Schedulability and Memory Interference Analysis of Multicore Preemptive Real-time Systems*

Jalil Boudjadar, Simin Nadjm-Tehrani
Department of Computer and Information Science
Linköping University, Sweden

ABSTRACT

Today's embedded systems demand increasing computing power to accommodate the ever-growing software functionality. Automotive and avionic systems aim to leverage the high performance capabilities of multicore platforms, but are faced with challenges with respect to temporal predictability. Multicore designers have achieved much progress on improvement of memory-dependent performance in caching systems and shared memories in general. However, having applications running simultaneously and requesting the access to the shared memories concurrently leads to interference. The performance unpredictability resulting from interference at any shared memory level may lead to violation of the timing properties in safety-critical real-time systems. In this paper, we introduce a formal analysis framework for the schedulability and memory interference of multicore systems with shared caches and DRAM. We build a multicore system model with a fine grained application behavior given in terms of periodic preemptible tasks, described with explicit read and write access numbers for shared caches and DRAM. We also provide a method to analyze and recommend candidates for task-to-core reallocation with the goal to find schedulable configurations if a given system is not schedulable. Our model-based framework is realized using Uppaal and has been used to analyze a case study.

Keywords

Schedulability, memory interference, processor utilization, multicore systems, task migration, model checking.

1. INTRODUCTION

Motivated by the need of high processing capacity and the decreasing cost of electronics, multicore platforms are finding their way into safety-critical embedded systems. Following the same trend as automotive industry, avionic system developers are considering the use of multicore platforms to leverage the potential for higher performance, and reduce the weight of on-board computing equipment. This is achieved by integrating different subsystems, potentially provided by different vendors, to enable incremental Design and Certification (iD&C) [29], recommended by the standard Integrated Modular Avionics (IMA) architecture [24]. In contrast to the classical federated architecture, IMA supports functions related to different subsystems to share the same computing platform with an efficient use of the hardware. Such a support is implemented using partitioning.

Partitioning [10] amounts to isolating, in space and time, the system processes to make the complexity and maintenance manageable. Moreover, it prevents propagating failure conditions from one system component to another, particularly from lower criticality to highly critical components, thereby enforcing fault containment. Multicore platforms can provide enormously larger processing capacity compared to the classical single core platforms so that more software-intensive systems can be deployed. However, the underlying complexity and interference to access shared resources may lead to lack of guarantees on schedulability and predictability.

Focusing on the temporal partitioning, in single core platforms the estimation of worst case execution time (WCET) is based on the time spent by the longest execution path of a process while assuming perfect memory access i.e. the shared memory is immediately available whenever an access request occurs. For systems with fixed priorities assigned at a global level, a separate term (blocking delay) is added for potential contention on shared resources. In multicore settings, where different applications running concurrently on different cores compete for the access to shared memories, the combination of local cache misses and interference delay for accessing a shared memory can be large and highly variable depending on the platform architecture and the number of parallel access requests. Interference represents a big challenge for the predictability of real-time embedded systems, therefore it must be considered at the design and integration stages.

In recent years, some progress on WCET and memory interference analysis of multicore systems has been achieved [15, 20, 8, 2, 7, 31]. In this paper, we provide a model-based framework for formal analysis of schedulability and memory interference of real-time applications running on multicore platforms. Our framework enables modular description of applications and platforms where models are described sep-

*This work has been supported by the NFFP6 project 2014-00917, and financed by Swedish Innovation Agency (Vinnova).

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

arately, thus different mappings of an application to different platforms can be studied as a state space exploration.

The platform consists of a set of cores each with a local scheduler and a local cache. The cores share the cache level 2 (L2) and Dynamic Random Access Memory (DRAM). We use the cache coloring policy [12] to arbitrate the concurrent access requests to L2. In addition, we adopt the policy First Ready-First Come First Serve (FR-FCFS) [22, 11] commonly used by modern COTS-based memory controllers to schedule the DRAM access requests. A schematic version of the policy is modelled in section 5.2.2 (see Algorithm 1).

The application model is given by a set of periodic preemptive tasks described with a fine grained behavior. Besides classical scheduling parameters, the description of each task includes the worst case resource access numbers (WCRA) [20, 6] stating how many times the task accesses to each shared memory for both read and write patterns. We distinguish between read and write actions to shared memories as read actions are blocking for cores, while write actions are not blocking and can be performed using dedicated buffers. To make the system behavior more realistic, we spread out the access requests to L2 and DRAM non-deterministically during tasks execution rather than using dedicated phases [25, 31]. This is motivated by the fact that a task execution, and thereby the issue time of data requests, may vary from one period to another following the changes in the computation environment.

Our model-based framework has been realized using Uppaal [3]. We use the symbolic model checker of Uppaal to analyze the system schedulability whereas interference-sensitive WCET (isWCET) and core utilization are analyzed using statistical model checking. By statistical we mean simulating different executions but not exploring the whole state space. We also propose an approach to support the distribution of tasks among cores so that the whole system becomes schedulable. This is done in two steps:

1. we analyze the system schedulability according to a given allocation of tasks to cores. If the system is not schedulable, we proceed to step (2).

2. we perform an analysis of core utilizations and average interference delay per access request to shared memories, and recommend a redistribution of the workload from overloaded cores to relatively less loaded ones by migrating tasks, then redo process (1) with the new recommended mapping.

The analysis process ends if a configuration is schedulable or system is deemed not schedulable despite some load distribution within the given constraints.

To sum up, the contributions of this paper are as follows:

- Model-based framework for modular description of multicore platforms and applications.

- Rigorous analysis of schedulability using symbolic model-checking.

- Statistical analysis of performance: core utilizations and memory interference.

- Method for reallocation of tasks to cores upon non-schedulability of a given configuration.

The rest of the paper is organized as follows: Section 2 reviews the related work. Section 3 describes the necessary background. Section 4 provides an overview of our work. Section 5 presents the Uppaal models of framework, whereas Section 6 describes schedulability, isWCET analysis and method for potential reallocation of tasks. In Section 7, a small case study is presented to show preliminary evidence of the feasibility and efficacy of the framework. Finally, Section 8 concludes the paper.

2. RELATED WORK

The analysis of schedulability and memory interference of multicore real-time systems is an active research area and can be categorized into 2 directions: computing DRAM sensitive WCETs [33, 20, 11, 2] and bounding the DRAM interference delays [17, 8, 27]. The common element of the different analysis settings is: a) measuring the WCET of each task in isolation on a single core; b) calculating the interference time caused by the access to the shared resource, e.g DRAM. The first element can be performed using dedicated measurement tools like SWEET [15] and Heptane [21], whereas the interference delays are obtained either by static analysis or using different theories (as described below).

Schranzhofer *et al.* [25] introduce an analytical approach to analyze the worst case response time of a task set running on a multicore platform. Each task runs over different dedicated phases like acquisition, execution and replication. The execution of a task takes place in a sequence of non-preemptible superblocks. The shared resource (DRAM) access requests occur only during the acquisition and replication phases of each superblock. The use of static predefined schedules among the superblocks of different tasks results in a static runtime where the analysis results are predictable.

In a similar way, Nowotsch *et al.* [20] present a theory to compute and bound interference-sensitive WCET (isWCET) of processes running on a multicore platform with a shared DRAM memory. Each process is assigned a certain share of DRAM capacity. The experiment results in [20] show that the estimated WCET values are an overestimation of the observed ones. Such an overestimation enormously affects the isWCET. In both of the above works [25, 20], the static allocation of DRAM using Time Division Multiple Access (TDMA) policy lowers the complexity but it leads to a poor utilization of DRAM.

