Timed Model Checking with Abstractions: Towards Worst-Case Response Time Analysis in Resource-Sharing Manycore Systems

Georgia Giannopoulou* Kai Lampka† Nikolay Stoimenov* Lothar Thiele*
*Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland
†Information Technology Department, Uppsala University, Sweden
{giannopoulou, stoimenov, thiele}@tik.ee.ethz.ch lampka@it.uu.se

ABSTRACT

Multicore architectures are increasingly used nowadays in embedded real-time systems. Parallel execution of tasks feigns the possibility of a massive increase in performance. However, this is usually not achieved because of contention on shared resources. Concurrently executing tasks mutually block their accesses to the shared resource, causing non-deterministic delays. Timing analysis of tasks in such systems is then far from trivial. Recently, several analytic methods have been proposed for this purpose, however, they cannot model complex arbitration schemes such as FlexRay which is a common bus arbitration protocol in the automotive industry. This paper considers real-time tasks composed of superblocks, i. e., sequences of computation and resource accessing phases. Resource accesses such as accesses to memories and caches are synchronous, i. e., they cause execution on the processing core to stall until the access is served. For such systems, the paper presents a state-based modeling and analysis approach based on Timed Automata which can model accurately arbitration schemes of any complexity. Based on it, we compute safe bounds on the worst-case response times of tasks. The scalability of the approach is increased significantly by abstracting several cores and their tasks with one arrival curve, which represents their resource accesses and computation times. This curve is then incorporated into the Timed Automata model of the system. The accuracy and scalability of the approach are evaluated with a real-world application from the automotive industry and benchmark applications.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; C.4 [**Performance of systems**]: Modeling techniques

Keywords

multi-core systems, worst-case response time analysis, resource contention

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'12, October 7–12, 2012, Tampere, Finland.
Copyright 2012 ACM 978-1-4503-1425-1/12/09 ...$15.00.

1. INTRODUCTION

Multicore systems have become increasingly popular as they allow increase in performance by exploiting parallelism in hardware and software without sacrificing too much on power consumption and cost. However, the reduction in cost is achieved by sharing resources between the processing cores. Such shared resources can be for example buses, caches, scratchpad memories, main memories, and DMA.

In safety-critical embedded systems, such as controllers in the Automotive Open System Architecture (AutoSAR) [1], accesses to the shared resources can be non-periodic and bursty, which means that the system may miss its real-time deadlines. Therefore, a designer needs to consider the interference due to contention on the shared resources in order to verify the real-time properties of the system. At the same time, the interference-induced delays need to be tightly bounded to avoid an extreme over-provisioning of the platform.

However, performing timing analysis for such systems is quite challenging because the number of possible interleavings of resource accesses from the different processing cores can be very 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 [24] and SymTA/S [11] 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 of a processing core until the access is completed, i.e. an access increases the tasks' WCRT. Once an access starts being served, it cannot be preempted by other accesses. Bounding the waiting time for an access under these assumptions is challenging as one has to take into account the currently issued accesses from other cores and the state of the resource sharing arbiter. In this paper, we deal with the second case of synchronous accesses.

Schliecker et al. [20], 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 maximum and the minimum numbers of resource accesses in particular time windows are assumed to be known. The arbiter to the shared resource uses a first-come first-served (FCFS) scheduling scheme and tasks are scheduled according to fixed priority. The authors evaluate the approach with a system with two processing cores. In [15] Negrean et al.

consider the multiprocessor priority ceiling protocol where tasks have globally assigned priorities, but similarly use a system with two processing cores. None of the above results consider complex scheduling policies such as FlexRay [4] or systems with high number of processing cores as analyzed in this paper.

Recent results in [18, 21, 23] have proposed methods for WCRT analysis in multicore systems where the shared resource is arbitrated according to FCFS, Round Robin (RR), Time Division Multiple Access (TDMA) or a hybrid time-/event-driven strategy, the latter being a combination of TDMA and FCFS/RR. The presented analysis, however, uses over-approximations which result in overly pessimistic results, particularly in cases of state-based arbitration mechanisms, like FCFS or RR. This shortcoming is of concern, as industrial standards like the FlexRay [4] bus protocol explicitly exploit state-dependent behaviors. In this paper, we rely on model checking techniques in order to model accurately the behavior of such state-dependent arbiters.

Note that the FlexRay protocol is considered in our work as an example of a complex state-based arbitration policy, even though it is not originally designated for e. g., bus arbitration in shared-memory distributed systems. FlexRay has been analyzed by Pop et al. [19] and by Chokshi et al. [7] but these approaches deal only with the asynchronous case of resource accesses. On the other hand, FlexRay with synchronous accesses has never been modeled with analytic approaches. In [23], Schranzhofer et al. have modeled a hybrid arbitration mechanism as a combination of a static (TDMA) and a dynamic segment, the latter behaving according to FCFS or RR. Model checking enables us, however, to model and analyze FlexRay with synchronous accesses for the first time.

Model checking techniques have been applied for timing analysis in resource-sharing multicore systems in [14, 10]. The methods deal accurately with complex arbitration schemes, however, none of them can scale efficiently beyond two cores. In order to alleviate the state-explosion problem which is inherent to model checking techniques, in this paper we combine model checking with several abstractions. First being that tasks are composed of superblocks. Second being that some of the processing cores can be substituted by simpler models that represent only their resource accesses.

The superblock model for structuring real-time tasks has been proposed by industry as part of the EU sponsored project Predator [3], since it fits very well signal-processing and control applications. The model assumes that tasks are composed of sequentially executed superblocks for which the minimum/maximum number of resource accesses and execution times are known. The model has been extensively used in several methods [18, 21, 23] for deriving worst-case response times of tasks in resource-shared multicore systems with synchronous accesses. Different variants of the model are compared in [22]. They differ mostly in that superblocks may have phases where resource accesses are not required (computation-only phases). Such phase structure can be actually enforced by a compiler as shown in [16]. Additionally, [8] shows how this model of resource accessing can be mapped to an AutoSAR system for automotive applications.

Arrival curves as known from Network and Real-Time Calculus [24] are used to bound the maximum number of events arriving in any time interval of any given length. Several methods [15, 18, 20] have utilized them before to represent the maximum number of resource accesses from a task. The novelty of this paper is their integration into a Timed Automata model of the system in order to combine accurate modeling of complex arbitration schemes with analysis scalability.

It should be noted that we consider only hardware platforms without timing anomalies, such as the fully timing compositional architecture proposed by Wilhelm et al. [25], and we assume that a task partitioning is pre-defined, i.e. tasks are mapped to dedicated processing cores.

