Noname manuscript No.
(will be inserted by the editor)

A Formal Approach to the WCRT Analysis of Multicore Systems with Memory Contention under Phase-structured Task Sets

Kai Lampka · Georgia Giannopoulou · Rodolfo Pellizzoni · Zheng Wu · Nikolay Stoimenov

Received: date / Accepted: date

Abstract Multicore technology has the potential for drastically increasing productivity of embedded real-time computing. However, joint use of hardware, e.g., caches, memory banks and on-chip buses makes the integration of multiple real-time applications into a single system difficult: resource accesses are exclusive and need to be sequenced. Moreover, resource access schemes of modern off-the-shelf multicore chips are commonly optimized for the average-case, rather than being timing predictable. Real-time analysis for such architectures is complex, as execution times depend on the deployed hardware, as well as on the software executing on other cores. This will ask for significant abstractions in the timing analysis, where the resulting pessimism will lead to over-provisioned system designs and a lowered productivity as the number of applications to be put together into a single architecture needs to be decreased. In response to this, (a) we present 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 shared resource, (b) we present a simulation framework which allows for detailed modeling and empirical evaluation of modern multicore platforms and applications running on top of them, and (c) we present experiments to demonstrate the advantages and disadvantages of the presented methodologies and compare their accuracy. For limiting non-determinism inherent to the occurrence of cache misses, we particularly take advantage from the PRedictable Execution Model (PREM) as discussed in recent works.

Kai Lampka
Department of Information Technology, Uppsala University (Sweden),
E-mail: kai.lampka@it.uu.se

Georgia Giannopoulou · Nikolay Stoimenov
Computer Engineering and Communication Networks Lab, ETH Zurich (Switzerland)
E-mail: {giannopoulou,stoimenov}@tik.ee.ethz.ch

Department of Electrical and Computer Engineering, University of Waterloo (Canada)
E-mail: {rpellizz,zpwu}@uwaterloo.ca

1 Introduction

1.1 Motivation

These days, distributed computing is arriving at chip-level and electronic control units are becoming increasingly multicored. This lures the integration of multiple embedded (control) applications into a single processor, as their mutual independence feigns the possibility of unlimited parallelism. However, race conditions which are imposed by the usage of shared resources such as caches, memory banks or buses on chip can easily become a threat to the timing correctness of the integrated real-time applications. As an example to this, one may think of a multicore system where at least 2 cores use the same Level 2 (L2) cache. The software executing on the different cores may mutually over-write some of their cache entries. This in turn, will significantly add to their execution times as code segments and data items must be (re-)fetched from the main memory. However, this interference does not stop at the level of caches. When fetching items from the main memory, cores need to wait for getting access to the memory. As the Dynamic Memory Access (DMA) controller might implement complex access patterns, on top of the non-deterministically arriving access requests, waiting times may also drastically add to the execution time of the software. For these reasons, it might be the case that a (control) software misses its assumed deadline and a control signal does not reach the actor in time. This in turn can harm the overall system's stability and provoke its damage or complete loss, not to mention human casualties.

In such scenarios the challenge is to tightly bound response times of applications, i. e. bound the time it takes the software to complete its (control loop) computation. Obtaining precise bounds for each application is, however, far from trivial: waiting times induced by the use of shared hardware are difficult to predict, since the arbitration schemes for coordinating access to the shared hardware can be arbitrarily complex, with a design meant to maximize overall throughput rather than serving the real-time applications in a predictable way, cf. [27] for an example.

A common principle to resolve this issue is known as isolation. Isolation mechanisms eliminate or strictly limit the interference among the competing applications, and allow for simplified and more accurate analysis methods of their *worst-case response times* (WCRT). Examples of such strategies are memory partitioning schemes, e.g. cache or memory colouring schemes [14] or time triggered schemes for partitioning the (bus) access times [34]. Achieving timing predictability through isolation mechanisms can become, however, very wasteful and might not always be possible, e.g., hardware details are not exposed or modifiable. In such situation state-based real-time modeling and exhaustive analysis techniques, appear to be useful, as they allow the modeling of almost arbitrarily complex system behaviours. Contrary to analytic methods, these techniques may not severely over-approximate a system's behaviour, as their operational semantics often reflect the modus operandi of the actual system. But there is no free lunch as state-based modeling and analysis require explicit expansion of all possible system behaviours, which in turn may require to produce an enormous number of system states to be analyzed. As their number can be exponential in the number of states of the concurrent applications, state-based modeling

and analysis techniques may not be applicable to systems of industrial size. In this respect, analytic methods are better as they provide high scalability. But, on the downside, they sacrifice accuracy of the results by making conservative assumptions about the system. This article will present an analysis methodology which aims at exploiting advantages of both worlds, tailored to the modeling and analysis of multicore control systems, where resource contention among concurrently executing real-time applications can not be avoided.
For limiting the non-determinism inherent to the occurrence of cache misses in time, we exploit the PRedictable Execution Model (PREM) as discussed in recent works.

1.2 Contribution

To balance computational effort and precision of results, past research has focused on development of heterogeneous analysis methods, i. e. combinations of analytic and state-based modeling and analysis techniques. This article exploits this line of research, however, aims at achieving scalability and effectively analyzing multicore systems with resource contention.
Exploiting a state-based modeling approach for bounding the WCRT of tasks deployed on multicore systems with shared main memory is motivated by the aim of decreasing the inherent pessimism of previously published analytic methods.
Along this line of thought, this paper presents the following contributions:

- A worst-case response time method based on Timed Automata which exploits the so-called superblock model of the work of Schranzhofer et al. [31,36,37], respectively the PRedictable Execution Model (PREM) of Pellizzoni et al. [29, 40,6]. This limits the time non-determinism inherent to the occurrence of cache misses, respectively memory (data) fetches.
- A worst-case response time method that replaces some of the Timed Automata models from the first approach with an abstract representation based on access request arrival curves.
- Experimental results which show the advantages and disadvantages of the two analysis methods and compare their accuracy when applied to six benchmarks from the AutoBench group of the EEMBC (Embedded Microprocessor Benchmark Consortium) [1].
- We also show results based on a simulation framework to validate system settings and to provide a lower bound on worst-case response time.

The results in this article extend our previous work [13] as it introduces (a) a novel approach for deriving access request curves which simplifies the derivation and increases the accuracy of the analysis results, (b) a novel simulation framework which allows for validation of the analytic results, and (c) a benchmarking of the formal and simulation-based analysis in order to compare their trade-offs.

1.3 Organization

The remainder of this article is organized as follows: Sec. 2 presents the background theory, i. e. , Real-time Calculus and Timed Automata. Sec. 3 - 5 presents our abstract system model and introduces further abstractions for achieving scalability in the analysis without introducing too much pessimism. Sec. 6 introduces the formal modeling of the system on top of Timed Automata, and details on steps for speeding up the analysis. Sec. 7 presents the simulation framework and the findings of the formal analysis and the simulation-based system evaluation. Sec. 8 concludes the article.

1.4 Related work

Performing timing analysis for multicore systems with shared resources is challenging as the number of possible orderings of access requests which arrive at a shared resource can be exponentially large. Additionally, resource accesses can be asynchronous such as message passing or synchronous such as memory accesses due to cache misses. For the asynchronous accesses, the timing analysis needs to take into account the arrival pattern of accesses from the processing cores and the respective backlogs. In this case, traditional timing analysis methods such as Real-Time Calculus [39] and SymTA/S [16] can compute accurate bounds on the worst-case response times (WCRT) of tasks and the end-to-end delays of accesses. For the synchronous case, however, an access request stalls the execution until the access is completed, i. e. an access increases the tasks' WCRT. This is because, once an access request is released, the task execution can not be preempted. Moreover, once service of an access request starts, the latter can also not be preempted by other accesses. Bounding the waiting times of tasks under these assumptions is far from trivial as one has to take into account the currently issued accesses from other cores and the state of the resource sharing arbiter. This paper is concerned with the second setting, i. e. the case of synchronous accesses.

Schliecker et al. [35] have recently proposed methodologies to analyze the worst-case delay a task suffers due to accesses to shared memory, assuming synchronous resource accesses. The authors consider a system where the minimum and maximum number of resource accesses in particular time windows are assumed to be known. The arbiter to the shared resource uses a first-come first-serve (FCFS) scheduling scheme and tasks are scheduled according to fixed priority. The authors evaluate the approach with a system with few processing cores. Shortcomings of this method, which lead to increased pessimism, are identified and addressed in the work of Dasari et al. [12], which by following a similar approach, provides tighter bounds on the worst-case delays that a task may experience due to interference on the shared bus to the memory.

An alternative approach to analyzing the duration of accesses to a TDMA-arbitrated resource is presented by Kelter et al. [18]. The proposed techniques is based on abstract interpretation and integer linear programming (ILP). It statically computes all possible offsets for access request within a TDMA arbitration cycle. This way the authors aim at high precision of analysis results and keeping analysis times reasonable.

As major shortcoming, all of the above works, do not consider complex scheduling policies or systems with high number of processing cores. This shortcoming is the major driver for the line of work presented in this article.

Schranzhofer et al. [31,36,37] have proposed methods for the WCRT analysis in multicore systems where the shared resource is arbitrated according to FCFS, round-robin (RR), TDMA or a hybrid time-/event-driven strategy, the latter being a combination of TDMA and FCFS/RR. Contrary to previous works, the proposed methodology is shown to scale efficiently. The analysis, however, uses over-approximations which sometimes result in extremely pessimistic results, particularly in cases of state-based arbitration mechanisms, like FCFS or RR. This shortcoming is of concern, as modern multicore architectures tend to exploit complex, state-dependent behaviours. The presented work exploits the same model of computation as Schranzhofer et al. [31,36,37]. Moreover, it makes use of the concept of event arrival curves for capturing the non-deterministic arrivals of service requests at the shared resource, similarly as in the above works. The main difference lies in that our work exploits a state-based, real-time modeling and analysis formalism. This way we ensure that contrary to the earlier work of Schranzhofer, our modeling and analysis captures accurately the behaviour of the state-dependent resource arbiters.

The presented analysis methodology is based on a combination of state-based and analytic analysis technique. Such techniques have been introduced in [20,21] for the combination of Timed Automata and RTC. The authors of [3] handle the case of synchronous data-flow component models and RTC and [38] introduces the coupling of parametric Timed Automata and RTC on top of a SMT-based analysis technique for deriving regions of parameters for task activation patterns under which the system is scheduled.
Our analysis for real-time tasks in a multicore setting is based on the aforementioned coupling of Timed Automata (TA) [4] and RTC [20,21], where we exploit specific concepts as implemented in the timed model checker Uppaal [7,8].
Yi et al. [28] were the first to use TA to represent a system resource (CPU or communication element) as a scheduler model together with a notion of discrete events that trigger the execution of real-time tasks on this scheduler. In this work they also show that the feasibility problem of this extended model can be transformed into a reachability problem for a standard TA, which allows one to check whether a system is feasible for some fixed set of parameters, e. g. buffer sizes, delays, execution times etc.
In the recent work [25] the basic idea has been extended for analyzing multicore architectures and different resource arbitration policies. A similar approach has been also followed by Gustavsson et al. in [15]. However, due to the state space explosion problem, the illustrated techniques in both publications are limited and can only be applied for few tasks and up to two cores. We overcome this by combining TA with an analytic abstraction, specifically event curves, for the modeling and analysis of multicore systems and state-dependent arbiters for coordinating the access to a shared resource.