Other authors [33, 11] introduce analytical frameworks to calculate memory interference delays in multicore systems. The interference of the request under analysis is calculated based on inter and intra-bank interference as well as *row-opening* (loading data from a row to a row-buffer) and *precharge* (moving data back from a row-buffer to a row). Each of these elements is calculated separately. According to the analysis results, the higher the number of sharing cores the longer the interference will be. Since each request is analyzed separately, a relevant question is how these analytical frameworks deal with memory-intensive systems where each process performs thousands of memory requests.

Madsen *et al.* [18] introduce a model-based framework to study the impact of execution platforms on schedulability. The platform model is given by a set of processing elements, each of which consists of a local memory (cache), a scheduler and a processor. However, not considering shared memories may lead to an underestimation of the workload because, in practice, the delays resulting from the access to shared

cache and DRAM enormously impact the response time of tasks.

Lv *et al.* [17] combine the abstract interpretation of software systems running on multicore platforms with model checking to calculate the isWCET. The abstract interpretation aims to obtain the local cache behavior of a program running on a given core in order to capture the precise timing information when the program accesses DRAM (cache miss). However, characterizing each program instruction with an execution time and an access pattern (hit/miss) is not considered as feasible for a design phase of systems. The authors approach would require a rather expensive analysis of the binary code. Instead, we use a granularity at task level.

Gustavsson *et al.* [8] investigate a method based on model checking to calculate the WCETs of programs running on multicore platforms. The authors consider both local and shared caches as well as DRAM. The miss/hit of local and shared caches is non-deterministic. The WCET estimate is obtained using a binary search, which could be expensive if the initial WCET estimates are far away from the final values. Our work aligns with this but we use statistical model checking, where the analysis process runs once and performs the evaluation of thousands of different executions, rather than repeating the binary analysis process manually.

Subramanian *et al.* [27] develop a scheduling scheme (MISE-Fair) to minimize the maximum slowdown of applications running on multicore platforms. Application slowdown is the delay experienced by the application due to the wait to access shared resources compared to the case when the application runs alone on the platform. In essence, MISE-Fair estimates the slowdown of each application and redistributes the memory bandwidth to reduce the slowdown of the most slowed-down applications. However, the minimization of DRAM-related slowdown of tasks may impact the scheduling at cores level.

In some of these papers, the authors calculate the isWCET of tasks running on a given multicore platform, then apply appropriate analysis to check the schedulability. This means that the estimated isWCET cannot be used to check the schedulability if the application tasks run on a different platform. So far, for each new platform the isWCETs need to be calculated using the platform description and one needs to rerun the appropriate schedulability analysis process. In our work, we combine the analysis processes so that for a given platform we check whether the application tasks will be schedulable or not.

Compared to the state of the art, our framework enables modular description of applications and platforms with a flexible mapping. The application is given by a fine grained behavior including explicit numbers of read and write requests to each shared memory, whereas the platform includes a set of processing cores and a hierarchy of shared memories. Each of these elements is separately modeled by a template, i.e. a parameterized state-transition behavior. To make the application model more realistic, we spread out the access requests to L2 and DRAM non-deterministically during task execution rather than using dedicated phases. System configurations (consisting in an application instance, a platform instance and a mapping of the application to the platform) can be created by just providing template parameters, so that design space exploration can be performed. We use model checking to analyze schedulability and performance, in terms of memory interference and core utilization. Moreover, a novelty of our work is the introduction a method for reallocate tasks to cores upon non-schedulability based on performance metrics (core utilization, average interference delay per access request).

3. BACKGROUND

Before diving into the description of our contribution, we introduce the scheduling fundamentals of shared resources in multicore platforms: cores, shared caches and DRAM.

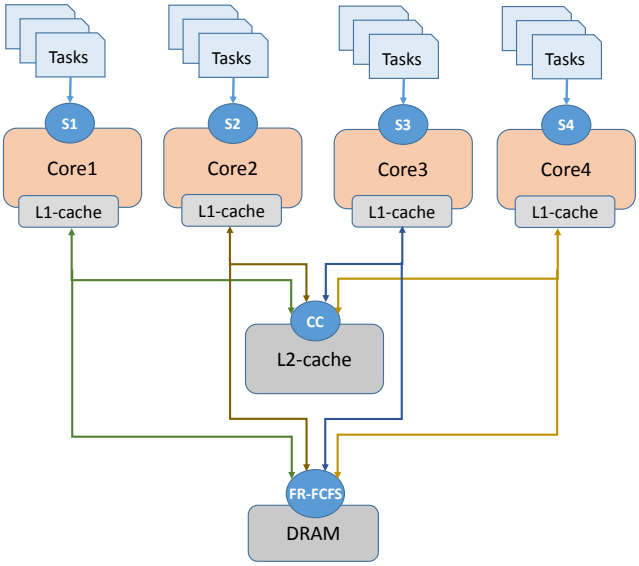

Figure 1: Simplified multicore system architecture.

The overall system architecture we consider in this paper is depicted in Figure. 1, where *cc* is the cache coloring policy and s_i are CPU scheduling policies.

3.1 Core Level Scheduling

To leverage the processing performance of a computing system, a processing unit (core in our context) can be assigned more than one task, only one task running at any point in time. The arbitration between the different task executions is performed according to a scheduling policy.

Basically, a scheduling policy determines, at any point in time, which task from the ready queue must execute first and whether a given task should be preempted by another. The most commonly used scheduling algorithms are Earliest Deadline First (EDF), Fixed Priority Scheduling (FPS) and Rate Monotonic (RM). The key factor in selecting a task from ready queue can be priority, remaining execution time, etc.

Another recent alternative to schedule memory intensive application tasks is the use of a memory-centric policy [31, 30], where tasks are sorted in the queue according to a decreasing order of their WCRA (numbers of memory accesses). In our setting, we adopt FIFO, FPS and EDF as scheduling policies for the individual cores, but the method could easily be adapted to a memory-centric one.

3.2 Shared Cache Scheduling

In order to enhance the processing performance of multicore platforms, some of modern multicore processors [1] consider a shared cache level (L2) in addition to private caches (L1). The primary goal of sharing a cache between different cores is to reduce the access requests to the main memory DRAM, and by that shorten the DRAM interference time since the interference time is strongly correlated to the number of access requests [20].

Cache coloring policy [12, 32] is an algorithm to control the access to the shared L2 cache. It has been introduced to aid performance optimization where physical memory pages are mapped to cache pages, in contrast with old caching systems where virtual memory is mapped to the cache. This means avoiding clearance of cache pages on each context switch. During execution, the algorithm frees the old pages as necessary in order to make space for currently scheduled applications (recoloring). The coloring algorithm sorts the concurrent access requests according to their release times.

Estimating the optimal cache size for each application, in order to minimize the cache miss ratio, is a non-trivial task as it requires expensive analysis of the system execution. Moreover, the recoloring operation leads to a significant overhead if the pages are not intelligently selected [12].

In our framework, we do not consider the detailed internal architecture and size of DRAM and shared cache, we focus rather on measuring the delays caused by the concurrent accesses. The reason behind this is that the impact of these characteristics on the interference is already captured when performing the static analysis and identifying the WCRAs. Accordingly, recoloring is beyond the scope of this paper.

3.3 DRAM Access Scheduling

Conventionally, a DRAM is shared by all the platform processing units. However, this increases the complexity of managing memory accesses. DRAM controllers have adopted scheduling mechanisms similarly to processor scheduling.

Naive conventional policies, like First Come First Served (FCFS) and Read-First, schedule access requests according to their arrival times with a special preference to read requests since they cause the processor to stall while write requests can normally be performed using write buffers. Another alternative to schedule accesses to DRAM, in presence of a three-dimensional structure (bank, row and column), is the Hit-First policy. The Hit-First algorithm schedules row buffer hits before misses to reduce the average memory access latency and to improve bandwidth utilization [9, 22]. This is due to the fact that requests hitting in the row buffer have shorter latency than a row buffer miss.