The contribution of this paper is summarized as follows:

- For the proposed system model, we introduce a state-based WCRT analysis approach. Timed automata (TA) are used to model (a) concurrent execution of processing cores and their tasks and (b) resource access arbitration according to an event-driven (FCFS, RR), time-driven (TDMA), hybrid (FlexRay) policy or any other policy of arbitrary complexity. The Uppaal model checker is then used to compute the exact WCRT of each task in the system.
- To increase the scalability of the approach, an abstraction based on arrival curves is introduced. Tight curves that bound the number of resource accesses in the time-interval domain are derived for each core from the actual accesses specified in the superblocks of the tasks mapped on the core. Using the interfaces between Real-Time Calculus and TA presented in [12], the TA model of the system is reduced by replacing the models of several processing cores with a single model that can generate non-deterministic streams of access requests according to the arrival curves of the abstracted cores.
- The accuracy and efficiency of the approach are demonstrated using a set of embedded benchmark applications (EEMBC) and a real-world application from the automotive industry. The WCRT bounds obtained with the proposed method are compared to those of state-of-the-art analytic approaches.

In the remainder of the paper, Section 2 shortly introduces some of the necessary theory on TA and Real-Time Calculus. Section 3 defines the considered system models. Section 4 introduces the new state-based analysis method, addressing explicitly the challenge of scalability. Finally, Section 5 presents the empirical evaluation of the proposed approach and Section 6 concludes the paper.

2. BACKGROUND THEORY

In this section we briefly introduce some important concepts from the theories of Timed Automata and Real-Time Calculus which will be needed throughout the paper. For more details, the reader is referred to the respective literature.

2.1 Timed Automata

Let C be a set of clocks and let $ClockCons$ be a set of constraints on these clocks. With Timed Automata (TA) [5] the clock constraints are conjunctions, disjunctions and negations of atomic (clock) guards of the form $x \bowtie n, x \in C, n \in \mathbb{N}_0$ with $\bowtie \in \{<, \leq, >, \geq, =\}$. A TA $\mathcal{T}$ is then defined as a tuple $\mathcal{T} := (Loc, Loc_0, Act, C, \hookrightarrow, I)$ where Loc is a finite set of locations, $Loc_0 \subseteq Loc$ is the set of initial locations, Act is a (finite) set of action (or edge) labels, C is the finite set of clocks, $\hookrightarrow \subseteq Loc \times ClockCons(C) \times Act \times 2^C \times Loc$ is an edge relation and $I : Loc \rightarrow ClockCons(C)$ is a mapping of locations to clock constraints, where the latter are referred to as location invariants.

Let the active locations be the locations the TA currently resides in. The operational semantics associated with a TA can be informally characterized as follows:

- *Delay transitions.* As long as the location invariants of the active location(s) are not violated, time may progress, where all clocks increase at the same speed.

- *Edge executions.* The traversal of an edge (one at a time) potentially changes the set of active locations; self-loops are possible. The traversal, or "edge execution" is instantaneous and possible as long as the source location of the edge is marked active and the guard of the edge evaluates to true. Upon edge executions, clocks can be reset.

This operational semantics yields that for a TA one may observe infinitely many different behaviors. This is because edge executions may occur at any time, namely as long as the edge guard evaluates to true. However, with the concept of clock regions [5] it is possible to capture all possible behaviors in a finite state graph, such that timed reachability is decidable. In fact, modern timed model checkers incorporate *clock zones* [13] as they often result in a coarser partitioning of the clock evaluation space in comparison to the original definition of clock regions.

In this paper we will exploit the timed model checker Uppaal [6], which implements timed safety automata extended with variables. Similarly to clocks, variables can be used within guards of edges and location invariants. Upon an edge execution, a variable can be updated. The used variable values must build a finite set, which, however, does not need to be known beforehand, i. e., it can be constructed on-the-fly upon the state space traversal. For conciseness, we omit further details, and refer the interested reader to the literature [5, 26, 6].

2.2 Real-time Calculus

Real-time calculus (RTC) [24] is a compositional methodology for system-level performance analysis of distributed real-time systems. We briefly recapitulate the basic concepts that are used in this paper.

In the context of real-time systems design, the timing behavior of event streams is usually characterized by real-time traffic models. Examples of such typically used models are periodic, sporadic, periodic with jitter, and periodic with burst. RTC provides an alternative characterization of streams: a pair (α^l, α^u) of arrival curves bounds the number of events seen on the stream for any interval of length $\Delta \in [0, \infty)$. Let $R(t)$ be a stream's cumulative counting function which reports the actual event arrivals for the time interval $[0, t)$. The upper and lower arrival curves bound $R(t)$, i. e., the possible event arrivals in the time interval domain, as follows:

$$\alpha^l(t-q) \leq R(t) - R(q) \leq \alpha^u(t-q) \text{ with } 0 \leq q \leq t. \quad (1)$$

In this work we restrict our attention to the case of *discrete amounts* of events. In RTC, such scenarios are modeled as staircase curves. In particular, the upper arrival curve $\alpha^u(\Delta)$ in this case is defined as the staircase function:

$$\alpha^{st}(\Delta) := B + \left\lfloor \frac{\Delta}{\delta} \right\rfloor \cdot s \quad (2)$$

Parameter $B > 0$ models the burst capacity, namely it is the number of events that can arrive at the same time instant in the stream bounded from above by $\alpha^{st}(\Delta)$. Parameters δ and s are related to the maximum long-term arrival rate of events in the stream and the step size (y-offset), respectively.

The timing behavior of streams modeled as staircase arrival curves (Eq. 2) can be correctly and completely modeled with timed automata models as described in [12].

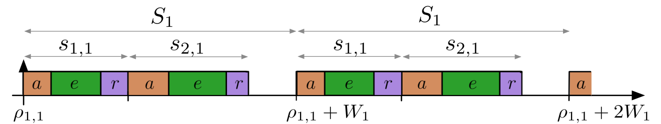


Figure 1: Dedicated sequential superblock model

3. SYSTEM MODEL

This section presents the real-time task model that was considered in our analysis and introduces the arbitration mechanisms for coordinating access to the shared resource.

3.1 Tasks and Processing Cores

A system consists of multiple *processing cores* $p_j \in P$. The cores in P execute independent tasks but can access a common resource, such as an interconnection bus to a shared memory. We assume a given task partitioning, in which a task set T_j is assigned to a predefined processing core $p_j \in P$. The *tasks* in T_j are specified by a sequence of *superblocks* S_j, which are non-preemptable execution units with known lower/upper bounds on their computation time and the number of resource accesses that they may require.

For each core p_j, we assume a static schedule of the assigned tasks, which is repeated periodically with period W_j. Periods of the processing cycles may be different among the cores. In the first processing cycle, the first superblock $s_{1,j}$ of the task set T_j starts executing at time $\rho_{1,j}$. The static schedule provides an order of the superblock set S_j[1]. Therefore superblocks in S_j are repeatedly executed in $[\rho_{1,j}, \rho_{1,j} + W_j)$, $[\rho_{1,j} + W_j, \rho_{1,j} + 2W_j)$, and so forth, with each superblock $s_{i+1,j}$ being triggered upon completion of its predecessor, $s_{i,j}$[2](see, e. g., execution of superblock sequence $S_1 = \{s_{1,1}, s_{2,1}\}$ on core p_1, Figure 1). The starting times of processing cycles on different cores may be synchronized, i. e., the first superblock in all first processing cycles starts at time 0 ($\rho_{1,j} = 0, \forall j$), or non-synchronized, i. e., $\rho_{1,j} \in [0, W_j)$.

