \in o} \mathrm{UCB}_\iota$. The set of ECBs is the set of all cache sets of memory blocks on trace o.

$$\mathrm{MEM}^{\mathrm{in}}_{\mathrm{ca}}(o) = \left(\left|\left\{m^{\mathrm{in}} | \iota_i = (m^{\mathrm{in}}, \_, \_) \in o \wedge \neg \mathrm{Hit}(m^{\mathrm{in}}, \iota_i)\right\}\right|, \overline{\mathrm{UCB}}^{\mathrm{in}}_o, \mathrm{ECB}^{\mathrm{in}}_o\right) \tag{8}$$

Since we assume a write-through policy, each write access contributes to the cache contention and has to be treated accordingly.

$$\mathrm{MEM}^{\mathrm{da}}_{\mathrm{ca}}(o) = \left(\left|\left\{m^{\mathrm{da}} | (\iota_i = (\_, \_, r(m^{\mathrm{da}})) \in o \wedge \neg \mathrm{Hit}(m^{\mathrm{da}}, \iota_i)) \vee (\_, \_, w(m^{\mathrm{da}})) \in o\right\}\right|, \overline{\mathrm{UCB}}^{\mathrm{da}}_o, \mathrm{ECB}^{\mathrm{da}}_o\right) \tag{9}$$

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

$$\mathrm{MEM}(o) = \left(\mathrm{MD}^{\mathrm{in}}_o + \mathrm{MD}^{\mathrm{da}}_o, \overline{\mathrm{UCB}}^{\mathrm{in}}_o \cup \overline{\mathrm{UCB}}^{\mathrm{da}}_o, \mathrm{ECB}^{\mathrm{in}}_o \cup \mathrm{ECB}^{\mathrm{da}}_o\right) \tag{10}$$

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 $\mathrm{MEM}^{\mathrm{da}}(o) = \left(\mathrm{MD}^{\mathrm{da}}_o, \overline{\mathrm{UCB}}^{\mathrm{da}}_o, \mathrm{ECB}^{\mathrm{da}}_o\right)$ 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:

$$\mathrm{BUS}\colon \mathbb{N} \times \mathbb{P} \times \mathbb{N} \to \mathbb{N} \tag{11}$$

where $\mathrm{BUS}(i, x, t)$ determines an upper bound on the number of bus accesses that can delay task τ_i on processor 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 processor P_x on which it executes. Other task indices are represented by j, k etc. while y, z are used for processor indices.

Let $S^x_i(t)$ denote an upper bound on the total number of bus accesses due to τ_i and all higher priority tasks that run on processor P_x during an interval of length t. Let $A^y_j(t)$ be an upper bound on the total number of bus accesses due to all tasks of priority j or higher executing on some processor $P_y \neq P_x$ during an interval of length t. (Note, j may not necessarily be the priority of a task allocated to processor P_y).

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 processor $P_y \neq P_x$ during an interval of length t. In Section 6 we show how the values of $S_i^x(t)$, $A_j^y(t)$ and $L_j^y(t)$ are computed and explain why $S_i^x(t)$ and $A_j^y(t)$ are subtly different and hence require distinct notation.

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 (i.e. +1 in the equations). This holds because such blocking can only occur at the start of the 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:

$$\text{BUS}(i, x, t) = S_i^x(t) + \sum_{\forall y \neq x} A_i^y(t) + \min\left(S_i^x(t), \sum_{y \neq x} L_i^y(t)\right) + 1 \quad (12)$$

The term $\min\left(S_i^x(t), \sum_{y \neq x} L_i^y(t)\right)$ upper bounds the blocking due to tasks of lower priority than τ_i running on other cores.

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

$$\text{BUS}(i, x, t) = S_i^x(t) + \sum_{y \in HP(x)} A_n^y(t) + \min\left(S_i^x(t), \sum_{y \in LP(x)} A_n^y(t)\right) + 1 \quad (13)$$

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

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

$$\text{BUS}(i, x, t) = S_i^x(t) + \sum_{\forall y \neq x} A_n^y(t) + 1 \quad (14)$$

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

For a Round-Robin bus with a cycle consisting of an equal number of slots v per processor, we have:

$$\text{BUS}(i, x, t) = S_i^x(t) + \sum_{\forall y \neq x} \min(A_n^y(t), v \cdot S_i^x(t)) + 1 \quad (15)$$

The worst-case situation occurs when each access in $S_i^x(t)$ is delayed by each core $P_y \neq P_x$ for v slots. Interference by core P_y is limited to the number of accesses from core P_y. Again, as we already account for all accesses from all other cores, there is no separate contribution to blocking. Note unlike TDMA, Round-Robin moves to the next slot immediately if a processor has no access pending.