2 Background theory

2.1 Preliminaries

When analyzing and designing embedded systems with hard real-time constraints, contemporary methods abstract over individual events and their individual processing. Streams of signals, together with their processing software, denoted as tasks are the main ingredients of an abstract system understanding, which in the past decade has successfully served for establishing formal analysis methods in the design cycle of embedded real-time systems. This abstract system understanding which will be decisive for this paper can be summarized as follows.
Tasks are sharing computing devices and are communicating via (non-blocking) FIFO-buffers of infinite capacity. A task consumes streams of uniform input signals from the environment or some up-streamed set of tasks. As output, it emits streams of uniform output signals, either to the environment or some down-streamed tasks. Each task is statically mapped to a hardware unit. Consequently, each hardware entity like a communication bus or an external cache is abstractly modeled by a dedicated task.
Signal or event processing of a task is assumed to be organized in a first-in-first-out manner as far as events from the same stream are concerned, i.e., events of the same stream can neither overtake nor preempt each other. Please note that splitting and merging of streams is possible. Processing the demand imposed by signals of other types, i.e., signals processed by other tasks, but which are mapped to the same device, is resolved according to some standard (scheduling) policy, e. g. earliest deadline first (EDF), priority-based, Round-Robin (RR), First-In-First-Out (FIFO), Time-Division-Multiple Access (TDMA) etc.
The maximal computation time, e.g., number of processor cycles, consumed by a task for processing any of its input stimuli, is denoted as *worst-case execution* time (WCET). Computing WCET is out of the scope of this paper, commercial products like aiT [17] can be exploited for obtaining the core-local processing demand requested by a task.

As one may have noted, the above definition solely considers the time a task is active. However, in practice tasks often do not make progress with their computations as they are waiting for input stimuli from the environment, e.g., some data item to be fetched from the main memory. This induces waiting times which heavily depend on the environment, e.g., the behaviour of other tasks executing on different devices, but also racing for the access to the same resource, e.g., the main memory. The tight bounding of the waiting times of a task is the major issue of this work, as in case of complex environments this is far from trivial. The maximal waiting time of a task, together with its WCET yields its *worst-case response time* (WCRT). The WCRT ultimately bounds any delay experienced by a signal processed by the task under analysis. As long as it is below some threshold value, commonly denoted as deadline, the system is said to be schedulable.

2.2 Streams and their abstract representation

In this work we will exploit the concept of streams and arrival curves for abstractly capturing the service requests sent to a shared resource.
A timed event is a pair (t, τ) where τ is some event label or type and $t \in \mathbb{R}_{\geq 0}$ some non-negative time stamp. A timed trace $tr := (t_1, \tau_a); (t_2, \tau_b); \ldots$ is a sequence of timed events ordered by increasing time stamps, s. t. $t_i \leq t_{i+1}$ for $i \in \mathbb{N}$ and $\tau_a, \tau_b \in Act$ with Act as a set of event labels or signal types. A possible infinite set of traces referring to the same signal e is commonly denoted as (event) stream.
An arrival curve α is a pair of upper and lower curves $(\alpha^{low}, \alpha^{up})$. It provides a generic characterization of an event stream as follows [23,39]: Let $R(t)$ denote the number of events that arrived on the stream in the time interval $[0, t)$, then an upper and lower arrival curve satisfy the following equation

$$\alpha^{low}(t-s) \leq R(t) - R(s) \leq \alpha^{up}(t-s) \tag{1}$$

for all $0 \leq s \leq t$, where in the following we use the notation $\Delta \in [0, \infty)$ for addressing the time intervals $t - s$.
As each event from a stream may trigger behaviour within a down-streamed task, arrival curves provide abstract lower and upper bounding functions α^{low} and α^{up} w. r. t. the resource demand experienced within time interval Δ. Furthermore, one may note that for a given pair α^{low} and α^{up} there might be a (possibly infinite) set of traces of computation stimuli, namely all traces the counting function of which satisfies Eq. 1.
Let $\alpha_i := (\alpha_i^{up}, \alpha_i^{low})$ and $\alpha_j := (\alpha_j^{up}, \alpha_j^{low})$ be two event-compatible arrival curves. If $\alpha_i^{up}(\Delta) \leq \alpha_j^{up}(\Delta)$ and $\alpha_i^{low}(\Delta) \geq \alpha_j^{low}(\Delta)$ holds for all $\Delta \in [0, \infty)$ one says α_i is bounded by or included in α_j and writes $\alpha_i \subseteq \alpha_j$. As pointed out above, each arrival curve presents a potentially infinite number of traces. Hence $\alpha_i \subseteq \alpha_j$ implies that α_i is over-approximated by α_j, as all timed traces bounded by α_i are also included in the set of traces represented by α_j. This is a crucial as it allows one to replace a complex arrival curve α_i by a less complex curve α_j, but still guaranteeing the validity of the deduced WCRT. Details on this feature can be found in [11,21].

2.3 State-based modeling and analysis of real-time systems

The complex behavior of arbiters coordinating the access to a shared resource calls for modeling formalisms which are capable of capturing state-based behavior. For convenience, this work employs TA [4] and the related model checking techniques as implemented in tools such as Uppaal [7,8] or Kronos [10].

2.3.1 Timed automata (TA): basic concept

A TA [4] is a state machine extended with clocks. Instead of states one commonly speaks of locations and instead of transitions one uses the term edges, which is useful for separating the syntactic from the semantic level.
The location the TA currently resides in is denoted as active location. As main feature

clocks can either be inspected or reset, where inspection takes place when checking the validity of clock constraint associated with locations, denoted as location invariants or when checking the Boolean guards of edges. A clock constraint g_c is of the following kind: $g_c := x \bowtie k | x - y \bowtie k | g_c \wedge g_c | \neg g_c$ with x and y as clocks, $k \in \mathbb{Q}$ as a clock constant and the binary operator $\bowtie \in \{<, \leq, >, \geq, =\}$.
The basic construction can be extended with variables, where variables like clocks can be inspected at any time and their values are changed upon edge traversals. The used variable values must build a finite set, which, however, does not need to be known on beforehand. The set can be constructed on-the-fly, i. e. during the state space traversal.
An example of a network of interacting TA is depicted in Fig. 4.

We characterize the operational (or execution) semantic of TA informally as follows:

While a TA resides in a location, all clocks increase at the same speed and the location invariants need to hold. Exceptions to this are so called urgent and committed locations, marked with a **u**, **c** respectively. In these locations time is not allowed to progress, i.e., urgent or committed locations have to be left instantaneously, in zero time respectively.

Discrete event: an edge is enabled as soon as its (Boolean) guard evaluates to true. An enabled edge emanating from the currently active location can be traversed (or executed) which yields possibly a new active location. Clock resets are executed when the enabled edge is traversed. The traversal of the edge is instantaneous, i. e. it takes zero time.

The operational semantic as defined above allows one to derive an infinite state-to-state transition system, where transitions either refer to the traversal of edge(s) or to the progress of time. As clocks only evaluate to values from finitely many different intervals, induced by the employed clock constants, and given that variables only take values from a finite domain, the infinite state-to-state transition system of a TA can be characterized by a finite graph. This finite representative which is known as region graph is a symbolic representation of all possible behaviours of a TA. Hence validity of timed and un-timed properties associated with system states or sequence of states is decidable for TA [4]. One may note that modern timed model checkers incorporate clock zones as they often result in a coarser partitioning of the clock evaluation space in comparison to clock regions.

2.3.2 Hierarchic modeling: networks of TA

One may compose a set of TA into a *network* of cooperating TA. In such a setting, clocks and variables can be shared among the different TAs, and dedicated sending and receiving edges are jointly traversed which depend on the used synchronization policy.
A state of a network of cooperating TA $\mathcal{T}$ is defined by the active locations, the active clock regions and the values of the variables. With active locations we refer to the locations the different (component) TA currently reside in, and with active clock regions we refer to the set of constraints which contain the current clock evaluations.

In the following we generically speak of TA, but referring to networks of TA as specified as input to the timed model checker Uppaal. We briefly re-capitulate some composition mechanisms which defines the interaction semantics of the (component) TA put together into a network.
Edges are commonly labelled, and the labels are denoted as signals. The notation $a!$ refers to the sending of a corresponding a signal and the notation $a?$ refers to the reception of a signal a. The sending and receiving of a signal takes place once the respective edges of a sending TA and a receiving TA are executed, i. e. the sender and receiver are different TA. The step of sending and receiving is atomic, i. e. synchronous, and instantaneous. As a result, execution of an executable sending edge can take place if and only if a receiving edge labelled with the corresponding signal can be executed as well. An exception to this is the broadcasting of a signal which will be discussed below. Likewise, a receiving-edge of a TA can only be executed if a corresponding sending edge of another component TA is executable as well. Execution of sending and receiving edges is done either according to a $1:1$ or $1:n$-relation, where the former situation is denoted binary synchronization and the later case as brodcasting.

Binary synchronization. Pairs of sending and receiving edges labelled with the same signal identifier can only be executed jointly. If there are more than one sending or receiving edge, all possible pairs of sending and receiving edges are scheduled for being individually executed, one pair after the other and potentially yielding different system states.

Broadcast. Execution of a broadcasting edge can take place in isolation, i. e. without any receiving edge being executable. However, in case were are some receiving edges enabled, broadcasting edge and receiving edges have to be executed together. Execution of the broadcasting edge in isolation can be avoided by exploiting location invariants, and incrementing a global variable. E.g., the target location of a sending edge requires that a dedicated variable i is strictly larger than 0. The increment of the variable takes place upon execution of the broadcast-receiving edges. This way one may also enforce full synchronization among one sender and K receivers.

For further details on the modeling mechanisms of the timed model checker Uppaal, please refer to its online manual or related publications, e. g. [7,8]. The automata decisive for the presented technique are depicted in Fig. 4 to 6 and more details follow in Sec. 5 and 6.

2.3.3 Model checking: verifying key metrics of real-time systems

Determining if for a TA-based system model $\mathcal{M}$ a dedicated property Φ holds is denoted as timed verification. The properties to be verified can either be associated with states, as so called state or safety properties or with sequences of states. In this paper we concentrate on the verification of state properties as it already allows to bound the aforementioned waiting times. These quantities can directly be associated with clocks or variables of the TA, s. t. states can be either labelled accordingly or one makes use of an observer TA for guarding the respective property, e. g. a delay bound. Asserting the invalidity of (bad) state properties is straight-forward: one simply annotates

the locations of the TA-based models with dedicated labels, e. g. `violation` or any clock, variable constraint respectively and checks the state space of the TA for the absence of a state containing the according location label. With Uppaal, the absence of violating locations can be queried by the statement `A[] not violation`, which stands for 'always invariantly' $not\ violation$.
Verifying the validity of a delay bound is in fact more evolved: with an observer TA one non-deterministically measures the delay for any pair of input and corresponding output event by a clock x. The observer TA is a receiver of the respective input and output signal, it resets clock x upon reception of the input signal and resides in location $pause$ until it receives the output signal. By querying if statement `A[](pause imply x` $\leq$ `D)` holds, with D as respective delay bound, one asserts that the time elapsing between the input and the respective output event is bounded for all such pairs of signals by D time units. It is straight-forward to use this mechanism for finding an upper bound on the waiting time of a task as induced by the blocking behaviour experienced when waiting for accessing a shared resource such as the main memory.