In order to reduce non-determinism w. r. t. the occurrence of access requests, superblocks are separated into three phases, known as *acquisition*, *execution*, and *replication* which are denoted with a, e, and r in Figure 1. The model represents that a superblock reads the required data during the acquisition phase and writes back the modified/new data in the replication phase, after computations in 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.

The dedicated superblock structure is the first abstraction proposed in this paper, since the restriction of resource accesses to dedicated phases leads to increased predictability when analyzing the system's timing behavior. In particular, Schranzhofer et al. have shown in [22] that the schedulability of the dedicated sequential superblock model dominates that of other models, in which accesses may occur any time during computation.

As a result of the discussed structure, a superblock is fully defined by the following parameters: (a) the minimum and maximum number of access requests during its acquisition and replication phase, $\mu_{i,j}^{min,\{a,r\}}$ and $\mu_{i,j}^{max,\{a,r\}}$, and (b) the minimum and maximum computation time during its

[1]Mixing of superblocks of different tasks, i. e., preemption on task level, is possible.

[2]Idle intervals between successive superblocks may also exist.

execution phase, $exec_{i,j}^{min}$ and $exec_{i,j}^{max}$. Note that the necessary computation to initiate the accesses during the dedicated phases is considered negligible compared to the time required for their completion and hence, can be ignored. For a superblock with logical branches, the above numbers might be overestimated, but the worst-case execution time will be safely bounded. We assume that the access request parameters can be derived either by profiling and measurement for the case of soft real-time systems, as shown in [17], or by applying static analysis when hard safe bounds are necessary.

3.2 Shared Resource and Arbiter

In the considered systems, task execution is suspended every time an access request is issued, until the latter is completely processed by the resource arbiter. Once the arbiter grants access of a request, the accessing time is (bounded by) C time units. That is, if a superblock $s_{i,j}$ could access the shared resource at any time, its worst-case execution time would be $exec_{i,j}^{max} + (\mu_{i,j}^{max,a} + \mu_{i,j}^{max,r}) \cdot C$. It is assumed that access can be granted to at most one request at a time and that processing of an ongoing access cannot be preempted. Once a pending access request is served, either task execution on the corresponding core is resumed by performing computations or a new access request is issued or the core remains idle until the start of a new processing cycle (or the next superblock). Access to the shared resource is granted by a dedicated arbiter. The implemented arbitration policy can be time-driven, e.g., TDMA, event-driven, e.g., FCFS or RR or hybrid, e.g., the FlexRay bus protocol. A more detailed discussion on the possible arbitration schemes follows.

3.2.1 TDMA Arbiter

In a TDMA arbitration scheme, access to the shared resource is statically organized by assigning time slots to each core. That is, access to the resource is partitioned over time and only a single core can acquire it at any moment. TDMA arbitration policies are widely used in timing- and safety-critical applications to increase timing predictability and alleviate schedulability analysis, since they eliminate mutual interferences of the tasks through their time isolation.

A TDMA arbiter uses a predefined cycle of fixed length, which is specified as a sequence of time slots. The time slots can be of variable lengths and there is at least one slot for each core $p_j \in P$ in every schedule. An access request issued by the current task of core p_j during the i-th time slot of the TDMA cycle is enabled immediately if the latter slot is assigned to core p_j and the remaining time of the slot is enough to accommodate processing of the new access. Requests that arrive "too late" have to wait until the next allocated slot. To provide meaningful results, we assume that all slots in a TDMA schedule are at least of length C.

3.2.2 RR Arbiter

RR-based arbitration can be seen as a dynamic version of TDMA with a varying arbitration cycle. This is because the unused slots of the cycle are skipped whenever the respective cores do not need to access the shared resource. To implement this behavior, the RR arbiter checks repeatedly (circularly) all cores in P, starting with the first identifier, p_1, up to the last one, p_N. If the core with the currently considered identifier p_i has a pending request, access is granted to it immediately. Otherwise, the arbiter checks the next identifier, and so forth.

3.2.3 FCFS Arbiter

The resource arbiter, in this case, is responsible for maintaining a first-in first-out (FIFO) queue with the identifiers of the cores which have a pending access request. Access is granted based on the time ordering of their occurrence, i. e., the oldest request is served first. This scheme guarantees fairness given that, if core p_i issues an access request before core p_j, p_i's request will be served at an earlier point in time, without considering any priorities between the two cores.

3.2.4 FlexRay Arbiter

The FlexRay protocol [4] has been introduced by a large consortium of automotive manufacturers and suppliers. It enables sharing of a resource (usually interconnection bus among the processing cores of an automotive system), featuring both time- and event-driven arbitration.

In the FlexRay protocol, a periodically repeated arbitration cycle is composed of a static (ST) and a dynamic (DYN) segment. The ST segment uses a generalized TDMA arbitration scheme, whereas the DYN segment applies a flexible TDMA-based approach. The lengths of the two segments may not be equal, but they are fixed across the arbitration cycles. Both segments are defined as sequences of time slots. The ST segment includes a fixed number of slots, with constant and equal lengths, d. Each slot is assigned to a particular core and one or more access requests from this core can be served within its duration. The DYN segment is defined as a sequence of *minislots* of equal length, $\ell \ll d$. The actual duration of a slot depends on whether access to the shared resource is requested by the corresponding core or not: if no access is to be performed, then the slot has a very small length (*minislot length*, ℓ). Otherwise, it is resized to enable the successful processing of the access (*access length*, here equal to C). To obtain reasonable results, we assume that each ST slot as well as the DYN segment are at least equal to C. The assignment of ST or DYN (mini)slots to the processing cores is static. However, since the introduction of FlexRay 2.0, cycle multiplexing has become also possible for the DYN segment, i. e., some minislots may be assigned to different cores in different cycles, resulting in more than one alternating arbitration schedule.

The above description implies that in the ST part of FlexRay, like in TDMA, interference can be neglected due to isolation. In the DYN part, however, the actual delay of an access is interference-dependent, which makes it difficult to analyze without a state-based approach. The accurate modeling and analysis of such an arbitration policy for the case of synchronous resource accessing has been one of the major motives of our developed approach.

4. TIMING ANALYSIS USING MODEL CHECKING

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 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 required verification effort.

4.1 Multicore System as a TA Network

4.1.1 Modeling Concurrent Execution

The parallel execution of superblock sets S_j on the system's cores is modeled using instances of two TA for each core, mentioned in the following as *Scheduler* and *Superblock*, and depicted in Figure 2. In a system with N cores

and a total of M superblocks executed on them, $(N + M)$ TA are needed to model execution.

The *Scheduler* TA (Figure 2(b)) specifies the scheduling mechanism of each core p_j. It is responsible for activating the superblocks in S_j sequentially, according to the predefined execution order (static schedule), which is repeated every W_j time units. Whenever a new superblock must be scheduled, the *Scheduler* emits a `start[sid]` signal. Due to the underlying composition mechanism, this yields a synchronization between the *Scheduler* and the respective *Superblock* instance. When the superblock's execution is completed, the *Scheduler* receives a `finish[sid]` signal. If this superblock was not the last of the current processing cycle (condition checked through user-defined function `last_sb()`), the *Scheduler* triggers the next superblock. Otherwise, it moves to location `EndOfPeriod`, where it lets time pass until the processing cycle's period is reached.