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

$$\text{BUS}(i, x, t) = S_i^x(t) + ((\ell - 1) \cdot v) \cdot S_i^x(t) + 1 \quad (16)$$

Since TDMA is not work-conserving, the worst case corresponds to each access in $S_i^x(t)$ just missing a slot for processor P_x and hence having to wait at most $((\ell-1)\cdot v+1)$ slots to be serviced. Effectively, there is additional interference from the $(\ell - 1) \cdot v$ slots reserved for other processors on each access, irrespective of whether these slots are used or not. As all accesses due to higher priority tasks on P_x may be serviced prior to the last access of task τ_i we require $S_i^x(t)$ accesses in total to be serviced for P_x. Note that when $v = 1$, Equation (16) 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 is reserved for future work.

6. RESPONSE TIME ANALYSIS

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

A task set is deemed schedulable, if for each task τ_i, the 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 [6] [24] 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 processor 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 processor.

6.1 Interference on the Core

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

$$\text{PD}_i = \max_{o \in O_i} \sum_{(\_,\Delta,\_) \in o} \Delta \quad (17)$$

where Δ is the execution time of an instruction without memory delays. Task τ_i suffers interference $I^{\text{PROC}}(i, x, t)$ 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:

$$I^{\text{PROC}}(i, x, t) = \sum_{j \in \Gamma_x \wedge j \in \text{hp}(i)} \left\lceil \frac{t}{T_j} \right\rceil \text{PD}_j \quad (18)$$

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

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

maximal memory demand MD_i of a task τ_i is defined by the maximum number of bus accesses of any of its traces:

$$\mathrm{MD}_i = \max_{o \in O_i} \left\{ \mathrm{MD} \middle| \mathrm{MEM}(o) = (\mathrm{MD}, \_, \_) \right\} \tag{19}$$

Note that the maximal memory demand refers to the demand of the combined instruction and data memory as defined in Equation (10).

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,

$$\overline{\mathrm{UCB}}_o = \overline{\mathrm{UCB}} \; \textit{with} \; \mathrm{MEM}(o) = (\_, \overline{\mathrm{UCB}}, \_) \tag{20}$$

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

$$\mathrm{ECB}_i = \bigcup_{o \in O_i} \left\{ \mathrm{ECB} \middle| \mathrm{MEM}(o) = (\_, \_, \mathrm{ECB}) \right\} \tag{21}$$

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 [2] approach as an exemplar of CRPD analysis, as it provides a reasonably precise bound on the pre-emption overhead with low complexity. Other techniques [3] [29] could also be integrated into this framework, but we omit the explanation due to space constraints. The ECB-Union approach considers the UCBs of the pre-empted task per pre-emption point and assumes that the pre-empting task τ_j has itself already been pre-empted by *all* tasks with higher priority on the same processor P_x. This nested pre-emption of the pre-empting task is represented by the union of the ECBs of all tasks with higher or equal priority than task τ_j (see [3] for a detailed description).

$$\gamma_{i,j,x} = \max_{k \in \mathrm{hep}(i) \cap \mathrm{lp}(j) \wedge k \in \Gamma_x} \left(\max_{o \in O_k} \left| \max_{\mathrm{UCB}_\iota \in \overline{\mathrm{UCB}}_o} \left| \left\{ \mathrm{UCB}_\iota \cap \left(\bigcup_{h \in \mathrm{hep}(j) \wedge h \in \Gamma_x} \mathrm{ECB}_h \right) \right\} \right| \right| \right) \tag{22}$$

6.3 Interference on the Bus

We now compute the number of accesses that compete for the bus during a time interval of length t, equating to the worst-case response time of the task of interest τ_i. We use $S_i^x(t)$ to denote an upper bound on the total number of bus accesses that can occur due to tasks running on processor P_x during that 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 Section 5. The maximum delay 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 preemptions occur.

$$S_i^x(t) = \sum_{k \in \Gamma_x \wedge k \in \mathrm{hep}(i)} \left\lceil \frac{t}{T_k} \right\rceil (\mathrm{MD}_k + \gamma_{i,k,x}) \tag{23}$$

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.