3 Abstraction I: Phase-based model of computation

3.1 The superblock model for characterizing executables

This work considers systems consisting of N processing elements or cores, $\mathfrak{P} = \{p_1, \ldots, p_N\}$. To each core $p_j \in \mathfrak{P}$ we statically assign a set $\mathfrak{T}_j$, which are the real-time tasks executing on p_j. The tasks are assumed to be independent and require access to a common (shared) resource, such as an interconnection bus to a shared memory. For structuring the tasks, we exploit the so called superblock model [36]. This phase-based model of computation assumes that each task consists of a static sequence of superblocks, which are non-preemptable execution units with known worst and best case execution (computation) times, and upper and lower bounds on the number of accesses to the shared resource. As we consider in-order execution on the cores, a superblock's execution is stalled every time an access request is pending, i. e. the access request has been issued to the resource, but has not been served yet.
We assume a static schedule among all superblocks originating from different tasks, but executing on the same core p_j, i. e. all superblocks of a core follow a pre-defined and static execution order. This order refers to preemptive or non-preemptive scheduling at the task level. In case of non-preemptive scheduling, the superblocks are ordered according to the sequence of tasks. With preemptive scheduling, each superblock refers to the possibly partial execution of a task, the interleaved sequence of superblcoks reflects than the interleaved execution sequence of the tasks of $\mathfrak{T}_j$.
Let the sequence of superblocks executing on core p_j be denoted $\mathcal{S}_j = (s_{1,j}, s_{2,j}, \ldots, s_{|\mathcal{S}_j|,j})$. This sequence is re-started after W_j time units, in the following referred to as processing cycle. The earliest starting time of a superblock $s_{i,j}$ within the k'th processing cycle is defined by $\rho_{i,j} + (k-1) \cdot W_j$, where $\rho_{i,j}$ is the relative offset of i'th superblock on the j'th core. Analogously, its *relative deadline* is denoted $\ell_{i,j}$ and the absolute deadline of superblock $s_{i,j}$ within the k'th processing

cycle is computed as $\rho_{i,j} + (k-1) \cdot W_j + \ell_{i,j}$.
Up to now, a superblock is fully described by its worst-case and best-case execution time, denoted as WCET and BCET respectively, and its upper and lower bounds on the number of accesses to the shared resource. Because execution of superblock $s_{i,j}$ can only start once the preceding superblock $s_{i-1,j}$ has been processed and their worst-case response time (WCRT) is unknown, the relative starting time of suberblocks are unknown too. Worst-case response time (WCRT) are unknown, as as blocking times of a superblock's execution at a shared resource are not known. The blocking takes place when an access request has been issued, but it has not been served at the shared resource yet.
In order to reduce the non-determinism w. r. t. the occurrence of access requests, superblocks are further divided into three phases, known as the acquisition, the execution, and the replication phase. As an example, one may consider tasks and their accesses to the main memory: during the acquisition phase a task reads the required data from the main memory. The writing back of the modified/new data takes place in the replication phase, after the non-interruptible computations of the execution phase have been completed. This is a common model for signal-processing and control real-time applications. For our analysis, we consider in particular the dedicated superblock model, in which resource accesses are performed sequentially in the acquisition and the replication phase, while no accesses are required in the execution phase. In this setting, we use the parameters $\mu_{i,j}^{min,\{a,r\}}$ and $\mu_{i,j}^{max,\{a,r\}}$ for addressing the minimum and maximum number of access requests in the *(a)cquisition phase* and *(r)eplication phase*, respectively. For simplification, we consider the computation time to initiate the accesses in the acquisition or replication phase, $ex_{i,j}^{\{a,r\}}$, as being zero. If this time is too large, i.e., cannot be neglected, we divide the corresponding acquisition or replication phases into several smaller acquisition/replication and execution phases so that eventually, each phase features either computation or accesses only. The lower and upper bound on the required computation time in the *(e)xecution phase* are denoted $ex_{i,j}^{min,e}$ and $ex_{i,j}^{max,e}$, respectively. Note that the terms computation time and execution time are used interchangeably in the following and do not include the time spent on resource accesses and blocking.
For a task with logical branches, the above phases and their lower and upper bounds on the parameters may not be tight estimates. However, in any case, they are assumed to be conservative, i. e. best-case bounds might be too optimistic and upper bounds might be too pessimistic. The values can be derived either by profiling and measurement for the case of soft real-time systems, as shown in [30], or when hard bounds are necessary, by applying static analysis and abstract interpretation techniques, e.g. in the style of [25] or by using commercial tools such as the aiT analysis tool [17].

3.2 The Superblock model in practice

For systems employing caches, we rely on the PRedictable Execution Model (PREM) to carefully control when tasks access shared memory. Under PREM, a task's su-

perblocks[1] are specially compiled such that during the acquisition phase, the core accesses either main memory or last level cache to prefetch into private core cache all memory required during the rest of the superblock. Then, during the subsequent execution phase, the task performs computation on the previously-fetched data, without incurring any private cache misses. Finally, as first proposed in [6], we assume that during the final replication phase, write-backs can be forced for modified cache lines in the private cache [37]. This ensures that modified data is written back to the shared memory resource.

Two PREM implementations have been proposed in the literature. In the original implementation [29], which we employ in the evaluation Section 7, PREM-compliant tasks are produced through modifications of existing applications written in standard high-level languages such as C. Certain constraints are present which limit the applications that can be made PREM-compliant. These constraints, however, are not significantly more restrictive than those imposed by state-of-the-art real-time static timing-analysis tools. The PREM implementation also uses a real-time C compiler prototype built using the LLVM Compiler Infrastructure [22]. In the implementation described in [26], a set of profiling techniques are employed to determine the set of virtual memory pages accessed during each superblock. Selected pages are then prefetched in cache. While this technique suffers a potential performance penalty since it can only manage memory at the larger granularity of a 4KB memory page, it has the benefit of requiring no code modifications.

Finally, one necessary consideration when using PREM is that each time a cache line is prefetched, it has the potential to evict another cache line. This can cause two types of problems. First, the new cache line may replace some other cache line which was to be used in the upcoming execution phase (*self-eviction*). Second, it can evict a cache line previously prefetched by another task that has been preempted (*storage interference*). To prevent self-eviction, analysis of cache associativity and the cache replacement policy can be used to compute an upper bound on the allowed memory footprint of a superblock [29]. Alternatively, memory regions can be locked in cache [26], thus preventing unwanted evictions. Storage interference is avoided because in our model, once a superblock starts running its acquisition phase, it will run to completion. Hence, no other task running on the same core can evict cache lines of the executing superblock from private core cache. On the other hand, when a superblock starts, we make no assumption on the state of the private cache, meaning that in the worst-case, all cache lines used during the superblock might not be available in private cache and must be fetched from either shared cache or shared main memory.

4 Abstraction II: From phases to an aggregated access request curve

This section presents how to derive an event arrival curve that bounds the resource accesses stemming from the execution of a sequence of superblocks on a single core. This is one of the fundamental abstractions which works for the scalability of the

[1] In [29], the term *predictable interval* is used with the same meaning as superblock.

presented analysis methodology, as this abstraction yields independence from the number of cores in the system under analysis.

4.1 Deriving an access request arrival curve

A sequence of dedicated superblocks S_j, executing on processing core p_j with period W_j, accesses a shared resource. The possible resource access patterns depend on the minimum and maximum number of access requests, $\mu_{i,j}^{min}$ and $\mu_{i,j}^{max}$ (sum for acquisition and replication phase), and the minimum and maximum computation time, $ex_{i,j}^{min}$ and $ex_{i,j}^{max}$ (execution phase), of each superblock $s_{i,j} \in S_j$, as well as the order of superblocks in the sequence. In the following, we introduce a method to represent abstractly the possible access patterns as an upper arrival curve. The latter will provide an upper bound on the number of access requests that can be issued by the corresponding core in any interval of time. The presented method provides tighter bounds than previously published methods [31, 13].

The arrival curve for a core p_j is derived assuming that no interference is present on the shared resource. Namely, the superblock sequence of p_j is analyzed in *isolation*, as if it had exclusive access to the resource. This is an overapproximation as it maximizes the issued resource access requests w. r. t. the time interval domain. The computation of the arrival curve is organized as follows.

4.1.1 Computation of an upper access request trace

In the time domain, different access request traces R_j can be derived for core p_j if one considers all possible instances of accesses and computation times within the lower/upper bounds of the superblocks in S_j. Among all traces, we identify the upper trace R_j^u, namely the trace computed by considering the *maximum* number of accesses and the *minimum* computation time for all superblocks of S_j, i.e., $R_j^u(t) \geq R_j(t)$, $\forall t \in [0, \infty)$.

It is clear that no other trace can represent more accesses at any time $t \geq 0$ than the suggested one. Let us prove this informally and for the purpose consider a single execution of the superblock sequence in the interval $[0, W_j)$. If we have computed a trace with less than the maximum access requests for at least one superblock, the computed trace will include less access requests in $[0, W_j)$ than R_j^u. On the other hand, if a larger than the minimum computation time is considered for at least one superblock, emission of the maximum number of access requests can be delayed in time when compared with R_j^u.

Note that since the core p_j is examined in isolation, it is assumed that the inter-arrival time between any two successive accesses issued by the core is equal to the resource access latency, which we denote C. This way, every time p_j emits an access request, the access is assumed to be granted immediately, therefore being served within C time units.

To illustrate the above with an example, we consider a single superblock specified in Table 1. If the superblock is solely executed on a core, then the trace R^u of access requests that this core can emit is given in Fig. 1, where μ^{tot} is the total maximum

Table 1 Superblock and Shared Resource Parameters

Acquisition $\{\mu^{a,min}, \mu^{a,max}\}$	{3,4}
Execution $\{ex^{min}, ex^{max}\}$	{50,70}
Replication $\{\mu^{r,min}, \mu^{r,max}\}$	{1,2}
Period W	250
Access Latency C	20

number of accesses in one period, i.e., in this example, $\mu^{tot} = \mu^{a,max} + \mu^{r,max}$. In particular, in the figure, in the interval $[0, 80)$ R^u makes 4 steps which reflect the maximum number of accesses in the acquisition phase of the superblock. Each access takes the same access latency of $C = 20$ which corresponds to the flat segments of length 20 after each step in R^u. These acquisition phase accesses are followed by the minimum execution phase of length 50 which corresponds to the flat segment in R^u in the interval $[80, 130)$. In the interval $[130, 170)$, R^u makes 2 steps which correspond to the maximum number of accesses in the replication phase. They are followed by a flat segment for the interval $[170, 250)$ which is denoted as the gap until the next execution of the superblock can start. Determining the length of this gap for the purpose of our analysis will be described next.

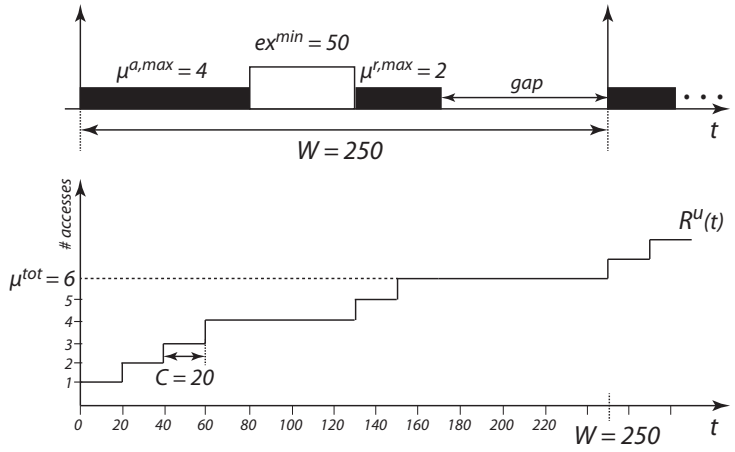


Fig. 1 Upper access request trace derived from the superblock parameters specified in Table 1

4.1.2 Lower bounding the gap between re-executions

In the computed upper access request trace, we can identify the *gap* (idle interval) between the end of the last phase of S_j and the start of the next processing cycle. In the example trace of Fig. 1, this gap can be computed as $W - (\mu^{tot} \cdot C + ex^{min})$ as a result of the isolation assumption. However, on the actual multicore system, where p_j competes against other cores for access to the shared resource, it is, in fact, possible that the incurred delays will cause the execution of S_j to be extended closer to the

end of the processing cycle. Therefore, in the general case, where no upper bounds on these delays can be provided, we need to consider a zero-interval as the minimum gap between two successive executions of S_j[2].