A *Superblock* TA (Figure 2(a)) 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 processed. Access requests are issued through channel `access[pid]`, whereas notification of their completion is received by the arbiter through channel `accessed[pid]` (see 'loop transitions' over `Acq` in Figure 2(a)). 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` (successive resource accesses).

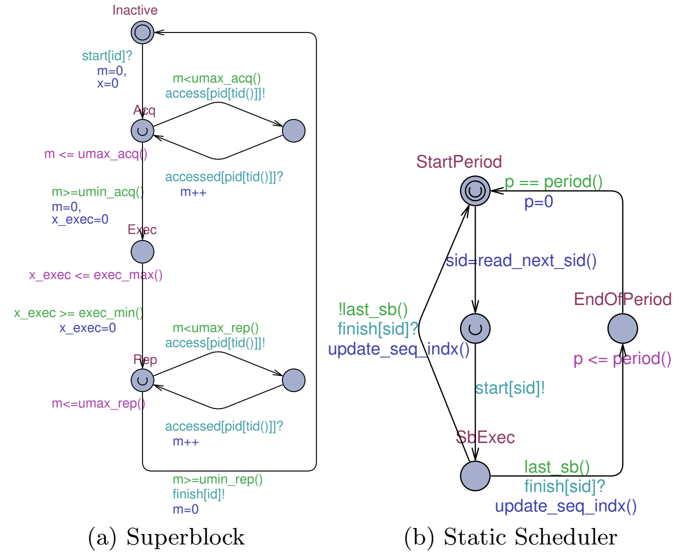
For the case of a single superblock in S_j, clock `x` is used to measure its total execution time. Checking the maximum value of clock `x` while the TA resides in its 'final' location, i. e., `Rep`, allows to obtain the WCRT of the whole superblock. With Uppaal this is done by specifying the lowest value of `WCRT`, for which the safety property `A[] Superblock(i).Rep imply Superblock(i).x <= WCRT` holds[3]. This way we ensure that for all reachable states, the value of superblock s_i's clock `x` is never greater than `WCRT`. To find the lowest `WCRT` satisfying the previous query, binary search can be applied. Upon termination, the binary search will deliver a safe and tight bound of the superblock's WCRT. Similarly, we can compute a WCRT bound on a sequence S_j (a task) by adapting the TA of Figure 2(a) to model more than 3 phases.

4.1.2 Modeling Resource Arbitration

The TA modeling the four suggested arbitration policies of Sec. 3.2 are depicted in Figure 3. Depending on the implemented policy, the respective model is included in the TA network of our system.

The *FCFS* and the *RR Arbiter* share a similar TA, depicted in Figure 3(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

[3] Query `sup{Superblock(i).Rep}:Superblock(i).x` could be also used for the same purpose.



(a) Superblock (b) Static Scheduler

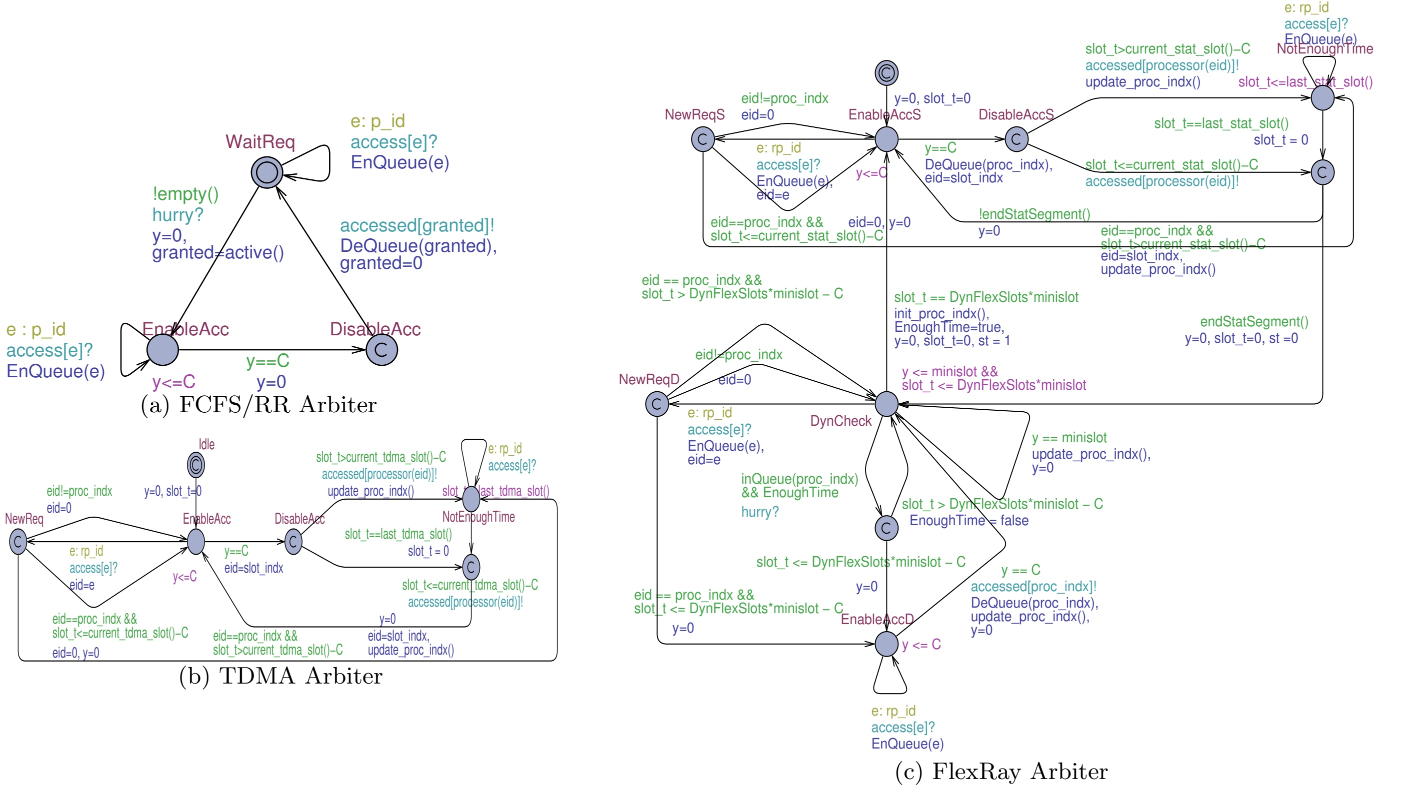
Figure 2: Superblock and Static Scheduler TA

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 time C, the access is completed, so 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 Figure 3(b) implements the predefined TDMA arbitration cycle, in which each core has one or more, sequential or non-deterministically 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.