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 processor $P_y \neq P_x$ 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 processor P_y. In contrast to the derivation of $S_i^x(t)$, for $A_n^y(t)$ we can make no assumptions about the synchronisation or otherwise of tasks on processor P_y with respect to the release of task τ_i on processor P_x. The value of $A_j^y(t)$ is therefore obtained by assuming for each task, that the first job 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. We assume that the first interfering job of a task τ_k has all of its memory accesses as late as possible during its execution, while for subsequent jobs the opposite is true, with execution and memory accesses occurring as early as possible after release of the job. This treatment is similar to the concept of carry-in interference used in the analysis of global multiprocessor fixed-priority scheduling [10], and is illustrated in Figure 2. The

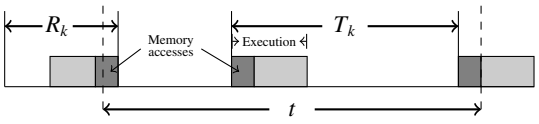

Figure 2: Illustration of the carry-in interference analysis.

number of complete jobs of task τ_k contributing accesses in an interval of length t on processor y is given by:

$$N_{j,k}^y(t) = \left\lfloor \frac{t + R_k - (MD_k + \gamma_{j,k,y}) \cdot d_{\mathrm{main}}}{T_k} \right\rfloor \tag{24}$$

Note the term $(MD_k + \gamma_{j,k,y}) \cdot d_{\mathrm{main}}$ represents the time for the memory accesses. Hence the total number of accesses possible in an interval of length t due to task τ_k and its cache related preemption effects is given by:

$$W_{j,k}^y(t) = N_{j,k}^y(t) \cdot (MD_k + \gamma_{j,k,y}) + min\left(MD_k + \gamma_{j,k,y}, \left\lceil \frac{t + R_k - (MD_k + \gamma_{j,k,y}) \cdot d_{\mathrm{main}} - N_{j,k}^y(t) \cdot T_k}{d_{\mathrm{main}}} \right\rceil\right) \tag{25}$$

Hence we have:

$$A_j^y(t) = \sum_{k \in \Gamma_y \wedge k \in \mathrm{hep}(j)} W_{j,k}^y(t) \tag{26}$$

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

$$L_j^y(t) = \sum_{k \in \Gamma_y \wedge k \in \mathrm{lp}(j)} W_{n,k}^y(t) \tag{27}$$

We note that the carry-in interference has not been accounted for in [27] Equation (5) and (6), resulting in potentially optimistic bounds on the number of competing memory requests in [27].

The number of accesses on the cores are used as input to the BUS function (see Section 5), which we use to derive the maximum bus delay that task τ_i on processor P_x can experience during a time interval of length t,

$$I^{\mathrm{BUS}}(i, x, t) = \mathrm{BUS}(i, x, t) \cdot d_{\mathrm{main}} \tag{28}$$

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

6.4 Global Memory

So far we have assumed a global memory with a constant access latency d_{main}. Global memory is usually realized based on dynamic random-access memory (DRAM), which needs to be refreshed periodically. Now, we show how to relax the constant-latency assumption 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 [34]: *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 burst 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:

$$\mathrm{DRAM}_{\mathrm{burst}}(t, m) = \left\lceil \frac{t}{T_{\mathrm{refresh}}} \right\rceil \cdot \#\mathrm{rows} \tag{29}$$

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.

Under distributed refresh, the upper bound is:

$$\mathrm{DRAM}_{\mathrm{dist}}(t, m) = \min\left(m, \left\lceil \frac{t \cdot \#\mathrm{rows}}{T_{\mathrm{refresh}}} \right\rceil\right) \tag{30}$$

This is the case, 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.

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 of task τ_i on core P_x as follows:

$$I^{\mathrm{DRAM}}(i, x, t) = \mathrm{DRAM}(t, \mathrm{BUS}((i, x, t)) \cdot d_{\mathrm{refresh}} \tag{31}$$

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:

$$R_i = \mathrm{PD}_i + I^{\mathrm{PROC}}(i, x, R_i) + I^{\mathrm{BUS}}(i, x, R_i) + I^{\mathrm{DRAM}}(i, x, R_i) \tag{32}$$

where $I^{\mathrm{PROC}}(i, x, R_i)$ is the interference due to processor demand from higher priority tasks running on the same processor assuming no misses on the local memory (see Equation (18)), $I^{\mathrm{BUS}}(i, x, R_i)$ is the delay due to bus accesses from tasks running on all cores including MD_i (see Equation (28)), and $I^{\mathrm{DRAM}}(i, x, R_i)$ is the delay due to DRAM refreshes (see Equation (31)).