In Time Division Multiple Access (TDMA) policy [23], the DRAM controller allocates statically a time slot to each core to access the DRAM in a predefined manner. TDMA provides a simple and fair scheduling among all cores, however, it does not exploit the spatial locality available in memory access streams as it does not consider the demands coming from different tasks at any time point.

To maximize data throughput and minimize the DRAM latency, DRAM controllers in modern COTS-based systems use First Ready-First Come First Serve (FR-FCFS) as a DRAM policy [22, 19, 11]. FR-FCFS considers a detailed DRAM structure in terms of banks, rows and columns. The DRAM scheduler can be viewed as a 2-level filter: bank level and bus level. The access requests can target different banks separately, where they will be queued in the corresponding bank queue. Access requests will be sorted at each bank queue first according to their readiness. Then, the candidates selected from banks level will be further sorted at bus level where the earliest request gains access, i.e. the first request showing up at bus level among the requests being selected by bank schedulers. If no request hits the row-buffer, older requests are prioritized over younger ones.

[1]E.g. Intel Core i7, AMD FX, ARM Cortex and FreeScale QorIQ processors.

4. ANALYSIS OVERVIEW

This section describes the inputs required by our framework, and how they can be derived from a concrete software system designed to run on a multicore platform. It also gives a birds eye view of the analysis process.

Conventionally, task periods and deadlines (potentially also priority and criticality levels) are identifiable during the requirements analysis. Given such attributes and the architecture of the execution platform, we calculate the interference sensitive WCET (isWCET) for each task and analyze the system schedulability. isWCET is basically the sum of WCET (on a single core) and the interference delays to access shared cache and DRAM.

4.1 Flow Analysis

WCET estimation is crucial for schedulability analysis, thus providing an accurate bound on the execution time needs to be performed in a rigorous manner. Flow analysis [28, 26, 7] is a technique to estimate the WCET of a program, it consists of simulating, or concretely running, a program in isolation and measuring the time spent. Flow analysis is exercized via different analysis tools, e.g. Chronos [14], SWEET [28] and Bound-T [1]. In case of single core platforms, memory interference is not a serious challenge because the CPU triggers one access request at time.

In multicore settings, execution time estimation has become a challenge because the number of possible interleavings increases exponentially with the number of processes, number of cores and number of shared resources [7]. In practice, concurrent programs may have astronomical numbers of legal interleavings which makes the interleaving analysis not feasible. An alternative is to reuse the knowledge acquired from the single core analysis and just focus on the interference resulting from the access to shared memories.

Technically, static analysis tools use symbolic execution engines to identify potential execution paths without necessarily having to run the program. Such representations can be structured in terms of control flow graph (CFG). Similarly to compilers, a static analysis tool parses a source code of a program and converts it to an intermediate representation where each state could be a class of different program configurations. WCET is then the time spent when executing the longest path of the CFG.

4.2 Profiling

System profiling [16, 34, 11] is a measurement-based approach to estimate how many times a process accesses shared memories. The system being analyzed is run for a sufficient number of times, each of which for a long enough duration enabling the execution of most of the system functions (code). The analysis focuses on each process individually, so that for each run we track how many times a process accesses a given shared memory. However, due to technol-

ogy limitations, the measurements can be performed at core level only, so that the obtained access number of a given core corresponds to the set of processes running on top of it. In order to obtain the memory access number for each process, one needs to run each process individually on one core during the analysis, so that the access number obtained corresponds only to the process being run. The number of accesses for each core can be obtained using performance monitor counters [34, 16], present in certain multicore platforms.

Profiling and static analysis techniques can both be used to measure WCET and WCRA [11, 34].

4.3 Formal Analysis

The output of flow analysis and profiling together with periods and deadlines, for a given allocation of tasks to cores, will be used as input to our framework to instantiate a system model reflecting the behavior and timing characteristics of the given software system. Such a system model can then be analyzed (using model checking) with respect to schedulability and isWCET as detailed in Section 6.

Once the system is determined to be non-schedulable for the given task-to-core allocation, we analyze what would be the utilization of cores and the interference delays if some tasks are reallocated to different cores in order to balance the workload. We will refer to such reallocations as *migration*, which should not be confused with the run-time movement of tasks. It is simply part of a pre-runtime analysis activity.

We compare different potential migration scenarios and recommend the configurations leading the system to be schedulable. In the context of IMA where tasks are encapsulated within partitions that are pre-allocated to given cores, one might need to move a whole partition instead of moving an individual task in order to satisfy both the *functional-architectural* constraints and *time composability* property of IMA. Besides, fault containment requirements must be respected when re-allocating partitions. The functional architectural constraints impose that the functionality of each architectural unit -partition- is maintained if the partition tasks do not execute on the same core. The time composability property states that the timing behavior of individual tasks will not change by the composition. In this work, since partitions are or regions are not formally modelled, we just focus on the impact of moving individual tasks. The load shedding created by moving a partition can be captured by the accumulated impact of re-allocation of its individual tasks, and the framework can be extended to model fault containment constraints.

Once a configuration is determined to be suitable, i.e. having a high probability to be schedulable according to the statistical analysis (SMC), one needs to rerun the rigorous schedulability analysis (symbolic model checking). Our framework is built to be flexible so that migrating a task from a core to another is feasible by just updating the coreId parameter of such a task with the Id of the new core.

5. SYSTEM MODEL DESCRIPTION

Basically, an embedded system is comprised of an application mapped to a platform. The application is a parallel composition of tasks. The platform consists of a set of processing elements, a shared cache level, a shared DRAM and mechanisms to share and control the access to the platform resources like processor schedulers and DRAM connectors.

We use $\mathcal{T}$ for the set of tasks and $\mathcal{C}$ for the set of cores. The assumptions in our models are as follows:

- Tasks are periodic and preemptible during CPU execution only, i.e. a task cannot be preempted while it is performing an access to a shared memory.
- Tasks assigned to the same core are arbitrated using a local CPU scheduler.
- We consider a local cache (L1) for each core, only one shared cache (L2) and one DRAM for all cores.
- We abstract each local cache using a number stating the time taken for fetching data from it.

5.1 Application Model

An application $AP = \{T_1, .., T_n\}$ is a set of tasks each of which describes the execution model of an individual process. The process behavior is abstracted at task level using WCET and WCRA for both shared cache and DRAM[2]. WCET is the pure execution time, considered in isolation, i.e. excluding the time to fetch data. Regarding data fetching, we consider 2 attributes $WCRA_c$ and $WCRA_m$ where: $WCRA_c$ corresponds to the maximum number of successful accesses (hits) to the shared cache L2; $WCRA_m$ is the number of DRAM accesses (corresponds to L2 miss) performed by a given process. Moreover, in order to distinguish between read and write accesses to each shared memory, we denote each of the attributes with r for read and w for write, i.e. $WCRA_c^r$, $WCRA_c^w$, $WCRA_m^r$ and $WCRA_m^w$. This is because read requests make the core stalling (strong impact on isWCET) while write requests do not as they can be performed using dedicated buffers.

Accordingly, an access request to a shared memory is given by a $pattern \in \{L2, DRAM\}$ stating to which memory the access hits and an attribute RW indicates whether it is a read (r) or write (w) action. Moreover, as we need to keep track of when the requests are issued, so that FR-FCFS algorithm determines the priorities of requests targeting DRAM, we use $issueT$. In fact, $issueT$ is initially empty and will be initialized by a core when the access request is triggered. Accordingly, an access request is formally given by $req = \langle pattern, RW, issueT \rangle$. $WCRA_c^r$ and $WCRA_c^w$, respectively $WCRA_m^r$ and $WCRA_m^w$, of a given task are then the numbers of read and write accesses to L2, respectively DRAM.