The *FlexRay Arbiter* in Figure 3(c) is substantially an extension over the TDMA arbiter. This extension models the DYN segment of the FlexRay arbitration cycle that is executed after each ST segment. Once the ST segment is completed (`EndStatSegment()`), the arbiter checks if the core assigned to the first DYN minislot has a pending request. If this is true (`inQueue(proc_indx)`), the DYN minislot is resized to `C` time units to accommodate the access. Otherwise, the arbiter waits until expiration of the minislot and then, it checks for pending requests from the core assigned to the next minislot. This procedure is repeated until the DYN segment expires. According to this model, during the FlexRay DYN segment, a new access request from core `eid` is served



(a) FCFS/RR Arbiter

(b) TDMA Arbiter

(c) FlexRay Arbiter

Figure 3: TA representation of arbitration mechanisms

immediately on condition that (a) the current minislot is assigned to `eid` and (b) the remaining time until the DYN segment expires is at least equal to C, so that the DYN segment cannot interfere with the upcoming ST segment. If a condition is not fulfilled, then serving `eid`'s access request is postponed until its following ST or DYN (mini)slot.

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.

4.2 Reducing the Number of TA-based Component Models

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 superblock sequence 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. Particularly, the more the processing cores, the more the required clocks for modeling execution on them, which leads gradually to state space explosion. In the following sections, we propose safe abstractions for achieving a better analysis scalability.

In the proposed abstractions, only one processing core (core under analysis, CUA) is considered at a time, while all remaining cores, which compete against it for access to the shared resource, are abstracted away (not ignored). To model the access requests of abstracted cores we use arrival curves (Sec. 2.2). This way an arrival curve capturing the aggregate interference pattern of the abstracted cores can be constructed and then, modeled using TA. Eventually, the network of TA that models our system will include only 1 *Scheduler*, M_i *Superblock* ($M_i = |S_i|$: number of superblocks executing on CUA p_i), 1 *Arbiter* and 2 *Request Generator* TA, i. e., the number of TA components will **not** depend on the number of abstracted cores.

4.2.1 *Abstract representation of access request patterns*

Representation of the access pattern of a core's periodically executed superblock sequence, using an arrival curve in the time-interval domain (as known from Real-Time Calculus), was introduced by Pellizzoni et al. in [18]. Their method for the arrival curve construction was based on the assumptions that (a) superblocks are general, i. e., computation and accesses can be performed any time, and (b) access requests from the same core can be buffered on the arbiter. The following part describes an extension to that method, namely it presents how to compute an upper arrival curve α_j which bounds the amount of access requests that a core p_j can issue in any time interval, while accounting for the dedicated superblock structure and the synchronous access requests that are assumed in our system model.

To derive this arrival curve for a core p_j, the core is considered in isolation, i. e., as if it had immediate exclusive access to the shared resource. For the curve construction, the following steps are necessary: (i) computation of all possible ordered phase subsets of the core's superblock sequence S_j, (ii) computation of the feasible time windows for each subset and the maximum number of access requests that can be issued during them, and (iii) construction of the arrival curve as a periodic function. These steps are elaborated briefly in the following. For a more detailed presentation and examples, the reader is referred to technical report [9].

(i) We first consider two subsequent instances of S_j (to account for the transition phase between successive processing cycles on p_j) and we specify all possible subsets of phases that can be executed during these instances. To do this, we reduce the superblocks in S_j to their three constituent phases, i. e., each superblock is specified as $s_{i,j} = \{f_{3\cdot(i-1)+1,j}, f_{3\cdot(i-1)+2,j}, f_{3\cdot(i-1)+3,j}\}$. Then, the superblock sequence $\{SjSj\} = \{s_{1,j}...s_{|Sj|,j}, s_{1,j}...s_{|Sj|,j}\}$ can be redefined as the phase sequence $\{f_{1,j}...f_{3\cdot|Sj|,j}, f_{1,j}...f_{3\cdot|Sj|,j}\}$, which yields a set of $\frac{3\cdot|Sj|(9\cdot|S_j|+1)}{2}$ unique ordered subsets of superblock phases. Each of these ordered subsets, denoted as $F_{m,d}$, is described by the index m of the first phase it contains and the distance d to the last phase, such that:

$$F_{m,d} = \{f_{m,j}, ..., f_{m+d,j}\} \quad , \quad \forall d \in [0...6 \cdot |S_j| - 1], \quad \forall m \in [1...3 \cdot |S_j|] \tag{3}$$

where $m + d \leq 6 \cdot |S_j|$.

(ii) Every phase subset $F_{m,d}$ can be associated with several time windows, during which new access requests are considered. We can compute these time windows such that they are as short as possible while they contain as many access requests as possible. This way the windows represent the worst-case interference that a phase subset can cause to any other task attempting to access the shared resource within their duration. The time windows $\Delta^{x,y}_{m,d,k}$ of $F_{m,d}$ and the respective numbers of access requests $\gamma^{y}_{m,d,k}$ during them are computed in Eq. 4 and 5, for all $k \in [1, \mu^{max}_{m+d,j}]$, such that each time window differs from the others by at least one access request. The superscripts $x = max$ or $x = min$ denote maximum or minimum computation time, and $y = max$ or $y = min$ denote maximum or minimum number or access requests, respectively, indicating a total of up to $4\cdot\mu^{max}_{m+d,j}$ new time windows for each subset $F_{m,d}$. Note that parameters $exec$ and μ in this case specify the computation time and access requests of each phase (rather than each superblock). The time windows are then computed as follows:

$$\Delta^{x,y}_{m,d,k} = \sum_{i=m+1}^{m+d-1} exec^{x}_{i,j} + \left(\sum_{i=m}^{m+d-1} \mu^{y}_{i,j} + k - 1\right) \cdot C \tag{4}$$

$$\gamma^{y}_{m,d,k} = \sum_{i=m}^{m+d-1} \mu^{y}_{i,j} + k \tag{5}$$

It is important to note that the time windows are computed only for those phase subsets $F_{m,d}$, whose destination $f_{m+d,j}$ is an accessing phase ($\mu^{max}_{m+d,j} > 0$), since no new access requests can be issued during an execution phase. Additionally, for the computation of the time windows, we neglect the computation time of the first and the last phase of each subset, so as to minimize the window length. This is allowed because if any of these phases is an accessing phase, then its computation time is 0 whereas if it is an execution phase, then access requests will occur only after it.

Computing the time windows for subsets $F_{m,d}$, whose phase sequence spans over the period, needs to consider the gap g between the last phase of S_j and the period W_j of the processing cycle. During real-time execution, when interferences actually occur over the shared resource, this gap between two consecutive executions of S_j can be arbitrarily small. Therefore, to derive a conservative arrival curve, we need to consider its minimal feasible length (under any possible runtime scenario), g_{min}, while computing the time windows. If no information about it is available, we consider $g_{min} = 0$, which implies that in the worst case a new execution of S_j starts immediately after the previous one is completed.