This estimation can be refined in particular cases. For instance, when the resource is FCFS- or RR-arbitrated, in the worst-case every access of S_j can be delayed by all $(N-1)$ competing cores. Since access requests are blocking, every core can have up to 1 pending access request at a time. Therefore, each access request of S_j can be delayed by at most $(N-1) \cdot C$ time units due to the interference of the other cores. Therefore, a lower bound on the gap can be computed as follows:

$$gap^{min} = W_j - (N \cdot C \cdot \sum_{\forall s_{i,j} \in S_j} (\mu_{i,j}^{a,max} + \mu_{i,j}^{r,max}) + \sum_{\forall s_{i,j} \in S_j} ex_{i,j}^{max}) \,. \quad (2)$$

4.1.3 Deriving the arrival curve

Derivation of p_j's access request arrival curve follows from the computed upper trace R_j^u and the lower bound on the gap gap^{min}. Before we proceed, we need to define the shift operator $\triangleright$ as:

$$(R \triangleright g)(t) = \begin{cases} R(t-g) & , t > \max(g,0) \\ 0 & , 0 \leq t \leq \max(g,0) \end{cases} \quad (3)$$

The arrival curve that will be derived upper bounds the amount of access requests that p_j can emit in *any* time interval Δ. To safely obtain this bound, one has to consider time intervals $[t-\Delta, t)$, with $\Delta \geq 0$ and $t \geq \Delta$, which may start any time after the first triggering of the superblock sequence S_j. Depending on the number of processing cycles over which the intervals may span, we differentiate three cases for the start and the end of the interval Δ, as illustrated in Fig. 2.

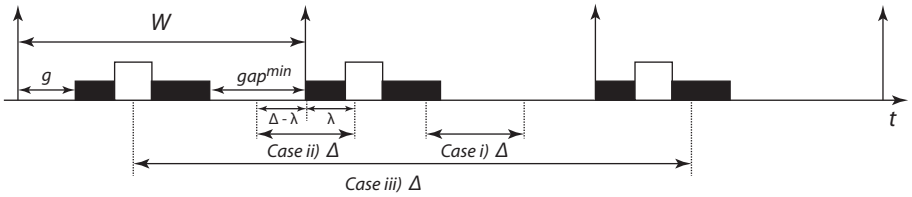


Fig. 2 Three cases for the position of the considered interval Δ: i) within one processing cycle, ii) starting in one processing cycle and ending in the next processing cycle, iii) spanning more than two processing cycles

(i) $\mathbf{0 \leq t - \Delta \leq t \leq W_j}$. Here, the intervals $[t-\Delta, t)$ are contained entirely in one processing cycle, as depicted in Fig. 2. The number of access requests in such an interval is computed simply as: $R_j(t) - R_j(t-\Delta)$. An upper bound on the access requests of this interval is computed by considering all possible positions

[2] Provided that the system is schedulable, i.e., execution of a superblock sequence always finishes within the current processing cycle.

for the interval Δ on the upper access request trace R_j^u and taking the maximum as follows:

$$\alpha_{j,i}^u(\Delta) = \max_{0 \le t}(R_j^u(t) - R_j^u(t-\Delta))\,. \tag{4}$$

(ii) $\mathbf{0 \le t - \Delta \le W_j \le t \le 2W_j}$. In this case, the intervals $[t-\Delta, t)$ span over two processing cycles, as they start in one processing cycle and end in the next one. An example is shown in Fig. 2. Again, the number of access requests in this interval is calculated with: $R_j(t) - R_j(t-\Delta)$.
Let us substitute: $\lambda := t - W_j$. Then $R_j(t)$ can be expressed as $R_j(W_j) + R_j(\lambda)$. In order to obtain an upper bound, we use the upper access request trace: $R_j^u(W_j) + R_j^u(\lambda)$. The maximum total number of access requests in one processing cycle is a constant, calculated as $\mu_j^{tot,max} = \sum_{\forall s_{i,j} \in S_j}(\mu_{i,j}^{a,max} + \mu_{i,j}^{r,max})$, i.e., we have $R_j^u(W_j) = \mu_j^{tot,max}$.
After the substitution, we can express $R_j(t-\Delta)$ as $R_j(W_j + \lambda - \Delta)$. In order to obtain an upper bound, we need to consider the minimum gap between the end of the superblock sequence in the first processing cycle and the start of the superblock sequence in the next processing cycle. For the purpose, we use the shifted to the right upper access request trace $R_j^u \triangleright g$ which takes into account the lower bound on the gap, gap^{min}, from Eq. 2, where g is computed as follows:

$$g = W_j - (C \cdot \sum_{\forall s_{i,j} \in S_j}(\mu_{i,j}^{a,max} + \mu_{i,j}^{r,max}) + \sum_{\forall s_{i,j} \in S_j} ex_{i,j}^{min}) - gap^{min}\,. \tag{5}$$

Considering all possible positions of the interval Δ and taking the maximum, for the arrival curve in this case we obtain:

$$\alpha_{j,ii}^u(\Delta) = \max_{0 \le \lambda \le \Delta}(\mu_j^{tot,max} + R_j^u(\lambda) - (R_j^u \triangleright g)(W_j + \lambda - \Delta))\,. \tag{6}$$

(iii) $\mathbf{0 \le t - \Delta \le kW_j \le (k+1)W_j \le t}$, $\mathbf{k \ge 1}$. In this case, the intervals $[t - \Delta, t)$ may span over more than two processing cycles, as shown in Fig. 2. We can observe that this case is similar to the previous one, however, the end of the interval Δ is not in the next processing cycle, but can be in later processing cycles. Therefore, knowing that the maximum number of access requests in one processing cycles is $\mu_j^{tot,max}$, and having K number of processing cycles between the start and the end of interval Δ, we can use the results from the previous case to obtain an arrival curve as follows:

$$\alpha_{j,iii}^u(\Delta) = \max_{1 \le K \le \lfloor \frac{\Delta}{W_j} \rfloor}\{\alpha_{j,ii}^u(\Delta - K \cdot W_j) + K \cdot \mu_j^{max,tot}\}\,. \tag{7}$$

Combining the individual results of all three cases, we obtain the upper arrival curve α_j^u that upper bounds all access request traces of sequence S_j executing on core p_j for any time interval $\Delta \ge 0$. To this end, we take the maximum of expressions (4), (6), and (7) as follows:

$$\alpha_j^u(\Delta) = \max\{\alpha_{j,i}^u(\Delta), \alpha_{j,ii}^u(\Delta), \alpha_{j,iii}^u(\Delta)\}\,. \tag{8}$$

The arrival curve obtained by Eq. 8 is a safe upper bound on the access requests that can be issued by a core, but it is more accurate than the upper bounds derived with previous methods [31, 13].

The method presented in [31] introduces inaccuracy of the computation of the arrival curve because for simplicity it assumes zero inter-arrival time of access requests from the core. The method presented here improves on this by considering that the minimum inter-arrival time is bounded by the access latency C of the shared resource.

Similarly, the method presented in [13] introduces inaccuracy because it considers that the minimum gap gap^{min} can appear between all successive executions of a superblock sequence, effectively shortening the processing cycle of the core, while the method presented here always considers the correct processing cycle.

4.2 Deriving the interference curve of multiple cores

In Sec. 6 we will use an arrival curve to represent the access patterns of several processing cores, when analyzing the WCRT of a particular superblock sequence which is executed on a *core under analysis* (CUA). In this case, the interference caused by all the other cores is taken as the sum of their individual arrival curves, which are computed with Eq. 8 and by considering the cores in isolation. The sum represents a safe over-approximation of the interference that the cores may cause on the arbiter as we deal with monotone systems, where the number of requests received by the arbiter for any given interval Δ cannot exceed the sum of the issued requests by the interfering cores.

The sum of the arrival curves of all cores except the CUA is mentioned in the following as the interference curve α and it is computed as follows:

$$\alpha(\Delta) = \sum_{p_j \in \mathfrak{P} \setminus \{p_{i(CUA)}\}} \alpha_j^u(\Delta)\,. \tag{9}$$

5 Abstraction III: From access request curves to state-based request generators

It is our aim to carry out system analysis based on Timed Automata and thereby precisely capture relevant implementation details of employed resource arbiters. Hence, it is required to translate the aggregated access request curve from above into a state-based request generator. Such a generator is capable of emitting all timed traces of access requests as bounded by the upper and lower access request curve. Please note, for the lower bound we use the trivial bound which is the constant 0-function.

To simplify the aggregated access request curve, we overapproximate it with the following staircase curve:

$$\alpha^{st}(\Delta) := N + \left\lfloor \frac{\Delta}{\delta} \right\rfloor \tag{10}$$

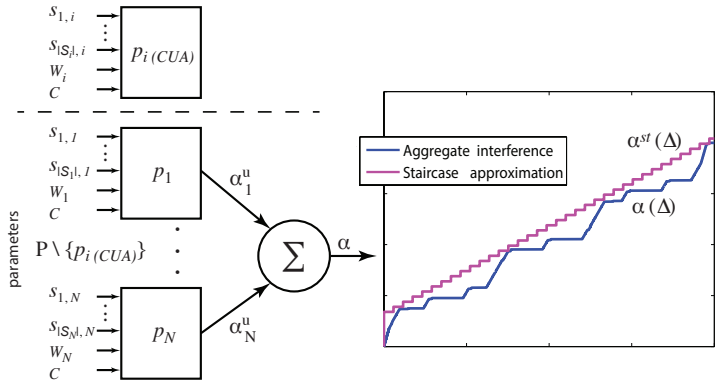
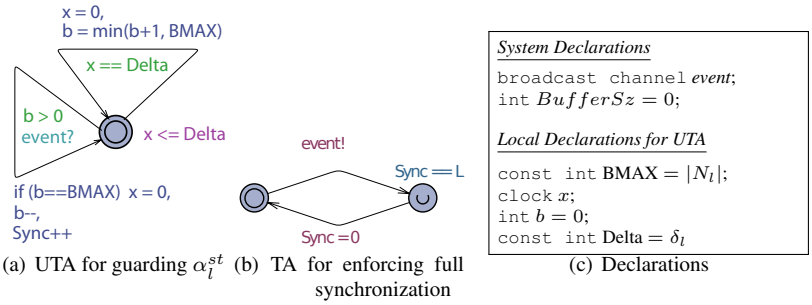


Fig. 3 RTC Interference representation



(a) UTA for guarding α_l^{st} (b) TA for enforcing full synchronization (c) Declarations

Fig. 4 TA-based implementation of Resource-access request event curve

where for soundness $\alpha(\Delta) \subseteq \alpha^{st}(\Delta)$ for all $\Delta \in \mathbb{R}^+$ needs to hold.[3]

In total this gives the following approach as illustrated in Fig. 3: we represent the access requests of each core by an arrival curve $\alpha_j(\Delta)$. The sum over the n arrival curves represents then the combined access requests of the respective cores and the obtained complex curve is safely over-approximated by a single staircase curve.

How to embed complex staircase arrival curve(s) was firstly presented in [19]. In this work we are limited to the case of a single staircase curve only. Arrival patterns abstractly defined by a single staircase curve, here α^{st}, can (exhaustively) be generated with the TA shown in Fig. 4, for the proof refer to [20].