Since the response time of each task can depend on the response times of other tasks via the functions (26) and (27) 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 deal with an apparent circular dependency. Iteration starts with $\forall_i \colon R_i = \mathrm{PD}_i + \mathrm{MD}_i \cdot d_{\mathrm{main}}$ 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 a pseudo-code algorithm 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.

We note that the analysis is sustainable [8] 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.

Algorithm 1 Response Time Computation

```
1:  function MultiCoreRTA
2:      ∀_i : R_i^0 = 0
3:      ∀_i : R_i^1 = PD_i + MD_i · d_main
4:      l = 1
5:      while ∃_i : R_i^l ≠ R_i^{l−1} ∧ ∀_i : R_i^l ≤ D_i do
6:          for all i do
7:              R_i^{l,0} = R_i^{l−1}
8:              R_i^{l,1} = R_i^l
9:              k = 1
10:             while : R_i^{l,k} ≠ R_i^{l,k−1} ∧ R_i^{l,k} ≤ D_i do
11:                 R_i^{l,k+1} = PD_i + I^PROC(i, x, R_i^{l,k})
12:                     +I^BUS(i, x, R_i^{l,k}) + I^DRAM(i, x, R_i^{l,k})
13:                 k = k + 1
14:             end while
15:         end for
16:         R_i^{l+1} = R_i^{l,k}
17:         l = l + 1
18:     end while
19:     if ∀_i : R_i^l ≤ D_i then
20:         return schedulable
21:     else
22:         return not schedulable
23:     end if
24: end function
```

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.

7. EXTENSIONS

Above, we instantiated the Multicore Response Time Analysis (MRTA) framework for relatively simple task and multicore architectural models. In the section, we briefly discuss extensions including: RTOS and interrupts, dynamic scratchpad management, sharing software resources, open systems and incremental verification, write-back cache policies and multi-level caches. However, 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.

7.1 RTOS and Interrupts

The analysis presented in the paper only considers tasks and their execution, as represented by traces. We now give a brief outline of 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.

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 restrictions on space and the wide variety of possible arrangements of interrupt priorities, we do not go into details here). In some cases, interrupts 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 the paper.

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 Dynamic Scratchpad Management

In Section 4.2, we assumed that scratchpad contents were static; however, dynamic scratchpad management schemes [44] are better able to make use of limited scratchpad memory in multitasking systems. In this case pre-emption costs are incurred, saving, loading and restoring the scratchpad contents on each pre-emption. These operations may be explicit, implemented by code in the operating system, in which case the additional processing and memory demands can easily be accounted for via the sub-traces for the RTOS. Alternatively, these operations may be under the control of specialised DMA hardware [44] requiring specific modelling of the additional memory demands.

7.3 Sharing Software Resources

The analysis presented in the paper 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 processor may share software resources that are accessed in mutual exclusion according to the Stack Resource Protocol (SRP) [7]. 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 and its integration into the MRTA framework is beyond the scope of this paper; the basic method is however explained in [3].

We note that blocking due to software resources accessed by tasks on other processors 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 (see section 5 of [3]). 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.

7.4 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 processors loadable post deployment, (ii) the system may be under development and the tasks on another processor not yet known, (iii) incremental verification may be required, so no assumption can be made about the tasks executing on another processor, (iv) the system may be mixed criticality and tasks on another processor 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 processor P_y where we have no information, or need to assume the worst, then 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.

7.5 Caches with a Write-Back Policy

In the paper, we consider write-through caches only; however, in practice *write-back caches* are usually preferred, as they reduce the number of accesses to main memory, and thus increase performance. Write-back caches introduce three challenges for future work: The first challenge is to devise analyses that precisely bound the number of write backs, which is equal to the number of evictions of dirty cache lines. The second and perhaps greater challenge is that write backs corresponding to the execution of a task τ_i may occur after the termination of τ_i and thus contribute to the delay of another task. Thirdly, write-back caches require the implementation of coherence protocols, which may generate additional traffic on the memory bus, which would have to be safely bounded. A naive solution to the first two challenges assumes pessimistically that each cache line is dirty and thus each cache eviction leads to two bus accesses. Alternatively, we can derive for each task in a closed system a set of dirty-cache lines, which have to be written back if evicted by another task. Write-backs can then be considered an additional source of interference in the framework. Details analysis for write-back caches remains an interesting area for future work.