Based on the above, each phase subset $F_{m,d}$ in $\{SjSj\}$ is characterized by a set of tuples $t^{x,y}_{m,d,k}$(Eq. 6), i. e., a set of time windows and the respective number of access requests that can be issued by p_j within them. The tuples are defined as follows (for $k \in [1, \mu^{max}_{m+d,j}]$):

$$t^{x,y}_{m,d,k} = (\gamma^{y}_{m,d,k}, \Delta^{x,y}_{m,d,k} + g_{min}) \tag{6}$$

(iii) The computed tuples of each phase subset will be used to derive the core's access request arrival curve. In particular, by retrieving the maximum number of access requests for every time window of length $\Delta = \{0...W_j\}$ from the computed tuples[4], we can obtain the first part of the arrival curve, $\tilde{\alpha}_j$. Formally, if function $\delta(t)$ returns the length of the time window and $\nu(t)$ the number of access requests for each tuple $t^{x,y}_{m,d,k}$, then the first part of the arrival curve, for $\Delta \in [0, W_j]$, is computed as:

$$\tilde{\alpha}_j(\Delta) = \underset{\forall t^{x,y}_{m,d,k};\delta(t^{x,y}_{m,d,k})=\Delta}{\text{argmax}} \nu(t^{x,y}_{m,d,k}) \tag{7}$$

The infinite arrival curve α_j is eventually constructed as a concatenation of the initial part, $\tilde{\alpha}_j$, and a periodic part, i. e., a scaled version of $\tilde{\alpha}_j$ which is repeated infinitely. The obtained arrival curve α_j provides then a safe upper bound on the number of access requests issued by the superblock sequence S_j on p_j in any time window Δ (proof similar to [18]).

It is interesting to notice that adding the individual arrival curves α_j of the cores that are abstracted in our system model yields a safe upper bound on the worst-case interference α that these cores may cause on the arbiter in any time interval Δ:

$$\alpha(\Delta) = \sum_{p_j \in P \setminus \{p_{i(CUA)}\}} \alpha_j(\Delta) \tag{8}$$

The sum of the abstracted cores' arrival curves is mentioned in the following as interference curve α.

4.2.2 Embedding an interference curve into the TA-based analysis

For embedding the worst-case interference arrival curve α of the abstracted processing cores into the TA-based system model, we exploit the results of Lampka et al. in [12]. The goal is to transform α into a meaningful set of TA that will model the emission of interfering access requests at such a rate so that α is never violated.

Initially, to reduce the complexity of the embedding, we over-approximate curve α by a single staircase function $\alpha^{st}(\Delta) \geq \alpha(\Delta), \forall\Delta \in [0, \infty)$ (Eq. 2), as shown in Figure 4. 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 event streams that are specified by this new arrival curve can be generated by two TA, as depicted in Figure 5. These TA are adapted versions of the ones presented in [12], so as to comply with our system specification. The *Upper Bound* TA (UTA) is responsible for guarding the upper staircase function α^{st} (with fixed parameters `Bmax`, `Delta` and `s`), whereas the *Access Request Generator* (ARG) emits access requests 'on behalf of' the abstracted cores on condition that UTA permits it.

UTA models partly what is known from the computer networks field as a 'leaky bucket'. When the leaky bucket contains at least one token (unreleased access request), a corresponding event (request emission) can take place. If the

[4]Note that the linear approximations required in the method of [18] are obsolete, since the computed tuples capture already all possible access request arrivals.

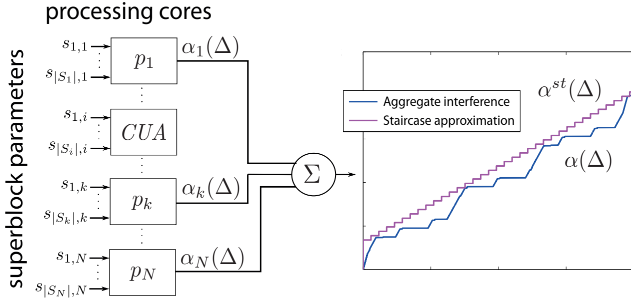
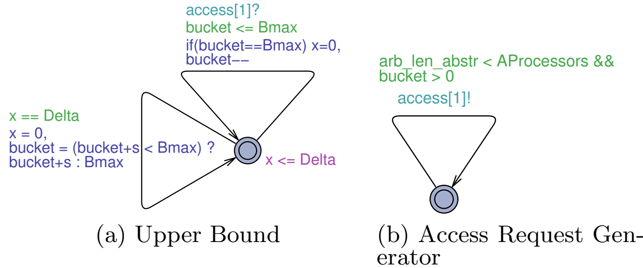


Figure 4: RTC Interference representation



(a) Upper Bound (b) Access Request Generator

Figure 5: Interference generating TA

leaky bucket is, however, empty, no requests can be emitted before new tokens are generated. The leaky bucket is configured so as to implement the upper staircase function α^{st}. Namely, it has a maximal capacity of B, being full in its initial state. An amount of s new tokens is produced every δ time units and one token is consumed every time a request is emitted by ARG.

Request emission by ARG is enabled as long as (a) at least one token is contained in the leaky bucket and (b) the current amount of pending interfering requests is lower than the number of abstracted cores (to consider only realistic scenarios). If both conditions are valid, then ARG may issue a new request, without restriction on the time point when the latter occurs (to account for all traces below α^{st}). Note that in the TA of Figure 5, '1' refers to the default 'identifier' of the access request generator as seen by the arbiter (representing without distinction any of the abstracted cores' identifiers).

Substitution of the $(N-1)$ *Scheduler* and the $(M-M_i)$ *Superblock* TA instances of the abstracted cores with the presented 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. 5, though, the pessimism in the WCRT estimates for the superblocks of CUA is limited.

4.2.3 Further Abstractions

On top of the two basic abstraction steps, i. e., the superblock model of execution and the interference representation with arrival curves, more abstractions and optimizations of our system specification can be considered. Some of them are briefly discussed in the following:

1. In the case of exact system specification (without interference abstraction) with a FCFS/RR arbiter, the state space exploration for the verification of a superblock's WCRT can be restricted to the duration of one hyper-period of the cores' processing cycles (least common multiple of their periods). Within a hyper-period, all possible interference patterns are exhibited, hence a safe WCRT estimate for each superblock can be computed.
2. In the case of system specification with the interference abstraction and a FCFS/RR arbiter, the *Superblock* TA can be simplified to model not periodic execution, but a single execution. Since all possible interference streams (bounded by α^{st}) can be explored for the time interval of one superblock execution and due to the property of sub-additivity of the arrival curves, the WCRT observed during it is a safe bound of the overall WCRT.
3. The previous optimization can be also applied for systems with a TDMA/FlexRay arbiter when we consider (enumerate and model) all possible offsets for the starting time of the superblock within the respective arbitration cycle.
4. For system models with a TDMA arbiter, the interference from the competing cores can be ignored (not modeled). This is because, it does not affect the WCRT of the superblocks executing on CUA. The same holds also for the ST segment of the FlexRay arbitration cycle.