6 WCRT analysis under resource contention

For the state-based analysis of resource contention scenarios in multicores, the system specification of Sec. 3 can be modeled by a network of cooperating TA. This section presents the TA that were used to model the system components. We discuss

[3] Instead of a single staircase curve, α^{st} can also be composed from sets of staircase curves put together via nested maximum and minimum operations[20,32]. This allows to model more complex curves, however, substantially adds to the complexity of the model checking problem to be solved when determining the WCRT of the CUA.

how the temporal properties of the system can be verified using the Uppaal timed model checker and we introduce a set of abstractions to reduce the complexity of the analysis.

6.1 Multicore system as a TA network

Modeling Concurrent Execution of the Cores. The concurrent execution of sequences of superblocks on the different cores is modeled using instances of two TA, denoted *Scheduler* and *Superblock* in the following. The implementation w. r. t. the Uppaal input language is given in Fig. 5. In a system with N cores and a total of M superblocks executed on them, $(N+M)$ component TA are allocated for modeling the concurrent execution of superblocks. The employed pattern works as follows.

Each of the N instances of the TA *Scheduler* enforces the activation order of superblocks on its associated core p_j. Whenever a new superblock must be executed, the respective instance of TA *Scheduler* emits a `start[sid]` signal, with `start` being an array of channels and `sid` the index of the respective superblock. Due to the underlying composition mechanism, this yields a joint execution of the `start`-labelled edges by the respective instances of the TA *Scheduler* and the respective instance of the TA *Superblock*. When the superblock's execution is completed, both components, i. e. *Scheduler* and *Superblock*, execute their `finish`-labelled edges, where we once again employ an array of channels, here `finish`. Once the last superblock in a processing cycle has terminated, the instance of TA *Scheduler* moves to location `EndOfPeriod`, where it resides until the processing cycle's period is reached. After W_j time units since the activation of the first superblock of core p_j, the *Scheduler* TA triggers a new execution sequence of superblocks w. r. t. to core p_j.

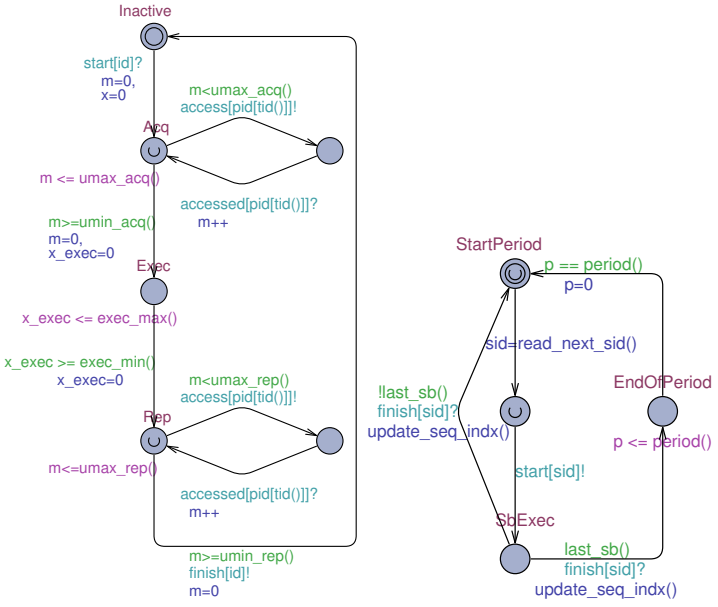
Modeling Superblocks. A *Superblock* TA models the three phases of each superblock and is parameterized by the lower and upper bounds on access requests and computation times. Once a *Superblock* instance is activated, it enters the `Acq` location, where the respective TA resides until a non-deterministically selected number of resource accesses within the specified bounds has been issued and served. Access requests are issued through channel `access[pid]`, whereas notification of their completion is received by the arbiter through channel `accessed[pid]`. For location `Acq`, we use Uppaal's concept of urgent locations to ensure that no computation time passes between successive requests from the same core, which complies with the specification of our system model. Subsequently, the *Superblock* TA moves to the `Exec` location, where computation (without resource accesses) is performed. The clock `x_exec` measures here the elapsed computation time to ensure that the superblock's upper and lower bounds, $exec^{max}$ and $exec^{min}$, respectively, are guarded. The behavior of the TA in the following location `Rep` is identical to that modeled with location `Acq`.

For the case of a single superblock in S_j, as shown in Fig. 5(a), clock `x` is used to measure its total execution time. Checking the maximum value of clock `x` while the TA is *not* in its `Inactive` location allows to obtain the WCRT of the whole

superblock. With Uppaal this is done by specifying the lowest value of `WCRT`, for which the safety property:

```
A[] Superblock(i).Rep imply Superblock(i).x <= WCRT
```

holds. The property implies that location `Rep` is never traversed with `x` showing a clock value larger than `WCRT`. This way we ensure that for all reachable states, the value of superblock s_i's clock `x` is bounded by `WCRT`. To find the lowest `WCRT` satisfying the previous property, binary search can be applied. Upon termination, the binary search will deliver a safe and tight bound of the superblock's WCRT. Similarly a WCRT bound on a sequence S_j (a task) can be calculated by adapting the TA in Fig. 5(a) to model more than three phases.

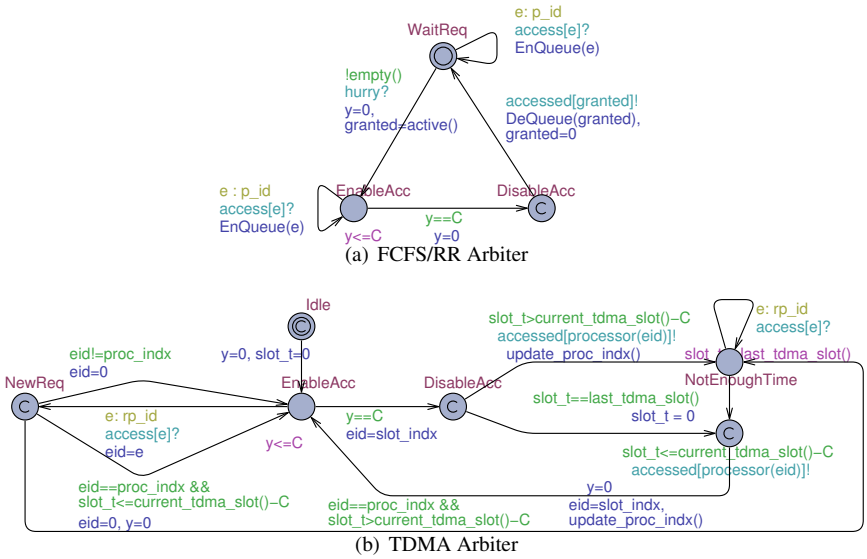


(a) Superblock (b) Static Scheduler

Fig. 5 Superblock and Static Scheduler TA

Modeling Resource Arbitration. In this work, we consider arbitration policies of any complexity for the shared resource. For the purposes of presentation, in the following we focus on two event-driven strategies, namely first-come-first-serve (FCFS) and round-robin (RR) and one time-driven strategy, namely TDMA. A hybrid event and time-driven strategy, i.e., the industrial bus arbitration protocol FlexRay [2] has been also modeled in [13], but its description is omitted here for brevity.

We assume that resource accesses are synchronous (blocking) and non-preemptive, and that the access latency is equal to C once a request is granted by the arbiter. The TA modeling the four aforementioned arbitration policies are depicted in Fig. 6. Depending on the implemented policy, the respective model is included in the



(a) FCFS/RR Arbiter

(b) TDMA Arbiter

Fig. 6 TA representation of arbitration mechanisms

TA network of our system. In the following we briefly detail on the resource arbiter models.

The *FCFS* and the *RR Arbiter* share a similar TA, depicted in Fig. 6(a). Both arbiters maintain a queue with the identifiers of the cores that have a pending access request. In the case of FCFS, this is a FIFO queue with capacity N, since each core can have at maximum one pending request at a time. When a new request arrives, the arbiter identifies its source, i. e. the emitting core, and appends the respective identifier to the tail of the FIFO queue. If the queue is not empty, the arbiter enables access to the shared resource for the oldest request (`active()` returns the queue's head). After C time units, the access is completed, and the arbiter removes the head of the FIFO queue and notifies the *Superblock* TA that the pending request has been processed.

For the RR arbiter a bitmap is maintained instead of a FIFO queue. Each position of it corresponds to one of the cores and pending requests are flags with the respective bit set to 1. As long as at least one bit is set, the arbiter parses the bitmap sequentially granting access to the first request it encounters (return value of `active()`).

The *TDMA Arbiter* of Fig. 6(b) implements the predefined TDMA arbitration cycle, in which each core has one or more assigned time slots. It is assumed that the cores (*Scheduler* instances) and the *Arbiter* initialize simultaneously such that the first slot on the shared resource and the first superblock on each core start at time 0 (assuming synchronized processing cycles among the cores). The arbiter's clock `slot_t` measures the elapsed time since the beginning of each TDMA slot. When `slot_t` becomes equal to the duration of the current slot, the clock is reset and a new time slot begins. According to this, a new access request from core `eid` is served as soon as it arrives at the arbiter on condition that (a) the current slot is assigned to `eid` and (b) the remaining time before the slot expires is large enough for the processing

of the access. If at least one condition is not fulfilled, the pending request remains 'stored' in the arbiter's queue and is granted as soon as the next dedicated slot to `eid` begins.

In all *Arbiter* TA, new access requests can be received any time, either when the queue is empty or while the resource is being accessed. Multiple requests can also arrive simultaneously.

6.2 With an access request generator to a reduced number of TA

A network consisting of N *Scheduler*, M *Superblock* and 1 *Arbiter* TA is used to model the multicore architectures under analysis. By verifying appropriate temporal properties in Uppaal, we can derive WCRT estimates for each superblock or sequence of superblocks that is executed on a processing core. However, scaling is related to the number of TA-based component models as the verification effort of the model checker depends on the number of clocks and clock constants used in the overall model, not to mention any variables. In the following sections, we propose safe abstractions for achieving a better analysis scalability.

In the proposed abstractions, only one processing core p_i (core under analysis, CUA) is considered at a time, while the behaviour of all remaining cores, which compete against it for access to the shared resource, is abstracted (not ignored). To model the access requests of abstracted cores, we use arrival curves. This way an arrival curve α capturing the aggregate interference pattern of the abstracted cores can be constructed (as in Eq. 9) and then, modeled using TA in the form of an access request generator (Sec. 5). The overall system is modeled as a network of TA which contains one instance of the TA *Scheduler* and $|S_i|$ instances of the TA *Superblock* (Fig. 5), one instance of the TA *Arbiter* (Fig. 6), and the TA implementing the access request generator (see Fig. 4). We over-approximate the access request curve α of the cores by a single staircase curve α^{st} only. The staircase function is selected so that (a) it coincides with the original α on the long-term rate and (b) it has the minimum vertical distance to it. The access request generator emits then traces of interfering access requests bounded by α^{st}. Emission of those requests can be restricted further, given that at any time the amount of pending requests in the system cannot be greater than the number of cores N.

Overall, the discussed abstraction yields a model whose number of component TA is independent of the number of cores of the system under analysis. To illustrate the reduction in the size of the TA-based system specification, one may consider a system with 32 cores, each executing a single superblock only. If all cores were modeled individually, i. e. each by its own component TA, the complete system model would consist of 65 TA: 32 instances of the TA *Scheduler*, 32 instances of the TA *Superblock*, and one instance of the TA *Arbiter*. Therefore, the system model would contain in total 97 clock variables and 128 synchronization channels. By applying the above abstraction, one obtains a system model which contains five component TA only: one instance of the TA *Scheduler*, one instance of the TA *Superblock*, one instance of the TA *Arbiter*, and one instance of the *Request Generator* which consists

of two TA. In total, this yields a system model with 5 clocks and 6 synchronization channels.