7.6 Multi-level Caches

Modern multicore processors often feature *multiple cache levels*, where usually one level is shared between multiple cores. Dealing with such a scenario in our framework is in principle feasible. As long as all caches are *private*, the challenge would be to integrate an extension of CRPD analysis to multiple cache

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

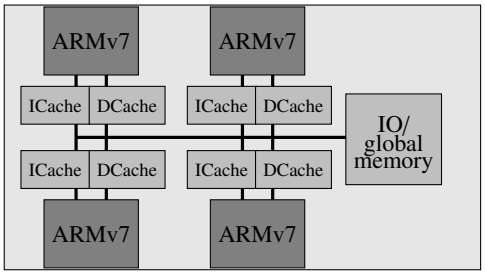

Figure 3: Multicore Architecture Case Study: $m = 4$ cores with local caches connected via a common bus to a global memory.

levels. Chattopadhyay and Roychoudhury [16], have recently proposed such an analysis for non-inclusive memory hierarchies. *Shared* second- or third-level caches add the extra complication of cross-core interference on the cache. Different more or less precise and efficient approaches to bound this interference are conceivable, and again form an interesting area for future work.

8. EXPERIMENTAL EVALUATION

In this section we describe the results of an experimental evaluation using the MRTA framework [3] . For the evaluation, we use the Mälardalen benchmark suite [21] to provide traces. We model a multicore systems based on an ARM Cortex A5 multicore[4] as a reference architecture to provide a cache configuration and memory and bus latencies. As this work is intended to provide an overview of our generic and extensible framework, we do not model all details of the specific multicore architecture. A case study comparing measurements on a real hardware with the computed bounds is future work.

The reference architecture depicted in Figure 3 is configured as follows: It has 4 ARMv7 cores connected to the global memory/IO over a shared bus assuming a round-robin arbitration policy and a core frequency of 200MHz. Each core has separate instruction and data caches, with 256 cache sets each and a block size of 32Bytes. 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 Section 6.4).

We examine derivatives of the reference configuration assuming the different bus arbitration policies presented in Section 5 and a hypothetical **perfect bus** which eliminates all bus interference if the bus utilization is ≤ 1. We compare the reference configuration with two alternatives architectures: The first, referred to as **full-isolation architecture** 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 in the reference architecture. The performance on the isolation architecture corresponds to the traditional two-step approach to timing verification with context-independent WCETs. The second alternative, referred to as **uncached architecture**, assumes no local caches except for a buffer of size 1, and uses round-robin bus arbitration. All other parameters are again the same as the reference configuration. The traces for the benchmarks were generated using the gem5 instruction set simulator [13] and contain statically linked library calls. As the benchmark code corresponds to independent tasks, no data is shared between the tasks. Table 1 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 or worst-case input is provided. 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 of more complex tasks including evaluation of the trade off between pessimism of independent upper bounds and the computational complexity of explicit traces remains as future work.

We identify three main sources of over-approximation of our multicore response time analysis framework: The number of memory accesses on the same core cannot be precisely estimated due to imprecison 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 too frequently if the number of main memory accesses is over-approximated. A sophisticated evaluation of the precision of our analysis requires measurements on a real architecture, which we cannot yet provide. However, the different architecture configurations provide an estimate of the influence of the different sources of pessimism. The reference architecture with a perfect bus eliminates any pessimism due to bus interference and DRAM accesses. Only the pessimism of the pre-emption cost analysis remains, which has been quantified in [2]. The full-isolation architecture removes all pessimism due to the bus interference and the pre-emption costs, and thus only suffers from the pessimism in the DRAM analysis.

We evaluated the guaranteed performance of the various 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 1.
- The base WCET per task τ_i, needed solely to set the task periods and deadline, was defined as

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

  C_i denotes the execution time of the task without any interference from any other task.
- The task utilizations were generated using UUnifast [12] 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.