Definition 1 (Task structure). *A task T is given by* $\langle \text{Prd}, \text{Offset}, \text{WCET}, \text{WCRA}_c^r, \text{WCRA}_c^w, \text{WCRA}_m^r, \text{WCRA}_m^w, \text{Dln}, \text{Pri} \rangle$ *where* Prd *is the task period,* Offset *is the periodic offset,* WCET *is the pure execution time,* Dln *is the relative deadline whereas* Pri *is the priority level associated to T.* $\text{WCRA}_c^r, \text{WCRA}_c^w, \text{WCRA}_m^r$ *and* WCRA_m^w *are described above.*

In order to make the application specification flexible, we do not (statically) specify the identifier of the core to which the task is assigned. The mapping will rather be given during system instantiation. The behavior of a task is a basically a state-transition system, where states represent potential configurations of the corresponding process and transitions correspond to the execution of actions and events.

[2]thus validity of these numbers affects the validity of the outcomes

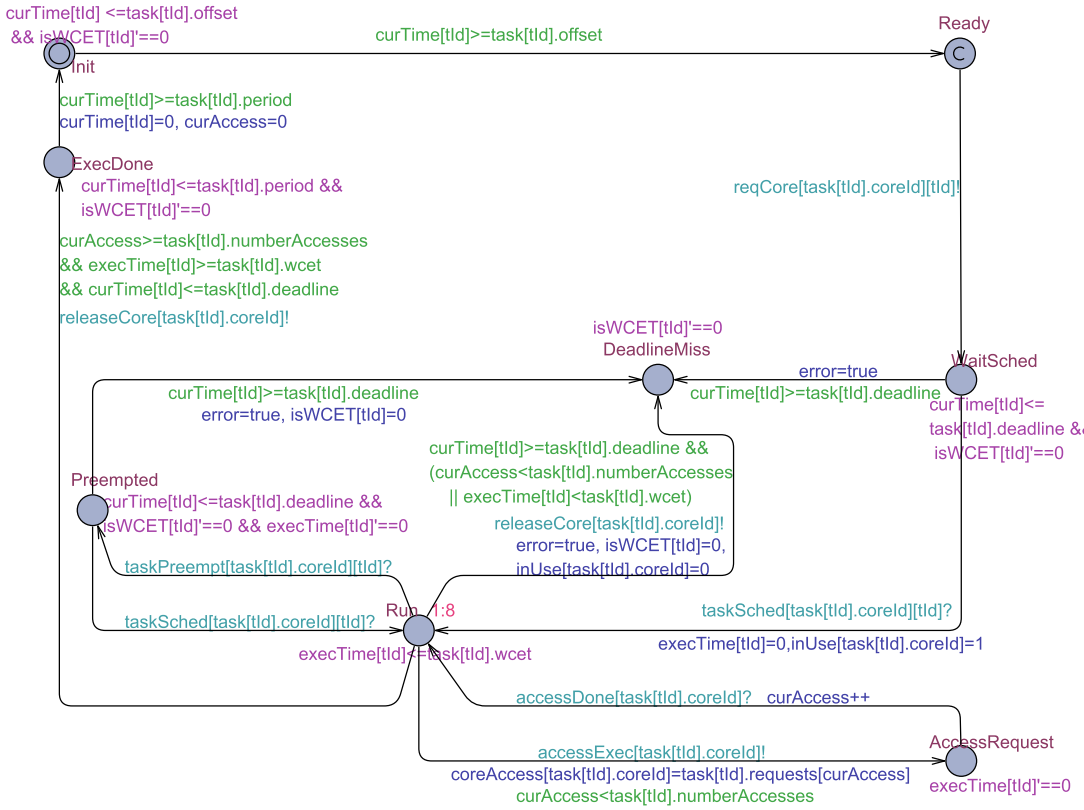

Figure 2: Task template model

Our Uppaal task template model is depicted in Figure. 2. To distinguish between different tasks, we associate to each task an identifier `tId` as a template parameter. The task starts at location Init where it initializes its variables if needed, during the offset time. Once the offset expires, the task moves to location Ready to request the core to which it is mapped, through a synchronization with the core scheduler on channel `reqCore`. The task waits to be scheduled at location WaitSched unless the deadline is reached by which it moves to location DeadlineMiss and updates a global variable *error* to true. Once a task is scheduled it updates the status of the corresponding core *inUse[]=1*, and moves to location Run where it executes. During its execution ($WCET$), a task non-deterministically triggers access requests to L2 and DRAM. For each access request, the task moves to location AccessRequest and waits until the access request is satisfied upon which it moves back to location Run. One can remark that, when a task is requesting and waiting for data the clock measuring $WCET$ is stopped (*execTime[tId]'==0*) so that only the effective execution at Run consumes $WCET$. This is implemented in Uppaal by assigning rate 0 to the derivative of *execTime[tId]*. Such a clock resumes at Run. From Run, the task joins either ExecDone, if the execution $WCET$ and accesses to L2 and DRAM ($numberAccesses = WCRA_c^r + WCRA_c^w + WCRA_m^r + WCRA_m^w$) are completed before deadline, or it moves to DeadlineMiss in case the execution or access requests are not achieved before deadline. Once the period expires, at location ExecDone, the task moves to Init to start a new period. The isWCET of a task is measured by clock *isWCET[tId]*. The measurement starts once a task is scheduled (location Run) and stops when the task execution is done (location ExecDone), so that it includes both $WCET$ and interference. The task template can be instantiated for different tasks by just providing the aforementioned parameters.

A task can be preempted during effective core utilization only (at location Run) by the scheduler of the core to which it is currently allocated, as during memory access the task is at location AccessRequest. Once a task is preempted it moves to location Preempted. If a ready task needs to preempt a running task, the preempting task has to wait until the current memory access of the running task is finished. One can see that we do not change the core status ($inUse$) when preempting a task and moving to location Preempted as the core keeps running but with the preempting task. When a task is preempted, neither its *isWCET* nor *execTime* clocks can progress. The task can exit the preemption location Preempted by either receiving a scheduling event *taskSched[]?* or reaching its deadline.

Our preemption pattern does not allow the preemption of tasks during access requests, thus it prevents a task to wait more than once for the same access request. This aligns to an operating system policy that has a preference for a lower interference time as opposed to a greedy utilization of cores.

5.2 Platform Model

A platform is composed of a set of processing elements PE, a shared cache (L2), DRAM memory and schedulers to manage the access to L2 and DRAM. Each processing element PE consists of a computation resource (core), a local cache (L1) and a scheduler to dispatch tasks to run on that core. The access time for local caches may vary from one PE

to another. Moreover, we consider the duration for an effective access (from grant to completion) to a shared memory as a platform parameter.

5.2.1 Modeling of Processing Elements

A processing element PE is given by $\langle C, sched, H \rangle$ where C is a core, *sched* is the scheduling policy (core scheduler) adopted and H is the local cache that we abstract using its access time *LocalCacheTime*. The core model is depicted in Figure. 3.