Substitution of the $(N-1)$ *Scheduler* and the $(M-|S_i|)$ *Superblock* TA instances of the abstracted cores with just a pair of interference generating TA is expected to alleviate significantly the verification effort for the WCRT estimation of the superblocks executing on CUA (core p_i). This comes with the cost of over-approximation, since the interference arrival curve α provides a conservative upper bound of the interfering access requests in the time-interval domain and a^{st} over-approximates it. As shown in Sec. 7, though, the pessimism in the WCRT estimates for the superblocks of CUA is limited.

6.3 Adaptations to improve scalability

Besides the basic abstraction steps, i. e. the dedicated superblock model of execution, the interference representation with arrival curves, and the modeling of the latter with TA, further abstractions and optimizations of our system specification can be considered. We briefly discuss these in the following, such that the experimental results of Sec. 7 are reproducible:

1. For system models, where execution on all cores is modeled explicitly and the resource is FCFS or RR-arbitrated, the state space exploration for a superblock's WCRT can be restricted to the duration of one hyper-period of the cores' processing cycles. The hyper-period is defined as the least common multiple of the cycles' periods ($lcm(W_1, \cdots, W_N)$) and within its duration all possible interference patterns are exhibited. Therefore, deriving a superblock's WCRT by exploring the feasible scenarios in this time window only is safe. Note that a similar simplification can be applied in case of TDMA- arbitrated resources, too, if the hyper-period is redefined to account for the period of the arbitration cycle ($lcm(W_1, \cdots, W_N, \Theta)$, where Θ denotes the length of the arbitration cycle).
2. In system specifications, where execution on some cores is abstracted and the resource is FCFS or RR-arbitrated, the *Superblock* TA can be simplified to model not periodic execution, but a single execution. Since all feasible interference streams bounded by α^{st} can be explored for the time interval of one superblock execution, the WCRT observed during this interval is a safe WCRT bound. This simplification also eliminates the need for including the *Scheduler* and the remaining *Superblock* instances of the CUA in the TA system model. The same can be applied to systems with a TDMA arbiter which requires to enumerate and model all possible offsets for the starting time of the superblock within the respective arbitration cycle.
3. To bound the WCRT of a superblock, one can add the WCRT of the individual superblock phases. Similarly as before, we can model single acquisition or replication phases and explore all interference streams bounded by α_{st} for the time interval of one phase execution. Based on the arrival curve properties, the obtained WCRT bound for each phase is safe. For the execution phase, one can simply consider the maximum execution time and add this to the previous sum.

The total WCRT of the superblock (sum) of the individual phases can be more pessimistic than the WCRT found when the sequence of superblocks is analyzed in a single step. This is because it may not be possible for all phases to exhibit their WCRT in a single superblock execution. Nevertheless, this simplification reduces the verification effort, by dividing the original problem into smaller ones, each analyzed independently.

4. In system models with a TDMA resource arbiter, the interference from the competing cores can be ignored (not modeled) due to the timing isolation that a TDMA scheme offers. Namely, for deriving a superblock's WCRT, the model checker needs to consider all possible relative offsets between the arrival of CUA's access requests and the beginning of its dedicated slot in the arbitration cycle. The interference from the remaining cores does not affect the superblock's WCRT.
5. In system models with a FCFS or RR resource arbiter, granularity of communication between the *Access Request Generator* and the *Arbiter* TA may become coarser, by letting access requests arrive in *bursts* at the arbiter. For this, the interference generating TA can emit requests in bursts of b, where b is a divider of the maximum burst size N_l in Eq. 10 and $b \leq N-1$. The arbiter TA can be adapted to store the requests and serve them like in the original system (as if they were emitted one by one). New bursts of interfering requests can be generated any time after the previous b requests have been served. This optimization decreases the need for inter-automata synchronization and also the number of explored traces below α^{st}, since the inter-arrival times among the requests in a burst are no longer non-deterministic, but fixed to 0. The traces that are not explored could not cause worse response times for CUA's superblocks than the ones with the bursty arrivals. This is because, if some of the b interfering access requests arrived later (non-zero inter-arrival time), the next access of CUA would suffer equal or less delay compared to the bursty case. Specifically, if the "delayed" interfering requests arrived still before the next request of CUA (FCFS arbitration) or before the turn of CUA (RR arbitration), the delay for serving the latter's request would be the same as if the interfering requests had arrived in a burst. Otherwise, the "delayed" interference requests would not be served before CUA's request, thus reducing its response time. Therefore, the omission of the non-bursty traces has no effect on the correctness or tightness of the WCRT estimates.
6. The *Access Request Generator* requires to reset its clock once it has emitted the maximal number of resource access request at a single point in time, i. e. it has produced a burst of access requests. It is safe to omit this clock reset, since the *Access Request Generator* could simply emit more events than bounded by the original curve α. E. g. any *Access Request Generator* derived from the TA of Fig. 4 could release BMAX events just before clock x expires; successive traversals of edge "event?". Without resetting clock x upon the $BMAX$'th travesal, one could actually release more access requests once clock x expires, i. e. $x == Delta$ holds. This yields emissions of more requests than bounded by the original α, which in turn could introduce pessimism into the analysis. On the other hand, it might help with lowering the state space explosion, as it reduces the number of clock resets. In the case of our model, we can not notice any additional pessimism w. r. t. the worst case response time of the task. That is because the maximum

number of pending requests is not derived from $BMAX$, but from the number of cores in the system. Additional resource access requests can only be issued at the rate of service experienced at the shared resource.

7 Evaluation

In this section, we provide an in-depth evaluation of the presented techniques, where we consider systems with two to six cores, on which a set of industrial benchmarks is executed. The cores access the main memory in a round-robin fashion when cache misses occur. We model the respective systems either fully with TA (FTA) or a combination of TA and arrival curves (ATA) and compare the *scalability* of the two analysis methodologies and the *accuracy* of the obtained results. We further compare our derived analysis bounds with results obtained from architectural simulations. While simulations by definition might fail to reveal the real worst-case scenario, they are nevertheless useful in validating the overall system settings and providing a lower bound on worst-case response time analysis to compare with the safe upper bound provided by analysis.
A comparison of the formal modeling and analysis techniques FTA and ATA with other state-of-the art analytic approaches for bounding the WCRT in the setting of phase-structured tasks was already presented in [13].
We base our evaluation on a simulated multicore platform using private LVL1 and LVL2 caches and shared main memory. The Gem5 [9] architectural simulator is used to execute selected benchmarks and obtain superblock parameters for accesses to main memory. The simulator is configured as follows: in-order core based on x86 instruction set running at 1 GHz; split LVL1 cache of 32kB for instruction and 64kB for data; unified LVL2 cache of 2MB; cache line size is 64 Bytes; a constant time of 32 ns for each memory access. Section 7.1 provides more details on the evaluated benchmarks. Section 7.2 describes our multicore simulation settings and presents comparative results among the simulation and the two suggested analysis methods. Finally, Section 7.3 evaluates the accuracy of our analysis by comparing WCRT estimates obtained with ATA to conservative WCRT bounds that can be derived (without model checking) for the considered arbitration policy.

7.1 Benchmarks and determination of superblock parameters

To evaluate the proposed techniques, we considered six benchmarks from the AutoBench group of the EEMBC (Embedded Microprocessor Benchmark Consortium) [1] and ported them to PREM. We examined benchmark representing streaming applications that process batches of input data and produce a stream of corresponding output. Specifically, the six benchmarks we used from the AutoBench group were a2times (angle to time conversion), canrdr (response to remote CAN request), tblook (table lookup), bitmnp (bit manipulation), cacheb (cache buster) and rspeed (road speed calculation). Ideally more benchmarks would have been examined, however, as detailed in Section 3.2, making a benchmark PREM-compliant is a time-consuming

operation. Also note that a similar approach to benchmarking was employed in past papers using PREM, in particular [29,40,6].

Each benchmark in the EEMBC suite comes with a sample data file that represents typical input for the application. The benchmark comprises required initialization code, followed by a main processing loop. Each iteration of the benchmark algorithm processes a fixed amount of input data, and the number of iterations is selectable. We configured the number of iterations such that each benchmark processes its entire sample data. Then, we compiled the whole main loop into a superblock; in this way, every periodic execution of the resulting task consists of a single superblock. During the acquisition phase, the whole sample data of the benchmark, static data and code of the main loop are loaded in cache; in the execution phase, the main loop is repeated for the selected number of times; and finally, during the replication phase all output data is written back to main memory. Note that we do not include the initialization code of the benchmark in the superblock, since such code must be run only once at the beginning of the benchmark, and is not executed as part of a periodic task.

Table 7.1 provides the derived characterization for the six benchmarks run on our architectural simulator. To obtain valid measures for our superblock model, each benchmark is run in isolation on one core, with no other computation in the system. We provide the number of iterations for each benchmark. We report the maximum number of accesses $\mu^{\max,a}, \mu^{\max,r}$ for the acquisition and replication phases, the maximum execution time $ex^{\max,a}$ (ns) for the acquisition phase, as well as minimum and maximum execution times $ex^{\min,e}, ex^{\max,e}$ (ns) for the execution phase. Note that in our simulations, the replication phase is implemented by flushing the cache content before the next superblock starts, hence the cache is empty at the beginning of each acquisition phase. Since furthermore the amount of processed and modified data is constant for a given number of benchmark iterations, we have $\mu^{\min,a} = \mu^{\max,a}$ and $\mu^{\min,r} = \mu^{\max,r}$, i.e., the number of accesses in the acquisition and replication phases is constant. Also note that the largest working set in the table (see tblook) is around 271 * 64 Bytes-per-cacheline = 17,344 bytes, which can fit in LVL2 cache for commonly used processors. The number of accesses during the execution phase is zero. The execution time during the acquisition phase is also constant, and dependent on the number of instructions required to prefetch $\mu^{\max,a}$ cache lines. We do not report the execution time of the replication phase since a single instruction can be used to flush the cache independently of its content. Finally, the minimum and maximum lengths of the execution phase depend on the input data, and are thus computed as follows: we first measure the minimum and maximum execution time for a single iteration of the benchmark. Then, we obtain $ex^{\min,e}, ex^{\max,e}$ by multiplying the number of benchmark iterations by the measured minimum and maximum per-iteration time, respectively. Since the provided sample data are designed to test all code paths in the algorithms, we believe that this way, we sufficiently approximate the minimum and maximum execution time bounds that would be computed by a static analysis tool. For an in-depth comparison of PREM versus normal (non-PREM) execution, we refer the interested reader to [29,40]; in general, the number of cache line fetches under PREM is slightly higher than the number of fetches under non-PREM execution, but this overhead is relatively low for most benchmarks.

Benchmark	Iterations	PREM				
		$\mu^{\max,a}$	$\mu^{\max,r}$	$ex^{\max,a}$	$ex^{\min,e}$	$ex^{\max,e}$
a2times	256	129	26	1561	215552	296448
canrdr	512	186	26	1821	110080	1047552
rspeed	256	90	23	1094	92160	163328
tblook	128	271	23	2453	103424	795648
bitmnp	64	170	47	1678	4669760	5173056
cacheb	32	100	33	1025	8704	11872

Table 2 Benchmark Characteristics

7.2 Evaluation results: FTA and ATA versus simulation

We simulated the parallel execution of two to six tasks on an equal number of cores, where each task is composed of a single superblock, based on one of the described benchmarks. We use simulations to provide a meaningful lower bound to the worst-case response time of each task considering contention for access to the shared resource, and to validate our model. Our resource simulator uses traces of memory requests generated by running each benchmark in isolation on Gem5, as described in Section 7.1. The simulator keeps track of simultaneous requests by multiple cores, and arbitrates accesses to the shared resource based on *round-robin* arbitration. Since we assume an in-order model where the core is stalled while waiting for main memory, the simulator accounts for the delay suffered by each memory access by delaying by an equivalent amount all successive memory requests performed by the same core.