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

$$U^{\text{BUS}} = \sum_i \frac{\text{MD}_i \cdot d_{\text{main}}}{T_i} \tag{33}$$

is the limiting factor. Only if $U^{\text{BUS}} \leq 1$, can the tasks be scheduled.

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

Figure 4 shows the number of schedulable task sets plotted against the core utilization (computed using the base WCETs) and Figure 5 against the bus utilization U^{BUS}. Most traces from Table 1 have a high memory demand, which results in a high

[3]The software is available on demand.

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

Name	# Instr. (PD)	Read/Write	MD	UCB	ECB
adpcm_dec	627553	123641	38575	144	332
adpcm_enc	628795	124168	38729	155	346
binarysearch	678	293	229	20	118
bsort100	272715	1305613	25464	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	34860	7861	2240	74	181
cover	3661	1495	696	19	231
crc	67359	20452	6656	44	162
duff	3121	1484	553	24	130
edn	164596	73857	15383	104	306
expint	8058	2221	716	27	118
fac	1096	411	274	17	108
fdct	5923	3098	1088	67	193
fft1	92289	11229	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	3023813	373874	120821	150	276
loop3	10539	4412	1820	16	351
ludcmp	8278	3004	768	59	189
matmult	384140	78058	11923	123	272
minver	16256	3627	1437	121	284
ndes	107957	50632	13186	96	252
nsichneu	8648	4841	1582	397	589
ns	25494	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	26167	3185	1438	62	151
statemate	62188	51792	13360	117	235
st	1498482	125946	31969	341	429

Table 1: Benchmark traces

number of bus accesses even at low core utilizations. Consequently, most 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 (dark blue line) and then TDMA (pink line). The full-isolation architecture (light blue) implementing TDMA and cache partitioning on the local caches performs nearly as well as the TDMA architecture, which indicates that the increased execution times due to cache partitioning only have a minor impact in this case. Note for TDMA and Round-Robin, we assume a cycle with 2 slots per processor.

The FIFO bus shows the lowest performance, similar to that of an uncached architecture, which uses round-robin. The worst-case arrival pattern for a FIFO bus (black 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 (dark green line) only assumes that co-running tasks assigned to a processor of higher priority have issued requests, 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 processor with higher priority.

The difference between the Fixed-Priority and

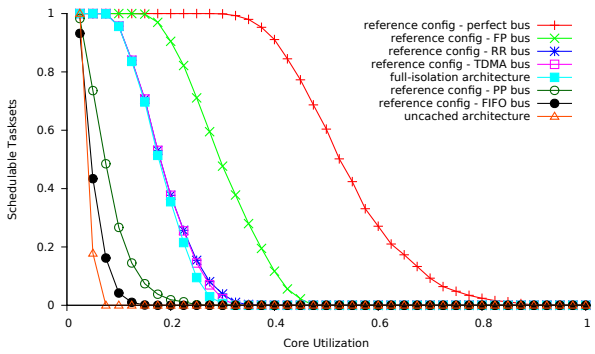

Figure 4: Number of schedulable task sets vs. core utilization

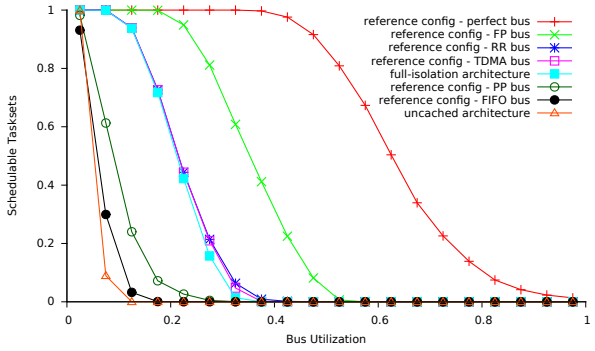

Figure 5: Number of schedulable task sets vs. bus utilization

Round-Robin/TDMA shows the MRTA framework is able to guarantee good performance even if the bus policy does not provide a tightly bounded bus latency for single accesses (as is the case for TDMA and Round-Robin).

Figures 4 and 5 only show the results for different bus policies and three cache configuration (uncached, partitioned and unconstrained cache usage). In the following, 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 [9], to show how schedulability varies with these parameters.