Table 1: EEMBC benchmarks as superblocks

Benchmark	Acq (μ^a)	Exec ($exec$, ns)	Rep (μ^r)	Period (μs)
canrdr01	187	2734.2	9	44
cacheb01	91	1544.9	10	24
tblook01	267	5061.3	7	62
a2time01	129	1514.7	9	30
rspeed01	94	1331.6	7	24
bitmnp01	171	100064.0	48	160

Further abstractions and optimizations can be found in the technical report [9].

5. EMPIRICAL EVALUATION

This section presents two case studies, where the proposed state-based WCRT analysis approach was applied in order to evaluate its scalability and accuracy (pessimism) of the obtained results. In the following, we refer to the WCRT analysis with TA exploiting the superblock structure as *TAS*, and to the WCRT analysis with TA exploiting the superblock structure and the arrival curve abstraction as *TASA*. All presented verifications were performed with Uppaal v.4.1.7 on a system with a Dual-Core AMD Opteron CPU @2.7GHz and 8 GB RAM.

Comparing Accuracy of TAS to Analytic Methods. In the first scenario, we compare the accuracy of the WCRTs computed with TAS to those computed with state-of-the-art analytic approaches which also exploit the superblock structure. As reference, we have selected the approach from [18] which can be considered representative and similar to other analytic approaches [21, 23].

For this purpose, we use six applications from the industry-standard EEMBC 1.1 embedded benchmark suite [2]. These applications, which are typical for the automotive domain, were programmed and compiled using the PREM framework [16] so as to comply with the superblock model of execution. In particular, four sequential executions of each application are specified by one dedicated superblock. The derived memory access requests due to cache misses for the acquisition (Acq) and replication (Rep) phases, and the worst-case computation times for the execution phase (Exec) of the respective superblocks on the target architecture are shown in Table 1.

Each application is assumed to be executed periodically and mapped to a dedicated core. All cores have access to

Table 2: WCRT results of EEMBC benchmarks

Cores	Benchmark Set	WCRT(TAS) FCFS/RR	Avg.Verif.Time FCFS/RR	WCRT([18]) FCFS/RR	Pessimism of [18](%)
1	canrdr01	9711.8	0.06 sec	9711.8	0
2	canrdr01	13307.4	0.64 / 0.65 sec	13307.4	0
	cacheb01	8722		8736.1	0.2
3	canrdr01	20078.4	3.9 / 4.0 sec	20285	1
	cacheb01	12317.6/12282		12331.7	0.1/0.4
	tblook01	25459.6		25638.1	0.7
4	canrdr01	25489.6	31.8 / 30.4 sec	28900.2	13.4
	cacheb01	15913.2/15877.6		15927.3	0.1/0.3
	tblook01	31164.1		38988.1	25.1
	a2time01	19829.2		19848.7	0.1
5	canrdr01	30936.4	102.4 / 58.7 sec	37622.2	21.6
	cacheb01	19402/19366.4		19522.9	0.6/0.8
	tblook01	42454.7		53833.3	**26.8**
	a2time01	23424.8		26078.7	11.3
	rspeed01	19188.4/19152.8		19309.6	0.6/0.8
6	canrdr01	41758.8/41794.4	14.2 / 5.2 min	43709.8	4.7/4.6
	cacheb01	22997.6		23118.5	0.5
	tblook01	53150.8/53222.0		61629.7	15.9/15.8
	a2time01	29690.4/29654.8		30991.5	4.8/4.5
	rspeed01	22784/22712.8		22905.2	0.5/0.8
	bitmnp01	141253.2/141288.8		146842.4	4.0/3.9

a shared memory in case of cache misses. The processing cycles of cores are considered synchronized (same starting time of first cycle). The application periods are specified such that execution will be completed within them. That is, each superblock's period is at least equal to its conservative WCRT estimate, $WCRT_{cons} = (\mu^a + \mu^r) \cdot N \cdot C + exec$, which assumes that every access request experiences the worst possible serving delay, i. e., $N \cdot C$ in the case of FCFS or RR resource arbitration (e. g., for $N = 6$ and $C = 35.6$ns, we have $WCRT_{cons} = 146.8\mu$s for benchmark bitmnp01).

The comparison considers first a single core on which benchmark canrdr01 is executed. Gradually, the system is expanded by including one more core with a different application mapped to it on every step (up to 6 cores). The analysis is being performed once for FCFS arbitration and once for RR arbitration on the shared memory controller. Table 2 presents the derived WCRT estimates and the required verification times to obtain them (using the `sup:Superblock.x` query in Uppaal).

The results show that for the considered benchmark suite, tight WCRT estimates (up to 26.8% better than those of a state-of-the-art analytic method) can be verified in a few minutes for systems with up to 6 cores when TAS is employed. This indicates that the first proposed abstraction (superblock structure with dedicated phases) already provides a considerable benefit regarding scalability (w. r. t. the number of cores) compared to earlier model checking-based analysis methods, in which analysis did not scale efficiently beyond 2 cores for event-driven resource arbitration scenarios.

Evaluating Scalability and Accuracy of TASA. In the second case study, we explore the scalability of the proposed approach when resource accesses of some of the processing cores are abstracted with an arrival curve, i.e., the TASA approach. We also evaluate the accuracy of TASA w. r. t. state-of-the-art methods.

For this scenario, we use a real-world automotive application provided by an automotive supplier, which consists of 4 independent tasks. Each task is defined as a sequence of 2-8 general superblocks, whose access requests and computation parameters (bounds) were derived with static analysis techniques[5].

We now consider systems with 2, 4, 8, 16, 24, 32 or 64

[5]The superblock and system parameters cannot be disclosed due to confidentiality restrictions.

Table 3: Verification time for a superblock's WCRT

Cores	FCFS		RR		TDMA	FlexRay	
	TAS	TASA	TAS	TASA	TAS/TASA	TAS	TASA
2	1.2 sec	1 sec	1.2 sec	0.5 sec	0.1 sec	0.7 sec	27.1 sec
4	-	1.7 sec	-	1.1 sec	0.2 sec	0.9 sec	28.5 sec
8	-	31.3 sec	-	1.2 sec	0.5 sec	2.8 sec	17.8 sec
16	-	4.9 min	-	1.3 sec	0.9 sec	-	10.4 sec
24	-	18.8 min	-	1.6 sec	1.1 sec	-	26.8 sec
32	-	-	-	1.9 sec	1.2 sec	-	57.4 sec
64	-	-	-	2.7 sec	1.7 sec	-	7 min

cores and a shared memory that can be arbitrated according to any of the policies addressed in this paper. Each of the application tasks is executed periodically on one core (for systems with more than 4 cores, tasks are replicated), and depending on the arbitration policy of the memory controller, we make the following additional assumptions:

- When the shared memory arbitration is FCFS or RR, the processing cycles of different cores are considered non-synchronized, i. e., execution of superblock $s_{1,j}$ can start at any time within $[0, W_j)$. On the other hand, for TDMA and FlexRay, the processing cycles are considered synchronized to each other and also to the arbitration cycle, i. e., all tasks and the arbitration cycle start at time 0.
- The DYN segment of FlexRay makes 20% of the total arbitration cycle and enables cycle multiplexing. Namely there are two minislot assignments which alternate with each other in consecutive arbitration cycles.