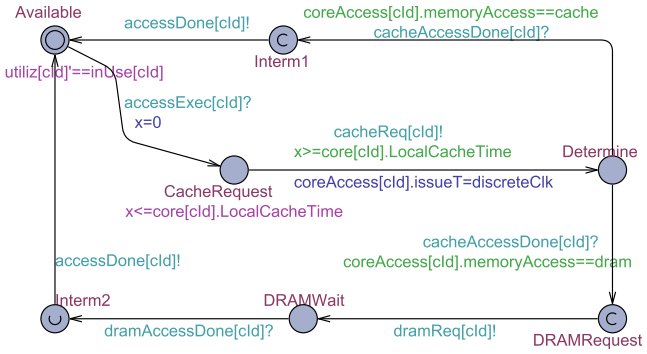

Figure 3: Core template model

Similarly to tasks, we assign to each core an identifier `cId` as a parameter to distinguish between the different platform cores. The core model is initially at location Available waiting for ready tasks. Through an allocation, the core model does not move from Available but the clock measuring its utilization *utiliz[cId]* starts counting (*utiliz[cId]'==inUse[cId]*). Such a clock can stop and resume according to the core status *inUse[cId]* manipulated at task level. Upon an access request to a shared memory (*accessExec[cId]?*), the core moves to location CacheRequest where it waits for the expiry of the local cache access time *LocalCacheTime* before performing the access request to the shared cache L2 and joins location Determine. The core updates the request issue time with the current time instant *issueT=discreteClk*.Once the access to L2 hits (*memoryAccess==cache*) and terminates, the core moves back immediately to location Available to continue executing the assigned task. Otherwise, once the L2 access terminates and misses (*memoryAccess==dram*) the core requests access to DRAM and joins immediately the location DRAMWait. The core blocking time on an access request (at locations Determine and DRAMWait) depends on the access nature. If it is a write action, the core will immediately be unlocked by the scheduler of the targeted memory, otherwise the core stalls until the read access finishes. Further details regarding how to handle read and write accesses will be provided in the description of L2 and DRAM schedulers.

The core needs to notify the running task when the current access request is done, i.e. once the core itself is notified by DRAM or L2 (according to the access pattern), so that it moves back as well from location AccessRequest to Run and accounts an access done (*curAccess++*). As it is not possible to associate two synchronization events with a transition in Uppaal, we introduce two intermediate locations Interm1 and Interm2. Thus, we create a sequence of 2 synchronizations without any delay in between. We use *urgentness* and *committedness* of Uppaal to enforce time to not elapse at a given location (locations marked with `U` and `C`).

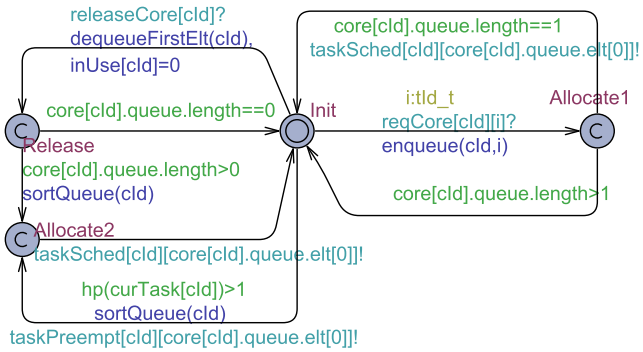

Figure 4: Scheduler template model

Figure. 4 depicts the core scheduler model. Initially at location Init waiting for a ready task, the core moves to location Allocate1 while queuing the identifier of the requesting task. If the core queue contains only one element (*queue.length==1*), which is the identifier of the newly added task, that task will immediately be scheduled otherwise the scheduler just moves back to Init, where it analyzes whether the newly added task has priority over the current running task. If so (*hp(curTask[cId])>1*), the scheduler preempts the current running task (when it is not performing an access to a shared memory) and sorts the ready queue while moving to location Allocate2. From that location, the scheduler schedules immediately the task having the highest priority in the queue. Channel *taskPreempt* is urgent, thus whenever the synchronization is available the transition from Init to Allocate2 is immediately triggered.

Once the core is released by the current running task through a signal on channel *releaseCore* due to execution termination, the scheduler moves to Release while removing the first element of the core queue. If the queue is still not empty, the scheduler calls the adopted scheduling policy *sortQueue()* of core *cId* to sort the queue and moves to location Allocate2, whereafter it schedules the task corresponding to the first element in the queue. Function *sortQueue(cId)* refers to the scheduling policy of core *cId*, which is a core parameter in our model and can be FIFO, FPS or EDF.

5.2.2 Modeling of Shared Memories

This section describes the modeling of shared memories L2 cache and DRAM as well as their schedulers. We consider both shared memories, L2 cache and DRAM, as black boxes and assume that the effective access duration[3] for fetching data is constant, regardless of the physical location in the memory. The interference delay for an access request is defined by the waiting time from the issue of the access request until the access is granted.

DRAM=$\langle$*DRAMStruct, DRAMSched, DRAMAccessTime*$\rangle$ is composed of a structure *DRAMStruct* (abstracted using the behavioral model depicted in Figure. 5), a scheduler *DRAMSched* (Figure. 6), and the length of time for an effective access *DRAMAccessTime*. The DRAM access time simulates the duration of fetching data from a physical address in DRAM once the access is scheduled. This is in fact to enable our abstraction of the DRAM internal architecture

[3]The time interval from the instant when the access is granted until the data delivery instant.

to capture the delay for accessing a DRAM bank/row. Our DRAM model can be viewed as a one-bank memory that is shared between all cores, but it can easily be extended for several banks by just duplicating the DRAM structure and assigning each to one core only [33].

The DRAM behavior model (Figure. 5) is simple. However, its allocation is complex as we will see in the description of Figure. 6. DRAM is initially waiting at location Idle for an access request, either read or write. The DRAM can be allocated, by its scheduler, to a given core i performing a read request *DRAMReqR[i]?* and moves to location Read. Similarly, DRAM can be targeted with a write request *DRAMReqW?*.

One can see that for write access requests the identifier of the involved core is missing. This is because write requests are not blocking, thus no need to keep track of which core needs to be unlocked once the access is done. At locations Read and Write, the DRAM waits for the expiry of the access time $DRAMAccessTime$ then moves to location Done. From Read, once the access time expires the DRAM unlocks the involved core through a synchronization *dramAccessDone[currentCore]!*, whereas from location Write no unlock action is needed. From Done, DRAM notifies its scheduler that the current access is done.

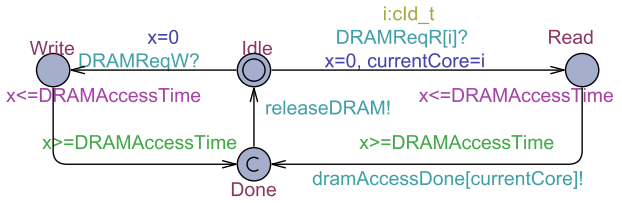

Figure 5: DRAM template model

We adopt the FR-FCFS policy to arbitrate accesses to DRAM. We assume that row opening and reload actions are instantaneous, so that we do not need to consider any preference based on the *already open row* policy [11]. This leads to considering the attribute $issueT$ (issue time) of each request as a *readiness*. Hence, we characterize each request to DRAM with a new attribute $arrivalT$, besides $issueT$. In fact, $issueT$ stores the time instant when the request is issued, whereas $arrivalT$ stores the instant when the request reaches the corresponding bank queue. Thus, we compare requests first based on their issue times (readiness) where an earlier request has priority over later ones. If requests have the same issue time, then the request having an earlier $arrivalT$ has priority over requests having later $arrivalT$. A sketch of the FR-FCFS is shown in Algorithm 1.

The DRAM scheduler is depicted in Figure. 6. Initially at location Init, upon the receive of an access request *dramReq[i]?* from any core i the DRAM scheduler inserts such a request together with the identifier of the requesting core into the queue and moves to location Allocate1. If such a request is a write ($rwAction{==}Write$), the requesting core will immediately be unlocked (*dramAccessDone[l]!*) as location is committed Allocate1. Moreover, if the write request is alone in the queue ($queue.length{==}1$) it will immediately be scheduled at location Unlocked.

In case of a read request ($rwAction{==}Read$), the DRAM scheduler does not unlock the requesting core after queuing the request. It just schedules the access (*DRAMReqR[DRAM.*

Algorithm 1 Sketch of the FR-FCFS algorithm

```
1: Int j
2: for each new request r do
3:     for i ∈ [0, q.length] do
4:         if (r.issueT < q[i].issueT) Or ((r.issueT =
   q[i].issueT) And (r.arrivalT < q[i].arrivalT)) then
5:             j := i
6:             i := q.length
7:         else
8:             j := i
9:         end if
10:    end for
11:    insert(r, j)
12: end for
```

queue.elt[0].core]!) if the current request is alone in the queue. In all of the four scenarios, the scheduler moves back to location Init.