For each scenario, we simulate a synchronous system, i.e., all tasks are activated for the first time simultaneously. Each task is then executed periodically with a given period, shown in Table 3, on its dedicated core. The task periods are selected so that each task can complete execution within them. Namely, a superblock's period is at least equal to its conservative WCRT estimate:

$$WCRT_{cons} = (\mu^{max,a} + \mu^{max,r}) \cdot N \cdot C + ex^{max,a} + ex^{max,e} \tag{11}$$

which assumes that every access experiences the worst possible delay, i.e, $N \cdot C$ under RR resource arbitration (e.g., for $N = 6$ and $C = 32$ns, we have $WCRT_{cons} = 327769$ ns for benchmark a2times). As described in Section 7.1, in both the simulations and the analytical model we allow the execution time of the execution phase to vary between $ex^{\min,e}$ and $ex^{\max,e}$. Therefore, we decided to simulate each scenario until the task with the largest period has executed 2000 times, and we record the maximum response time for each task during the simulation; the length of the execution phase of each job is randomly selected based on a uniform distribution in $[ex^{\min,e}, ex^{\max,e}]$. For each scenario, we also apply the proposed WCRT analysis methods, first the one where the system is *fully* modeled with TA (FTA) and then, the one where a part of the system (interfering cores) is *abstractly* modeled with arrival curves (ATA). Note that the additional abstractions of Sec. 6.3 are also applied whenever possible. Table 4 presents the WCRT of each task as observed during simulation as well as the difference among this value and the corresponding WCRT with the two analysis methods. The difference is defined for each task as:

$$Difference = 100 \cdot \frac{WCRT_{analytic} - WCRT_{simulation}}{WCRT_{simulation}}. \tag{12}$$

Table 3 Benchmark Periods for Simulation

Benchmark	Period (ns)	Benchmark	Period (ns)
a2times	360000	tblook	900000
canrdr	1350000	bitmnp	5400000
rspeed	200000	cacheb	40000

Table 4 WCRT results of EEMBC benchmarks: FTA vs. ATA vs. simulation for RR arbitration

Cores	Benchmark Set	Simulation	FTA		ATA	
		WCRT (ns)	Difference (%)	Verif. time	Difference (%)	Verif. time (sec)
2	a2times	305540	0.28	2.4 sec	0.78	1.5
	canrdr	1058020	0.09	2.5 sec	0.29	1.9
3	a2times	308431	0.51	16 min	1.44	2.7
	canrdr	1060294	0.15	16.25 min	0.43	3.3
	rspeed	172712	0.45	15.25 min	1.48	1.8
4	a2times	312839	0.64	79.9 hrs	1.60	3.6
	canrdr	1066062	-	-	0.78	5.2
	rspeed	175588	0.85	12.2 hrs	1.88	2.2
	tblook	819105	-	-	0.44	7.0
5	a2times	315704	-	-	2.25	4.6
	canrdr	1068112	-	-	1.42	6.6
	rspeed	178424	-	-	2.29	2.8
	tblook	822330	-	-	1.13	10.8
	cacheb	28666	-	-	19.22	4.0
6	a2times	319802	-	-	2.49	5.0
	canrdr	1074540	-	-	1.45	7.2
	rspeed	181249	-	-	2.69	3.4
	tblook	827793	-	-	1.43	14.4
	cacheb	32251	-	-	19.17	4.4
	bitmnp	5202608	-	-	0.26	11.1

Table 4 presents also the time required to verify one WCRT query with the Uppaal timed model checker in each case. Note that the total verification time will be a multiple of the presented time because of the binary search performed to specify a tight response time bound. All verifications were performed with Uppaal v.4.1.7 on a system with an Intel Xeon CPU @2.90 GHz and 128 GB RAM. The experimental results should be reproducible also on machines with a lower RAM capacity (e.g., 8 GB). In particular, the model checker required up to 4.3 GB RAM for the FTA analysis, for systems with 2-3 cores. For the WCRT analysis of benchmarks a2times and rspeed in the 4-core scenario, the RAM utilization surpassed 25 GB. This is the only case, where model checking for the FTA method could fail to complete on a machine with restricted RAM capacity. Respectively, model checking for the ATA analysis required several tenths of MB in most cases. The peak RAM consumption was observed when verifying the WCRT of benchmark bitmnp in the 6-core scenario and was equal to 1.2 GB.

The FTA analysis method could be applied to systems with up to 4 cores. For tasks canrdr and tblook in the 4-core scenario as well as for the 5-core and 6-core scenarios, verification of a WCRT query with Uppaal required more than 120 hours and was, thus, aborted. Nevertheless, for the scenarios where the FTA method could be applied, the obtained WCRT results are very close to the ones observed during simulation. The maximum deviation between the corresponding values is **0.85%**,

confirming that our modeling is a good reflection of the PREM execution method. The reason is as follows: PREM gives phase structured executables, respectively compiled code, and we used this code with the simulator. Therefore, it can actually be expected that higher timing determinism allows the upper bounding to be close to the measured results of the simulation. This also gives that in the following, the comparison of ATA and simulation is almost as decisive as the direct comparison between FTA and ATA. This is particularly of value for the cases where the FTA method runs out of memory or run-time.

Note that the scalability of FTA is already improved compared to earlier model checking-based analysis methods, e.g., [15,25], in which analysis did not scale efficiently beyond 2 cores for event-driven resource arbitration schemes. This improvement can be attributed to our first proposed abstraction, namely the dedicated superblock structure of the tasks.

On the other hand, the ATA analysis method scales efficiently since the verification of each WCRT query can be completed in few seconds in all cases, almost independently of the number of cores in the system. One can observe that the obtained WCRT results are slightly more pessimistic than the ones derived with FTA, as the differences to the simulation observations are now larger. However, the pessimism of ATA compared to FTA is limited, reaching at maximum **1.03%** for task rspeed in the 3-core and the 4-core scenarios.

How can the comparable results of the FTA and ATA methods be explained? The methods produce the same WCRT for a core under analysis if the system under analysis operates close to the conservative case. E.g., for RR arbitration of DRAM, each core uses its assigned slot for accessing the resource while an access of the core under analysis is pending. Likewise with FCFS-based resource arbitration, the input buffer is always filled with $N-1$ requests upon the arrival of a request from the core under analysis; assuming that there are N cores in the system.

It appears that with the considered benchmarks most of the time the system shows such a behaviour and both methods produce comparable results (see also discussion in Sec. 7.3). However, in cases, where the "real" system exhibits larger intervals between the access requests from the different cores, the ATA method can become extremely pessimistic. In reality, access requests may be not so bursty, whereas with the ATA method requests are placed on the resource in a bursty fashion. This is due to (i) the construction of the interference curve α (sum of individual curves, Sec. 4.2), which provides a conservative upper bound on the interfering access requests, (ii) its over-approximation by the staircase function α^{st}, and (iii) the behavior of the access request generator (Sec. 6.2), which emits interfering requests non-deterministically over time, thus enabling the exploration of certain request streams that are bounded by α^{st} but may never be encountered in the real-time system.

The maximum difference between the ATA-derived WCRT and the corresponding simulation-observed WCRT is observed for task cacheb in the 5-core scenario and is equal to **19.22%**. As pointed out above, a similar picture could be expected also when comparing the FTA and ATA method, as FTA and simulation produce similar WCRT due to the code structuring. The results of Table 4 show, however, also that the difference (pessimism) is limited since in most cases, the ATA-derived WCRT are only up to **2.5%** greater than the corresponding simulation results.

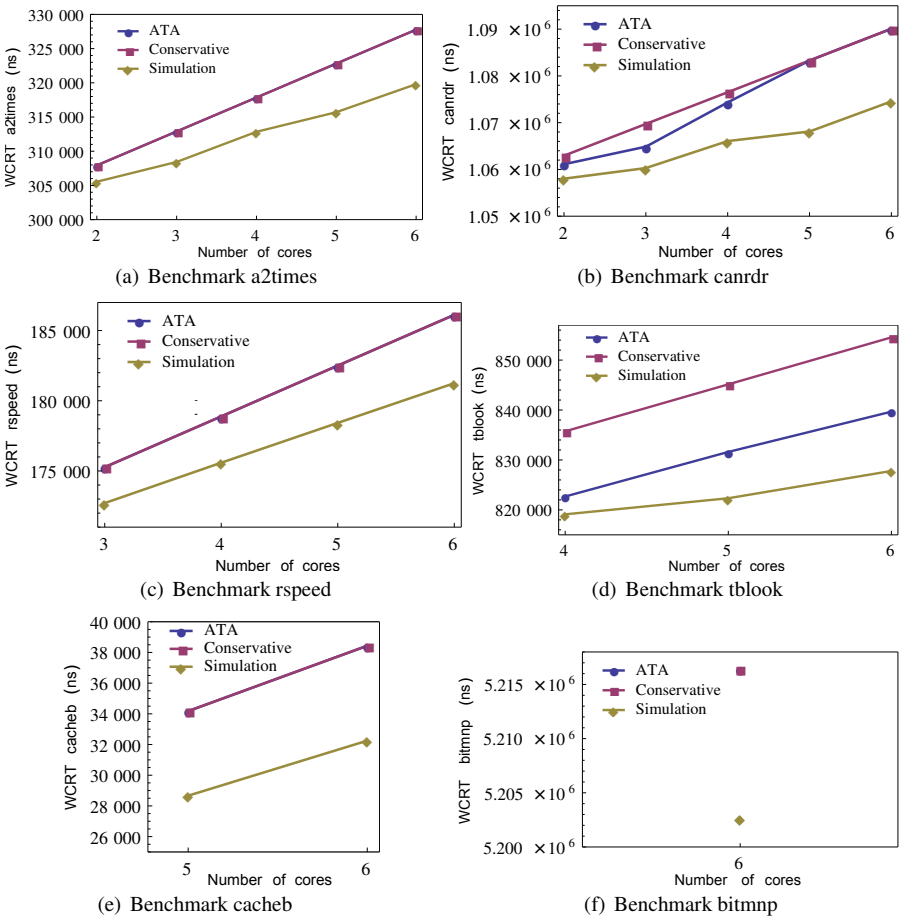


Fig. 7 WCRT of EEMBC benchmarks: ATA vs. Conservative Bound vs. Simulation for RR arbitration

This allows us to conclude that the gain in scalability, obtained by the abstraction of a part of the system with arrival curves, does not compromise the accuracy of the WCRT. The topic of accuracy is discussed further in the next section.

7.3 Evaluation results: ATA versus conservative WCRT estimation

Fig. 7(a)-7(f) illustrate the WCRT estimates of each EEMBC benchmark for each scenario (different number of concurrently executed benchmarks), as obtained by (i) the ATA analysis methodology, (ii) the simulation environment presented in Section 7.2, and (iii) a conservative WCRT estimation under RR resource arbitration, given by Eq. 11. The last estimation assumes that every single access of a core under analysis is delayed by *all* interfering cores in the system. This conservative, yet safe assumption enables modeling RR through a TDMA scheme, where each core has one

slot of length equal to C (access latency) in each arbitration cycle. Independently of whether the interfering cores emit accesses within a cycle, the corresponding slots cannot be used by the CUA, which has to wait for a whole cycle interval between any two successive accesses.