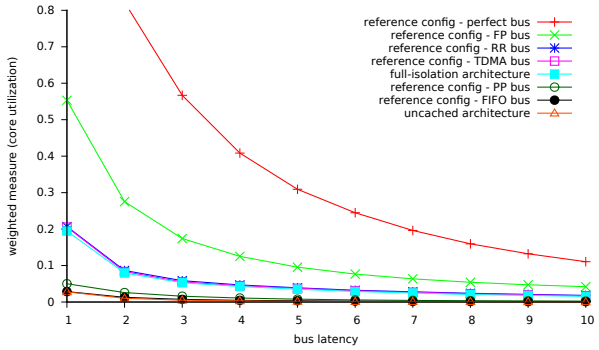

Figure 6: Weighted schedulability; varying bus latency.

As the memory demand of the benchmark traces is high, the bus latency d_{main} has a tremendous impact on overall schedulability (see Figure 6). The bus latency affects all bus policies similarly.

By increasing the number of cores, the number of tasks also

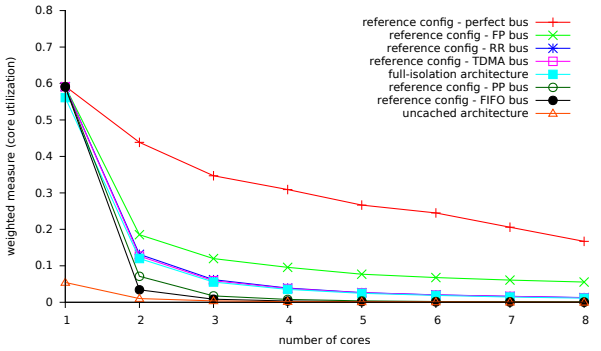

Figure 7: Weighted schedulability; varying number of cores.

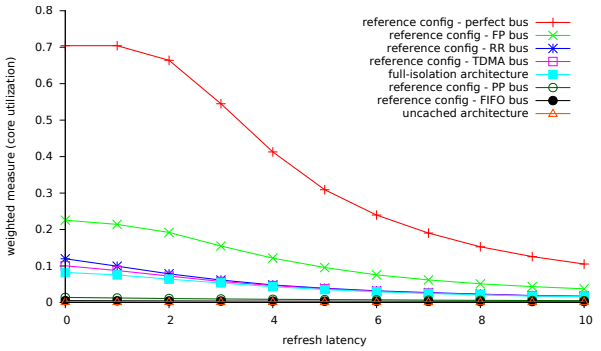

Figure 8: Weighted schedulability; varying DRAM refresh latency.

increases (assuming a fixed number of tasks per core) and so does the bus utilization. The performance of all configurations decreases (see Figure 7) as fewer task sets are deemed schedulable, irrespective of the bus policy.

As might be expected, longer DRAM refresh latencies have a significant detrimental effect on schedulability for all configurations, see Figure 8.

9. CONCLUSIONS

In this paper, we introduced a *Multicore Response Time Analysis (MRTA) framework*. 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 initial 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 provides a general approach to timing verification for multicore systems that is parametric in the hardware configuration (common interconnect, local memories, number of cores etc.) and so can be used both at the architectural design stage to compare the guaranteed levels of performance obtained with different hardware configurations, and also during development to verify the timing behaviour of a specific system.

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.

Our evaluation used the MRTA framework to model and analyse a generic multicore processor based on information about the ARM Cortex A5, with software from the Mälardalen benchmark suite used as code for the tasks in our case study. Our results show that while a full-isolation 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 policies such as the Fixed-Priority bus arbitration based on task priorities. Section 7 discusses a variety of ways in which the framework can be extended. In future we aim to explore these avenues, extending our work by instantiating the analysis for more complex behaviours and architectures, as well as to global and semi-partitioned scheduling policies. We also plan to run detailed (cycle accurate) simulations of the multicore architectures to examine the effectiveness of the MRTA framework compared to observed behaviour.

Acknowledgements

This work was supported in part by the COST Action IC1202 TACLe, by the DFG as part of the Transregional Collaborative Research Centre SFB/TR 14 (AVACS), 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 nr. 621429 (EMC2), by the INRIA International Chair program, and by the EPSRC project MCC (EP/K011626/1). 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`.

References

[1] S. Altmeyer. *Analysis of Preemptively Scheduled Hard Real-time Systems*. epubli GmbH, 2013.

[2] S. Altmeyer, R. I. Davis, and C. Maiza. Cache related pre-emption aware response time analysis for fixed priority pre-emptive systems. In *RTSS*, pages 261–271, December 2011.