Once an access request finishes, the scheduler is notified by the DRAM through a synchronization event *releaseDRAM?* and moves to Release while removing the head of the queue. If the queue is still not empty, the scheduler calls the algorithm FR-FCFS to sort the queue as other requests might have joined during the execution of the last access. At location Allocate2, the scheduler schedules the request in the first element of the queue *queue.elt[0]* using the appropriate channel ($DRAMReqR$ or $DRAMReqW$) according to the request nature; read or write.

Due to space limitations, we omit describing the shared cache L2 and its scheduler. In essence, L2 has the same elements as DRAM , except that it uses a separate queue to store its requests. Similarly, L2 scheduler has the same behavior as that of DRAM but it operates on the L2 queue using the cache coloring policy. However, since we do not consider the internal pages of L2, the coloring policy adopted in our framework behaves in similar way to FCFS policy.

Finally, a platform P is described as $\langle\langle PE_1, .., PE_m\rangle, DRAM, L2\rangle$. One can see that updating the specification of one platform ingredient does not necessarily affect the others.

5.3 System Model

In order to make our framework flexible, the application and platform are specified separately then mapped together. A system model S is given by an application $AP = \{T_1, .., T_n\}$, a platform $P = \langle\mathcal{PE}, DRAM, L2\rangle$ and a mapping $M : AP \rightarrow \mathcal{PE}$ assigning each task to a processing element $PE_i \in \mathcal{PE}$.

6. SCHEDULABILITY AND INTERFERENCE ANALYSIS

In this section, we describe how the schedulability will be analyzed inpresence of computed interference (isWCET). Moreover, in case of non-schedulability, based on core utilization and assumed delay per access request to shared memories we recommend task migration between cores so that a new potentially schedulable configuration can be identified.

6.1 Schedulability Analysis

In our framework, system schedulability is analyzed as a reachability property using symbolic model checking [5].

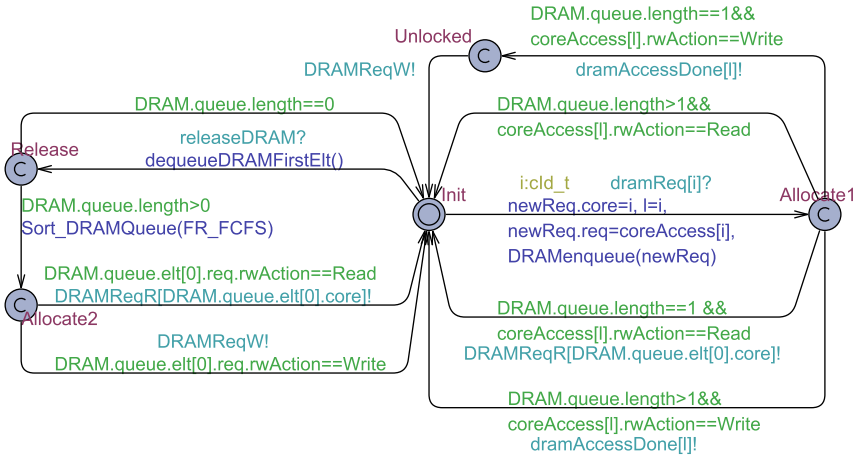

Figure 6: DRAM scheduler model

Following our task model, whenever a process misses its deadline it joins immediately the location DeadlineMiss (by which the global variable *error* is updated to true). Thus, the schedulability analysis process simply checks whether any task can reach its own DeadlineMiss location. Technically, to quantify on all tasks regardless of their identifiers we use the following CTL (Computation Tree Logic) query supported by Uppaal, where

$$\forall[]$$

denotes invariance over all reachable states, and ! denotes negation:

$$\forall[]\ !error \quad (1)$$

Using this query, the checker explores the whole state space and examines that in any state the value of variable *error* is false.

6.2 isWCET Estimation

The interference sensitive WCET (isWCET) of a task is the execution time including the effective execution (WCET), the delays caused by the interference to access shared memories as well as the effective access time to memories (L1, L2 and DRAM). Following the task model (Figure. 2), once a task is scheduled the clock $isWCET$ starts measuring time until the effective execution (WCET) is finished and all the task access requests are completed (location ExecDone). To do so, we simulate the system execution for X time units, each simulation runs for Y time units and accumulate the $isWCET$ values of given tasks $(T_1, .., T_n)$ using the following Uppaal SMC query:

$$simulate\ X\ [<= Y]\ \{isWCET[T_1], .., isWCET[T_n]\} \quad (2)$$

The simulation time Y should be greater than the least common multiplier of the task periods. In fact, the larger X and Y are the more accurate the results will be. To display the isWCET of a task T in terms of a probability distribution, the following SMC query can be used:

$$E[clk <= Y; X](max : isWCET[T]) \quad (3)$$

where the E operator identifies the trace (given X and Y constraints) in which maximum isWCET is obtained[4].

[4]Uppaal provides different presentations of the data collected during repeated runs, e.g. cumulative density function and distribution function. As long as any run reaches a isWCET > deadline this step has served its purpose and thereby we do not do further statistical analysis.

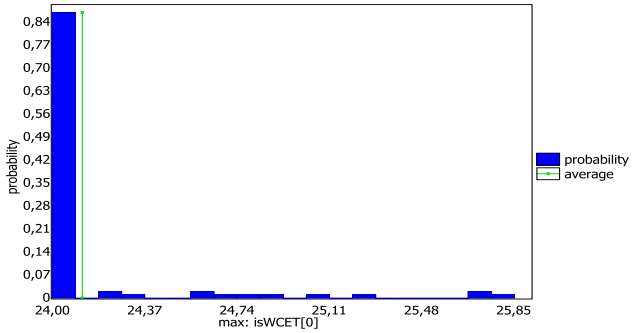

Figure 7: Probability distribution of isWCET of a given task.

Figure. 7 depicts a case where the x-axis shows the time units (simulation times) as used for all quantities below. It shows the probability distribution of the isWCET of a task T generated using query (3) where $X = 10^3$ and $Y = 10^4$. T runs in parallel with 3 other tasks mapped to 2 cores, each core serves 2 tasks. T has Prd=100, $WCET$=15, Dln=71, 13 read/write access to L2 ($WCRA_c = 13$) and 8 accesses to DRAM ($WCRA_m = 8$). Value 24 is the most likely because it has the highest probability (0.87), not far from the average isWCET 24.12.

6.3 Utilization Analysis

To estimate the utilization of cores, we need to run the execution simulation several times (X) each of which lasts for Y time units. We accumulate for each simulation the core utilization time via clock *utiliz[cId]*, and then consider the maximum value using the following SMC query:

$$simulate\ X\ [<= Y]\ utiliz[cId] \quad (4)$$

The utilization percentage of a given core is then obtained by dividing the accumulated utilization time over the total simulation time Y. Figure. 8 shows the average accumulated utilization time of 2 individual cores (C_0 and

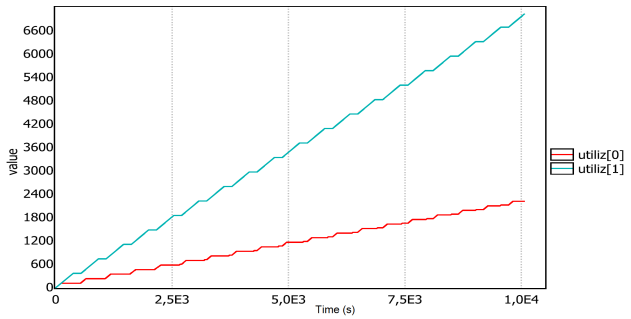

Figure 8: Simulation of cores utilization.