Based on the depicted results, the difference between the ATA estimates and the observed through simulation WCRT varies between 0.26% (bitmnp, 6 cores) and 19.22% (cacheb, 5 cores). Respectively, the difference between the ATA and the conservative WCRT estimates varies between 0% (benchmarks a2times, rspeed, cacheb, bitmnp) and 1.8% (tblook, 6 cores). The equality of the ATA and the conservative estimates for some benchmarks is a special case stemming from the nature of the considered benchmarks. Particularly, the six benchmarks (i) exhibit similar structure with a burst of hundreds of access requests at the beginning of their execution and (ii) are assumed to start synchronously (at time 0) among all cores. This results in massive interference on the shared memory at the beginning of each hyper-period, where the memory accessing (acquisition) phases of the benchmarks overlap. Consequently, for the considered case study the exhibited interference at runtime is closely described by the conservative assumption mentioned above. A larger deviation between the two estimates would be expected if the benchmarks started their execution after given phases, so that their acquisition phases would only partially (or not at all) overlap.

In general, for more complex arbitration policies than RR, like FlexRay [2], the derivation of a conservative bound is not trivial and can be only based on overly pessimistic assumptions (e.g., no access provided during the dynamic FlexRay segment) due to the complexity of modeling the state of the arbiter (see [13]). Under such arbitration scenarios, we expect the ATA WCRT to be more accurate than any conservative estimates[4]. The same applies also in cases, where the initial phases of the benchmarks (here, 0) are synchronized such that the interference on the memory path is reduced. If the interference arrival curves are computed by considering the initial phases, the ATA WCRT will be refined as opposed to the conservative bounds, which do not reflect this information.

8 Conclusion

8.1 Summary

The article presented a formal analysis and a simulation-based framework for the worst-case response time analysis of concurrently executing real-time tasks on resource-sharing multicore platforms. In such settings, cache misses at core level may stall the cores' local computation, while cache block fetching times from the main memory may vary extremely. This variation is due to the fact that fetch requests issued from other cores need to be processed at the shared main memory at the same time. Moreover, the resource arbitration scheme used for the main memory is commonly designed for speeding up the average case, rather than being timing

[4] Case studies with FlexRay are omitted here for brevity, but can be found in [13].

predictable. In a nutshell, the precise bounding of the accumulated waiting times experienced by a real-time task is extremely challenging, particularly when considering complex resource arbitration schemes beyond static time partitioning of accesses.

For adopting multicore architectures in hard real-time (control) applications, the use of formal and exhaustive analysis is inevitable in order to deliver tight safe bounds on the timing behavior. On the other hand, one must also ensure scalability to cope with systems of industrial interest. The proposed analysis technique achieves precision and scalability by employing timed model checking, explicit state-based modeling of the resource arbiter, aggregated access request curves, and the PREM approach. PREM yields a phase-type organization and modeling of software, where the tasks' read and write accesses to shared memory are performed only during acquisition and replication phases, while during execution phases, computation is performed without any cache misses. This reduces the non-determinism in the software and works for the scalability of the model checking driven approach.

PREM does not limit the applicability of the proposed analysis to a specific software system: given a set of tasks mapped to a core, one can derive a sequence of memory access phases and phases of pure local cache-served computations, and compute the worst-case execution time for these phases, ignoring memory access latencies. One then maps arbitrary tasks sets executing on a core to PREM by considering all possible interleaving of phases as appearing in the hyper-period of the task set.

8.2 Discussion

Analysis in the presence of timing anomalies. When it comes to multi-core architectures exhibiting timing anomalies the proposed approach needs to be considered with care.

- It only applies to timing monotone DRAM controllers, where a larger number of access requests placed on the DRAM controller by any core does not lead to faster servicing times of the requests.
- For processors, a timing anomaly commonly refers to the effect that a cache miss may yield a shorter worst-case execution time as opposed to a cache hit for the very same instruction / data item. Potential reasons for this are the out-of-order execution of the pipeline, branch-prediction and program pre-fetching enabled by a cache hit and suppressed with a cache miss and the cache replacement policy. Here one has to consider two cases: (1) Such effects are absent in so called timing composable platforms that are commonly considered for deploying of real-time systems, e. g. the Multi-Purpose Processor Array of Kalray [5]. In addition they might be avoidable by code modification techniques[5]; (2) Allowing the bounds on the number of cache misses and worst-case execution times of phases to originate from different program execution paths, breaks up the dependency between

[5] The authors of [24] propose a method which avoids timing anomalies due to out-of-order (micro)-instruction execution. It is empirical shown that such efforts might not utterly compromise the performance of the original software.

worst-case execution time and number of memory accesses, at the cost of tightness. This renders core-related timing anomalies insignificant for the proposed approach. However, in this case the core-local worst-case execution time analysis under timing anomalies and unknown memory access times is a pre-requisite to the proposed analysis [24, 33]. If this can not be performed, one would have to use a single integrated analysis framework which is truly beyond the scope of the presented approach.

Adequacy of abstraction of core-wise memory access patterns. PREM and the used abstractions, particularly the replacement of individual models of cores with memory access request curves, allow us to achieve scalability, crucial for any state-based modeling and analysis. For settings, where the core-individual memory access frequencies are not overly high, this abstraction does not introduce too much pessimism. This is demonstrated in the empirical evaluation, as in most cases the obtained worst-case response times are not too far away from the simulation results. However, for settings, where cores do not issue memory accesses at the same time, e. g. due to inter-core software synchronization mechanisms, the proposed summing of core-local access curves may introduce a significant amount of pessimism. This is because, this way we model a joint access of all cores to the main memory, which in reality may never occur. This explains why the method may produce worst-case estimates which are close to the conservative, trivial bound, particularly for settings with high memory access frequencies. By taking inter-core synchronization into account when computing and aggregating the memory access curves, one could compensate this and re-gain precision. How to do this, however, is left to future research.

Achieving precision when analyzing *arbitrarily complex* resource access schemes in multicores has been our main motivation and the lack of scalability the most difficult hurdle to overcome. In this respect, the presented approach improves significantly upon previous results: it handles arbitrarily complex resource arbitration mechanisms, i. e. it is not tied to a specific scheme and it avoids (extremely) pessimistic over-approximation of resource arbiters[6]. The strength of the proposed framework is highlighted in those cases, where any conservative, but trivial worst-case bounding becomes extremely pessimistic due to the complexity of the resource arbiter. In summary, the proposed framework and abstractions balance well precision of the obtained worst-case response times on the one hand, and scalability on the other hand. This has also been confirmed through the benchmarking of the proposed techniques.

Acknowledgements

We thank the anonymous reviewers for the valueable remarks and the resulting improvements of the article.
This work has received funding from the European Union Seventh Framework

[6] E. g. the dynamic programming approach of [31] is tailored to Round-Robin based resource access schemes. [36] exploits static time division to soundly over-approximate Round Robin based resource arbitration.

Programme (FP7/2007-2013) project CERTAINTY under grant agreement number 288175 and from the NSERC under Discovery Grant 402369-2011. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

References

1. EEMBC 1.1 Embedded Benchmark Suite. http://www.eembc.org/benchmark/automotive_sl.php.
2. Flexray communications system protocol specification, version 2.1, revision a. http://www.flexray.com/.
3. K. Altisen and M. Moy. ac2lus: Bringing SMT-solving and abstract interpretation techniques to real-time calculus through the synchronous language Lustre. In *22nd Euromicro Conference on Real-Time Systems (ECRTS)*, Brussels, Belgium, Jully 2010.
4. R. Alur and D. L. Dill. Automata For Modeling Real-Time Systems. In M. Paterson, editor, *Proc. of the 17th International Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *LNCS*, pages 322–335. Springer, 1990.
5. P. Aubry, P.-E. Beaucamps, F. Blanc, B. Bodin, S. Carpov, L. Cudennec, V. David, P. Dore, P. Dubrulle, B. D. de Dinechin, F. Galea, T. Goubier, M. Harrand, S. Jones, J.-D. Lesage, S. Louise, N. M. Chaisemartin, T. H. Nguyen, X. Raynaud, and R. Sirdey. Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. *Procedia Computer Science*, 18(0):1624 – 1633, 2013. 2013 International Conference on Computational Science.
6. S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo. Memory-aware scheduling of multicore task sets for real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*, pages 300–309, 2012.
7. G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.
8. J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2004.
9. N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.
10. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In A. Hu and M. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer Berlin Heidelberg, 1998.
11. S. Chakraborty, Y. Liu, N. Stoimenov, L. Thiele, and E. Wandeler. Interface-based rate analysis of embedded systems. In *RTSS 2006*, pages 25–34, 2006.
12. D. Dasari, B. Anderssom, V. Nelis, S. Petters, A. Easwaran, and J. Lee. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1068 –1075, Nov. 2011.
13. G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele. Timed model checking with abstractions: towards worst-case response time analysis in resource-sharing manycore systems. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '12, pages 63–72, New York, NY, USA, 2012. ACM.
14. N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, pages 245–254, New York, NY, USA, 2009. ACM.
15. A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards WCET Analysis of Multicore Architectures Using UPPAAL. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, pages 101–112, 2010.
16. R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis-The SymTA/S Approach. *IEEE Proceedings-Computers and Digital Techniques*, 152(2):148–166, 2005.

17. A. A. Informatik. http://www.absint.com/.
18. T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore wcet analysis through tdma offset bounds. In *23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3 –12, 2011.
19. K. Lampka, S. Perathoner, and L. Thiele. Analytic real-time analysis and timed automata: A hybrid method for analyzing embedded real-time systems. In *8th ACM & IEEE International conference on Embedded software, EMSOFT 2009*, pages 107–116, Grenoble, France, 2009. ACM.
20. K. Lampka, S. Perathoner, and L. Thiele. Analytic real-time analysis and timed automata: A hybrid methodology for the performance analysis of embedded real-time systems. *Design Automation for Embedded Systems*, 14(3):193–227, 2010.
21. K. Lampka, S. Perathoner, and L. Thiele. Component-based system design: analytic real-time interfaces for state-based component implementations. *International Journal on Software Tools for Technology Transfer*, pages 1–16, 2012.
22. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. of the International Symposium of Code Generation and Optimization*, San Jose, CA, USA, Mar 2004.
23. J.-Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag, Berlin, Heidelberg, 2001.
24. T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proc. of the 20th Real-Time Systems Symposium*, pages 12–21, 1999.
25. M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *IEEE Real-Time Systems Symposium 2010*, pages 339–349. IEEE Computer Society, 2010.
26. R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2013.
27. K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006), 9-13 December 2006, Orlando, Florida, USA*, pages 208–222. IEEE Computer Society, 2006.
28. C. Norström, A. Wall, and W. Yi. Timed Automata as Task Models for Event-Driven Systems. In *RTCSA '99*, page 182, Washington, DC, USA, 1999. IEEE Computer Society.
29. R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, pages 269–279. IEEE Computer Society, 2011.
30. R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In *Real-Time Systems Symposium*, pages 221–231, 2008.
31. R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Design, Automation, Test in Europe Conference (DATE)*, pages 741–746, 2010.
32. S. Perathoner, K. Lampka, and L. Thiele. Composing heterogeneous components for system-wide performance analysis. In *Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14-18, 2011*, pages 842–847. IEEE, 2011.
33. J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In F. Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*, OASICS. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.
34. 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 *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 49–60, 2007.
35. S. Schliecker, M. Negrean, and R. Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Design, Automation, Test in Europe Conference (DATE)*, pages 759–764, 2010.
36. A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing Analysis for TDMA Arbitration in Resource Sharing Systems. In *Proceedings of RTAS'10*, 2010.
37. A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 213–222, 2011.

38. A. Simalatsar, Y. Ramadian, K. Lampka, S. Perathoner, R. Passerone, and L. Thiele. Enabling parametric feasibility analysis in real-time calculus driven performance evaluation. In *Proc. of the 14th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 155–164. ACM, 2011.
39. L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. Intl. Symposium on Circuits and Systems*, volume 4, pages 101–104, 2000.
40. G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems Journal*, 48(6):681–715, Nov 2012.