The system model, in this case, exploits the interference abstraction (TASA). Therefore, we analyze the WCRT of a selected task running on a particular core (CUA), while all other cores are modeled by an access request generator that emits request streams bounded by their aggregate interference curve (Sec. 4.2.2). Additionally, abstractions 2 to 4 of Sec. 4.2.3 are applied whenever possible.

Table 3 shows the verification time required for each WCRT estimate for the different system configurations and arbitration policies. For comparison reasons, the respective verification times for TAS are also included, when available. For the FCFS and RR cases, the analysis scalability for TAS gets severely challenged due to the complexity of the considered industrial application and the assumption of non-synchronized processing cycles among the cores. Particularly, non-determinism w. r. t. the starting time of each task's first execution causes the analysis not to scale beyond 2 cores. The introduction of the interference abstraction (TASA), however, enables us to overcome this obstacle and obtain safe upper bounds on the WCRT of the considered tasks in a few minutes for systems with up to 24 cores for FCFS and 64 cores for RR and FlexRay. This is a major step forward compared to any of the existing approaches, opening the way for efficient interference analysis even in manycore systems.

In the case of TDMA arbitration, analysis scales very efficiently for any number of cores due to abstraction 4 of Sec. 4.2.3, i. e., the non-representation of interference in the system model. Because of this, the verification time in Table 3 is the same for both systems with (TASA) and without (TAS) the interference abstraction.

Furthermore, Table 4 shows the accuracy of TASA. For FCFS and RR arbitration schemes in systems with up to 2 cores, accuracy is compared against the best current results which are given by TAS. For systems with more than 2 cores, however, accuracy is compared against the methods presented in [18]. A value of '0', in this case, means that the results of both TASA and the analytic methods exhibit the same degree of pessimism. For TDMA systems, accuracy is

Table 4: Accuracy of TASA compared to state-of-the-art methods

Cores	FCFS(%)	RR(%)	TDMA(%)	FlexRay(%)
2	0	0	0	0
4	11.1	11.1	0	0
8	13.8	13.8	0	**1.8**
16	17.2	17.2	0	-
24	**17.3**	17.3	0	-
32	-	16.8	0	-
64	-	**25.4**	0	-

evaluated against results obtained with the method in [21]. Since for FlexRay arbitration no analytic methodology is known, the accuracy of the obtained WCRTs is compared to the results of TAS when the latter are available.

For FCFS and RR, comparison shows that the results of TASA can be more pessimistic (up to 25.4%) in particular cases. As main source of this pessimism, we identified the behavior of the ARG TA in the abstract system specification (Sec. 4.2.2), which emits interfering requests non-deterministically over time, thus enabling the exploration of several request streams that are bounded by α^{st}, but may never be encountered in the real-time system.

For systems with a FlexRay arbiter, the WCRT estimates are identical for the TAS and TASA approaches for systems with 2 and 4 cores, and only slightly more pessimistic (1.8%) for TASA for systems with 8 cores. Note that beyond 8 cores, pessimism could not be evaluated as no other approach can compute tight WCRT bounds for tasks with synchronous access requests to a FlexRay-arbitrated resource.

Results in this second case study point out that the gain in scalability of the analysis is not severely compromised by the accuracy of the obtained results.

6. CONCLUSION

The paper presents a framework for state-based WCRT analysis of tasks that are executed in parallel on a multi-core platform and have synchronous accesses to a shared resource under FCFS, RR, TDMA, or FlexRay arbitration schemes. For achieving scalability, tasks are organized in dedicated superblocks, namely sequences of resource access and computation phases. Empirical evaluations with benchmark applications show that the proposed approach delivers safe results, which due to its exact TA-based system specification (TAS) are tighter than those of state-of-the-art analytic methods. On a next step, we abstractly represent access requests of tasks, which compete for the shared resource against a task under analysis, by an RTC arrival curve and integrate it into the TA-based system specification (TASA). A case study based on a real-world industrial application shows that this method scales much better than previous state-based approaches without sacrificing on the accuracy of the results. What is more, the presented methods are the first to analyze WCRT of tasks with synchronous accesses to a shared resource arbitrated by FlexRay.

Acknowledgments

This work is supported by EU FP7 under grant agreement no. 288175. We would like to thank R. Pellizzoni for providing us with the EEMBC benchmark parameters for the evaluation of our method.

7. REFERENCES

[1] AutoSAR. Release 4.0. http://www.autosar.org.

[2] EEMBC 1.1 Embedded Benchmark Suite. http://www.eembc.org/benchmark/automotive_sl.php.

[3] European commission's 7th framework programme: Design for predictability and efficiency. www.predator-project.eu.

[4] Flexray communications system protocol specification, version 2.1, revision a. http://www.flexray.com/.

[5] R. Alur and D. L. Dill. Automata For Modeling Real-Time Systems. In *Automata, Languages and Programming*, pages 322–335. Springer, 1990.

[6] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems*, pages 200–236, 2004.

[7] D. B. Chokshi and P. Bhaduri. Performance analysis of flexray-based systems using real-time calculus, revisited. In *ACM Symposium on Applied Computing*, pages 351–356, 2010.

[8] A. Ferrari, M. Di Natale, G. Gentile, G. Reggiani, and P. Gai. Time and memory tradeoffs in the implementation of AUTOSAR components. In *Design, Automation, Test in Europe Conference*, pages 864 –869, 2009.

[9] G. Giannopoulou, N. Stoimenov, A. Schranzhofer, and L. Thiele. Derivation of access request arrival curves for dedicated superblock sequences. TIK-rep. 347, Computer Engineering & Networks Laboratory, ETH Zurich, 2012.

[10] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards WCET Analysis of Multicore Architectures Using UPPAAL. In *Workshop on Worst-Case Execution Time Analysis*, pages 101–112, 2010.

[11] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis - The SymTA/S Approach. *Computers and Digital Techniques*, 152(2):148–166, 2005.

[12] 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.

[13] K. G. Larsen, C. Weise, W. Yi, and J. Pearson. Clock difference diagrams. *Nordic Journal of Computing*, 6(3):271–198, 1999.

[14] M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Real-Time Systems Symposium*, pages 339–349, 2010.

[15] M. Negrean, S. Schliecker, and R. Ernst. Response-time analysis of arbitrarily activated tasks in multiprocessor systems with shared resources. In *Design, Automation, Test in Europe Conference*, pages 524–529, 2009.

[16] 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 *Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, 2011.

[17] 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.

[18] 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*, pages 741–746, 2010.

[19] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the FlexRay communication protocol. *Real-Time Systems*, 39(1):205–235, 2008.

[20] 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*, pages 759–764, 2010.

[21] A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *Real-Time and Embedded Technology and Applications Symposium*, pages 215–224, 2010.

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

[23] 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*, pages 213–222, 2011.

[24] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Symposium on Circuits and Systems*, volume 4, pages 101–104, 2000.

[25] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966 –978, 2009.

[26] S. Yovine. Model checking timed automata. In *Lectures on Embedded Systems*, pages 114–152. Springer, 1998.