C_1) for 1000 simulations. Each simulation runs for 10000 clock ticks (query (4)). Thus, the utilization of core C_0 is $2223/10^4 * 100 = 22.2\%$.

6.4 Task Migration

In case a given system is not schedulable, it might be possible to find another schedulable configuration by moving some tasks between cores. However, in practice one might need to know whether a task can functionally be allocated to another core, since the IMA architecture standards assume that the system is structured in terms of functional blocks called partitions. As stated earlier, this paper does not model partitions and shows the tasks reallocation principle.

Based on the utilization of cores and average delay per access request to L2 and DRAM, we recommend to move tasks from cores with heavy workload to the ones having less workload. To migrate a task from one core to another, we estimate what would be the target core utilization given its current utilization, WCET of the task to be migrated and the workload generated by *read* accesses to L2 and DRAM ($WCRA_c^r$ and $WCRA_m^r$) of that task, i.e. number of read accesses multiplied by the average delay per request.

The fact that read requests are blocking, contributes considerably to the utilization by making the cores stall. On the other hand, write accesses are not blocking and have a smaller impact on the utilization, even though write accesses make the waiting queue longer, which some how might delay other read accesses. Currently, we ignore the workload generated by write accesses when probing migration scenarios.

Migrating a task T to a core C having a utilization U leads to a new utilization U' of C given by:

$$U' = U + \quad (T.WCET + T.WCRA_c^r \times a_c^r + T.WCRA_m^r \times a_m^r)/T.Prd \qquad (5)$$

where a_c^r and a_m^r are the average delays per read access requests to L2 and DRAM respectively. To obtain the average delay per access request to L2 and DRAM of a task T, we use query (4) on 2 other clocks U_c^r and U_m^r to accumulate the delays spent by that task when performing all its read requests to L2 and DRAM, respectively. Namely, when a task moves to location AccessRequest to perform a read access, the delay at that location will be accumulated to either U_c^r or U_m^r according to the request pattern. The average delays a_c^r and a_m^r are then obtained by dividing the accumulated delays U_c^r and U_m^r by the number of requests $WCRA_c^r$ and $WCRA_m^r$ respectively.

We compare different potential migration scenarios and recommend the configurations leading to a more balanced workload between cores. The new configurations will first be analyzed using Uppaal SMC via query (6), so that we obtain a probability on the system schedulability, where

$$<>$$

denotes existance of some path in which the respective property (here error) holds.

$$Pr[<= Y](<> \ error) \qquad (6)$$

In fact, this query runs a set of simulations of the system execution each for a duration Y, and checks how many (if any) simulations satisfy the property *error*, i.e. encounter a deadline miss. The probability is then obtained by dividing the number of simulations satisfying the property over the total number of simulations. Configurations having high probability to be schedulable will then be analyzed further using symbolic model checking (query (1)) to obtain deterministic evidence on the system schedulability.

As an illustration, the task T analyzed in Figure. 7 is first assigned to a core C_1 giving a high utilization 70.4% compared to core C_0 22.2% as shown in Figure. 8. We analyze what would be the utilization of C_0 if T is migrated to it, given its current utilization $U = 22.2\%$ and both $a_c^r = 0.79$ and $a_m^r = 0.97$ estimated during simulation. The utilization of C_0 if T is assigned to it would become $U' = 0.222 + 0.33 = 55.2\%$. The utilization of the original core C_1 will be reduced accordingly by 33%. The statistical analysis of schedulability for the system after migrating task T to core C_0 generates a probability $p \in [0, 0.009]$ that the system misses a deadline. We consider this probability to be low enough for further analysis of the configuration using symbolic model checking.

When analyzing this example using query (1), the analysis shows that the new configuration is actually schedulable. The idea behind using statistical analysis (SMC) first is that the analysis using SMC is cheap (in terms of resources, analysis time and memory space) compared to symbolic model checking. In this experiment, SMC consumed 8.53 seconds and 7.96MB memory space, whereas symbolic model checker consumed 46.39 seconds and 582.52MB.

7. CASE STUDY

In order to illustrate our method, we analyze a component of an autonomous vehicle system [13] presented earlier. The task functions are obtained from the PARSEC benchmark suite [4] and used to capture different components of complex real-time embedded applications such as sensor fusion and computer vision in an autonomous vehicle system.

Essentially, the application consists of 4 periodic tasks T_1 (StreamCluster), T_2 (Ferret), T_3 (Canneal) and T_4 (FluidAnimation) running on 2 identical cores (C_0 and C_1) sharing L2 and DRAM. T_1 and T_2 are assigned to C_0, whereas T_3 and T_4 execute on C_1. We calculated the WCRA by dividing the time spent by each task to fetch data over the average duration for one access from [13]. As the numbers of access requests to shared cache and DRAM are not explicitly distinguishable in [13], we rely on the analysis results obtained by Ye *et al.* [32] using the cache coloring policy, where only 22.2% of the access requests hit L2 while 77.8% of the requests need to access to DRAM.

The characteristics of the task set are shown in Table 1, where the offset is omitted since all tasks have offset equal to zero. $WCRA_c$ and $WCRA_m$ are given in terms of (reads;

writes). All time units are given in milliseconds (ms). The platform description is given in Table 2, where H is the access time for the core local cache. The time taken for effective accesses to L2 ($L2AccessTime$) and DRAM ($DRAM-AccessTime$) are $1/10^3$ and $1/10^2$ ms respectively.

Table 1: Attributes of the task set

Task	Prd	WCET	$WCRA_c$	$WCRA_m$	Dln
T_1	400	120	(40; 0)	(110; 10)	400
T_2	1200	130	(60; 60)	(136; 273)	1200
T_3	1800	500	(134; 266)	(705; 705)	1800
T_4	6000	440	(314; 626)	(1828; 1462)	6000

Each of the statistical analysis results of the case study has been calculated from 1000 simulations. Each simulation experiment runs for 1 million clock ticks. The accumulated utilization times of cores C_0 and C_1 are depicted in Figure. 9. During 1 million time units, cores accumulate 419983 and 360350 respectively. Thus, the average utilization of C_0 is 42% and the average utilization of C_1 is 36%. Schedulability analysis shows that the system is schedulable. The estimated maximum isWCETs of tasks T_1 and T_2, respectively T_3 and T_4, in terms of probability distributions, are depicted in Figure. 10 and Figure. 11, respectively. Accordingly, the most likely isWCET values of the case study tasks are 261, 262, 971 and 972 respectively. One may need to consider the maximum values for safety reasons. It is obvious that tasks performing more read requests, in particular to DRAM, produce longer isWCET.

Since the system requirements are met and the core workloads are modest and relatively comparable, there is no need to migrate any task. The longest analysis time spent by Uppaal is when running the statistical analysis to measure the core utilizations. In fact, for a simulation having 1000 runs each of which lasts for 1 million clock tick (a total of 10^9 clock ticks) Uppaal spent 3426.7 seconds. The performance and analysis results from the case study are encouraging but further studies are needed on scalability to larger systems.

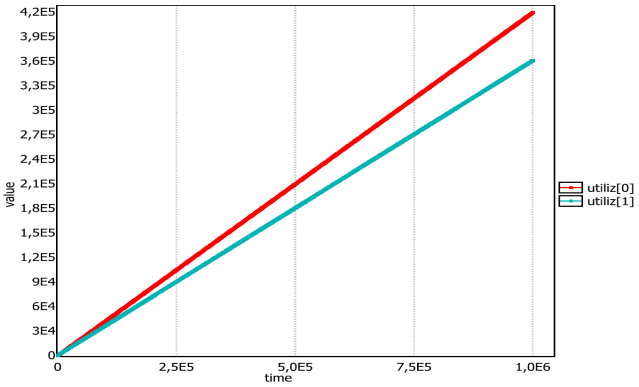

Figure 9: Accumulated core utilization.