[3] S. Altmeyer, R. I. Davis, and C. Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.

[4] S. Altmeyer, R. Douma, W. Lunniss, and R.I. Davis. Evaluation of cache partitioning for hard real-time systems. In *ECRTS*, pages 15–26, July 2014.

[5] P. Atanassov and P. Puschner. Impact of DRAM refresh on the execution time of real-time tasks. In *IEEE International Workshop on Application of Reliable Computing and Communication*, pages 29–34, December 2001.

[6] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[7] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3:67–99, April 1991.

[8] S. Baruah and A. Burns. Sustainable scheduling analysis. In *RTSS*, pages 159–168, December 2006.

[9] A. Bastoni, B. Brandenburg, and J. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *OSPERT*, pages 33–44, July 2010.

[10] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *RTSS*, pages 149–160, 2007.

[11] B. Bhat and F. Mueller. Making DRAM refresh predictable. *Real-Time Systems*, 47(5):430–453, September 2011.

[12] E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30:129–154, 2005.

[13] N. Binkert et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[14] R. J. Bril, S. Altmeyer, M. M. H. P. van den Heuvel, R.I. Davis, and M. Behnam. Integrating cache-related pre-emption delays into analysis of fixed priority scheduling with pre-emption thresholds. In *RTSS*, pages 161–172, 2014.

[15] D. Bui, E. Lee, I. Liu, H. Patel, and J. Reineke. Temporal isolation on multiprocessing architectures. In *DAC*, pages 274–279, June 2011.

[16] S. Chattopadhyay and A. Roychoudhury. Cache-related preemption delay analysis for multilevel noninclusive caches. *ACM TECS*, 13(5s):147:1–147:29, July 2014.

[17] S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *SCOPES*, pages 6:1–6:10, June 2010.

[18] D. Dasari, V. Nelis, and B. Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Systems*, pages 1–51, 2015.

[19] R. I. Davis, A. Burns, Jose Marinho, V. Nelis, S. M. Petters, and M. Bertogna. Global and partitioned multiprocessor fixed priority scheduling with deferred preemption. *ACM TECS*, 14(3):47:1–47:28, April 2015.

[20] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2-3):163–189, 1999.

[21] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In *WCET*, pages 137–147, July 2010.

[22] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In *WCET*, pages 101–112, Dagstuhl, Germany, July 2010.

[23] S. Hahn, J. Reineke, and Wilhelm R. Towards compositionality in execution time analysis – definition and challenges. In *CRTS*, December 2013.

[24] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, May 1986.

[25] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore WCET analysis through TDMA offset bounds. In *ECRTS*, pages 3–12, July 2011.

[26] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Static analysis of multi-core TDMA resource arbitration delays. *Real-Time Systems Journal*, 50(2):185–229, 2014.

[27] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *RTAS*, 2014.

[28] C.-G. Lee, J. Hahn, Y.-M. Seo, S.L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[29] C.G. Lee, K. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE TSE*, 27(9):805–826, 2001.

[30] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS*, pages 57–67, December 2009.

[31] Yau-Tsun S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, pages 456–461, June 1995.

[32] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *ICCD*, September 2012.

[33] M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *RTSS*, pages 339–349, December 2010.

[34] Micron Technologies, Inc. Various methods of DRAM refresh. Technical report, 1999.

[35] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS*, pages 109–118, July 2014.

[36] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. *SIGARCH Comput. Archit. News*, 37(3):57–68, June 2009.

[37] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *DATE*, pages 741–746, March 2010.

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

[39] J. Reineke and J. Doerfert. Architecture-parametric timing analysis. In *RTAS*, pages 189–200, April 2014.

[40] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*, pages 49–60, Dec. 2007.

[41] S. Schliecker, M. Negrean, and R. Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *DAC*, pages 759–764, June 2010.

[42] A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *RTAS*, pages 215–224, April 2010.

[43] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *RTAS*, pages 213–222, April 2011.

[44] J. Whitham, R.I. Davis, N.C. Audsley, S. Altmeyer, and C. Maiza. Investigation of scratchpad memory for preemptive multitasking. In *RTSS*, pages 3–13, December 2012.

[45] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *RTAS*, pages 80–89, 2008.

[46] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *ECRTS*, pages 299–308, 2012.