Table 2: Platform description and mapping

Core	Sched policy	H	Mapped tasks
C_1	EDF	$1/10^3$	T_1, T_2
C_2	EDF	$1/10^3$	T_3, T_4

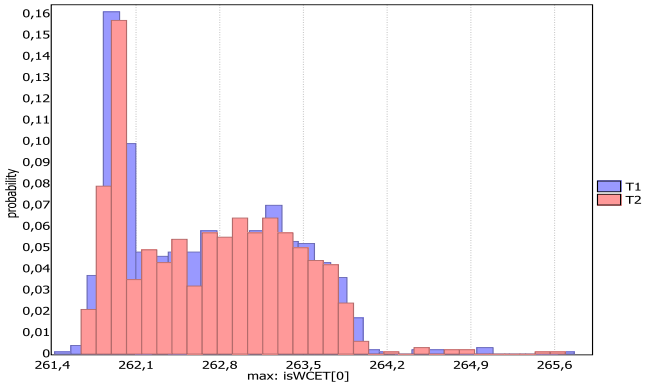

Figure 10: Probability distribution of isWCET for tasks (T_1, T_2).

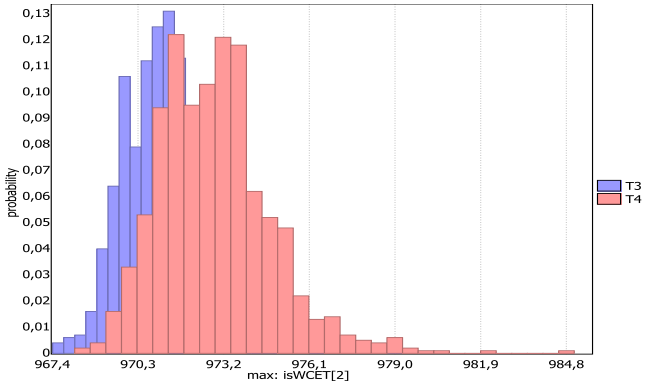

Figure 11: Probability distribution of isWCET for tasks (T_3, T_4).

8. CONCLUSION

This paper introduces a model-based framework for schedulability and memory interference analysis of multicore preemptive real-time systems. The framework captures both the multicore platform model, having a hierarchy of shared memories, and a system application given in terms of periodic preemptible tasks. In case of non-schedulability, we provide a technique to analyze the utilization of cores and study the impact of task migration with the goal to find potential mappings with which the system becomes schedulable. Our framework is realized using timed automata and stopwatch clocks of Uppaal, while the analysis of schedulability, memory interference and performance is performed using model checking technique.

As future work, interesting directions are to model and analyze more complex architectures, such as networked multiprocessor platforms and system on chips, analyze an actual avionic case study including partitions, fault containment aspects as well as cases where task re-allocation is necessary, and study the scalability of our framework.

9. REFERENCES

[1] Bound-T Execution Time Analyzer, http://www.bound-t.com/.

[2] B. Andersson, A. Easwaran, and J. Lee. Finding an upper bound on the increase in execution time due to contention on the memory bus in cots-based multicore systems. *SIGBED Rev.*, 7(1):4:1–4:4, Jan. 2010.

[3] G. Behrmann, A. David, and K. G. Larsen. *A Tutorial on Uppaal*, pages 200–236. Springer, 2004.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of PACT '08*, pages 72–81. ACM, 2008.

[5] A. Boudjadar, A. David, J. H. Kim, K. G. Larsen, M. Mikucionis, U. Nyman, and A. Skou. A reconfigurable framework for compositional schedulability and power analysis of hierarchical scheduling systems with frequency scaling. *SCP journal*, 113:236–260, 2015.

[6] J. Boudjadar, J. H. Kim, and S. Nadjm-Tehrani. Performance-aware scheduling of multicore time-critical systems. In *Proceedings of ACM/IEEE MEMOCODE*, pages 105–114, 2016.

[7] S. Chattopadhyay, C. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A unified wcet analysis framework for multi-core platforms. In *RTAS'12*, pages 99–108. ACM, 2012.

[8] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards WCET Analysis of Multicore Architectures Using UPPAAL. In *Proceedings of WCET'10*, pages 101–112, 2010.

[9] S. Hong, S. McKee, M. Salinas, R. Klenke, J. Aylor, and W. Wulf. Access order and effective bandwidth for streams on a direct rambus memory. In *Proceedings of HPCA'99*, pages 80–89. IEEE, 1999.

[10] H. W. Jin and S. Han. Temporal partitioning for mixed-criticality systems. In *ETFA'11*, pages 1–4. IEEE, 2011.

[11] H. Kim, D. de Niz, B. Andersson, M. H. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *RTAS'14*, pages 145–154. IEEE, 2014.

[12] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *ECRTS'13*, pages 80–89. IEEE, 2013.

[13] H. Kim, A. Kandhalu, and R. Rajkumar. Coordinated cache management for predictable multi-core real-time systems. Technical report, Carnegie Mellon Univ, 2014.

[14] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *SCP Journal*, 69:56 – 67, 2007. Special issue on Experimental Software and Toolkits.

[15] B. Lisper. SWEET - A tool for WCET flow analysis (extended abstract). In *Proceedings of ISoLA'14*, volume 8803 of *LNCS*, pages 482–485. Springer, 2014.

[16] A. Löfwenmark and S. Nadjm-Tehrani. Experience report: Memory accesses for avionic applications and operating systems on a multi-core platform. In *Proceedings of ISSRE'15*, pages 153–160. IEEE, 2015.

[17] M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Proceedings of RTSS'10*, pages 339–349. IEEE, 2010.

[18] J. Madsen, M. R. Hansen, K. S. Knudsen, J. E. Nielsen, and A. W. Brekling. System-level verification of multi-core embedded systems using timed-automata. In *IFAC'08*, pages 9302–9307, 2008.

[19] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *Proceedings of MICRO'06*, pages 208–222. IEEE Computer Society, 2006.

[20] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *Proceedings of ECRTS'14*, pages 109–118. IEEE, 2014.

[21] A. Oliveira Maroneze, S. Blazy, D. Pichardie, and I. Puaut. A Formally Verified WCET Estimation Tool. In *Proceedings of WCET'14*. OASIcs, 2014.

[22] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings ISCA'00*, pages 128–138. ACM, 2000.

[23] J. Roseen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proceedings of RTSS'07*, pages 49–60. IEEE, 2007.

[24] RTCA co. DO-297/ED-124 - Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations, 2005.

[25] A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *Proceedings of RTAS'10*, pages 215–224. IEEE, 2010.

[26] R. Sen and Y. N. Srikant. WCET estimation for executables in the presence of data caches. In *Proceedings EMSOFT '07*, pages 203–212. ACM, 2007.

[27] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In *HPCA'13*, pages 639–650. IEEE, 2013.

[28] L. Tan. The worst-case execution time tool challenge. *STTT journal*, 11(2):133–152, 2009.

[29] A. Wilson and T. Preyssler. Incremental certification and integrated modular avionics. In *Digital Avionics Systems Conference. DASC'08. IEEE/AIAA 27th*, pages 1.E.3–1–1.E.3–8, 2008.

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

[31] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE Transaction on Computers*, 2016.

[32] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: A dynamic cache partitioning system using page coloring. In *Proceedings of PACT'14*, pages 381–392, 2014.

[33] H. Yun, R. Pellizzoni, and P. K. Valsan. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *ECRTS'15*, pages 184–195. IEEE Computer Society, 2015.

[34] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Proceedings of ECRTS'12*, pages 299–308. IEEE Computer Society, 2012.