Performance-aware Scheduling of Multicore Time-critical Systems*

Jalil Boudjadar[1], Jin Hyun Kim[2], Simin Nadjm-Tehrani[1]
[1] Linköping University, Sweden
[2] University of Pennsylvania, USA

***Abstract*—Despite attractiveness of multicore processors for embedded systems, the potential performance gains need to be studied in the context of real-time task scheduling and memory interference. This paper explores performance-aware schedulability of multicore systems by evaluating the performance when changing scheduling policies (as design parameters). The model-based framework we build enables analyzing the performance of multicore time-critical systems using processor-centric and memory-centric scheduling policies. The system architecture we consider consists of a set of cores with a local cache and sharing the cache level L2 and main memory (DRAM). The metrics we use to compare the performance achieved by different configurations of a system are: 1) utilization of the cores; and 2) the maximum delay per access request to shared cache and DRAM. Our framework, realized using UPPAAL, can be viewed as an engineering tool to be used during design stages to identify the scheduling policies that provide better performance for a given system while maintaining system schedulability. As a proof of concept, we analyze and compare 2 different cases studies.**

I. INTRODUCTION

Today's embedded systems demand increasing computing power to accommodate the growing software functionality. Multicore platforms are finding their way into automotive and avionic areas where they are a target to deploy safety-critical applications. In avionic systems, an ultimate goal of using multicore platforms is to leverage the computing capabilities, reduce the weight of on-board computing equipment and lower the energy consumption. Multicore avionic systems are usually built by integrating different subsystems to enable incremental Design and Certification (iD&C) [31], recommended by the standard Integrated Modular Avionics (IMA) architecture [26]. However, due to the safety requirements performance can be sacrificed, up to a certain degradation level, if the safety and reliability are in peril. Hence, performance is aimed to be as high as the system safety permits.

In contrast to the classical federated architecture, IMA supports functions related to different subsystems to share the same computing platform. Such a support is implemented using partitioning, where different partitions running on different cores compete for the access to a set of shared resources, such as shared memories and buses. However, having applications running simultaneously and requesting access to shared memories concurrently leads to interference. The non predictability resulting from interference at any shared memory level may lead to violation of the timing properties in safety-critical real-time systems. Moreover, the interference on shared memories leads cores to stall, in particular for read access requests, so that the performance is affected as well. Hence, bounding memory interference is a key factor to guarantee schedulability and improve performance, e.g. [16], [21].

Many recent works have been devoted to the analysis of safety and predictability of multicore systems [19], [22], [3], [8], [23]. Fewer studies focus on the compensation of performance while maintaining the timing predictability guarantees [30], [27], [24], [28]. Processor-centric scheduling policies are no longer sufficient to guarantee schedulability without accounting for unacceptable pessimism, in particular for memory-intensive applications running on large multicore platforms [32]. To leverage the performance of multicore systems while avoiding crippling pessimism, the notion of memory-centric scheduling has been introduced [32]. However, the performance gain obtained when using memory-centric scheduling depends on the system application and could be less important compared to the performance achieved when running processor-centric scheduling. When designing a software system, there is a potential margin within which different configurations, obtained by slightly tweaking some of the design parameters, can functionally behave in the same way as the original design while achieving better performance and satisfying the predictability requirements [27], [23]. Moreover, it is not always trivial to classify a system as memory-intensive or processor-intensive as it may include a mix of applications with different natures. Hence, identifying the scheduling policies leading to a better performance needs extra analysis processes.

This paper proposes a formal, tool-supported performance-aware schedulability analysis for multicore systems. Our model-based framework enables analyzing the schedulability and performance of multicore systems while varying the core scheduling policies so that configurations with better performance are identified. We adopt the Asymmetric Multi-Processing (AMP) [15] avionic scheduling architecture, where partitions are statically allocated to specific cores. This choice is motivated by the need to prevent error propagation between the applications running on different cores. The scheduling strategies we consider are memory-centric and processor-centric (classic CPU scheduling policies). The platform architecture consists of a set of cores, having each a local cache, and sharing the cache level 2 (L2) and DRAM. Application tasks are modeled with fine-grained behavior such as read

* This work was supported by the Swedish Governmental Agency for Innovation Systems under grant number NFFP6-2014-00917.

and write accesses to L2 and DRAM in addition to the classic scheduling parameters. The performance metrics we consider are the *utilization of cores* and *maximum delay per access request* to shared memories (L2 cache and DRAM). We provide rigorous schedulability analysis using symbolic model checking, while performance metrics are measured using statistical model checking. To sum up, the contributions presented here are:

- A framework for modeling and formal analysis of multicore systems on platforms with a hierarchy of shared memories, and fine-grained application descriptions.
- A method for understanding trade-offs using two metrics to evaluate system performance: utilization of cores and maximum interference delay per access request to shared memories.
- Providing templates in a formal analysis tool to support automatic creation and multi-criteria comparison of system configurations, both with processor-centric and memory-centric scheduling.

The rest of the paper is organized as follows: Section II reviews the related work. Section III describes the necessary background. Section IV introduces the performance metrics. Section V presents UPPAAL templates of our framework. Section VI describes the performance analysis. Section VII is the analysis of 2 case studies. Section VIII is a conclusion.

II. Related Work

The performance of multicore systems has been studied intensively. Here we focus on works manipulating real-time scheduling strategies to achieve better performance.

Putrycz et al.[24] present a survey on performance analysis techniques for COTS-based systems and discuss the trade-off between the design models used to capture system design and the performance attributes to be measured. They identified delays and throughput as main performance metrics. In a similar way, Chhibber and Garg[9] present a set of performance metrics (throughput, energy consumption, memory contention, *etc*) and criteria (time, cost, accuracy) to compare a set of multicore platforms. Löfwenmark and Nadjm-Tehrani [21] presented a validation methodology for benchmarking of WCET using avionic platforms.

Major recent works [34], [16] introduce analytical frameworks to calculate memory interference in multicore systems. The interference of the parallel requests is calculated based on banks interference as well as *row-opening* and *precharge* (moving data back from a row-buffer to a row). Since each request interference is calculated separately, a relevant question is how these frameworks deal with memory-intensive systems where each process performs thousands of requests.

Teodoro et al. [30] propose new scheduling strategies, Performance-Aware Dequeue Adapted Scheduler (PADAS) and Performance-Aware Multiqueue Scheduler (PAMS), to improve the performance of multicore systems. The platform consists of a set of cores with different clock frequencies. The execution model is implemented in terms of code annotations. In PADAS, tasks are sorted in one ready queue (global) based on the acceleration (speedup) expected by each task. A task is assigned to a core based on whether the core frequency satisfies the task acceleration or not. In PAMS, local scheduling is adopted for each core type so that tasks having certain range of acceleration are sorted in the appropriate queue according to a descending order of the acceleration. This framework relies on an efficient task-to-core mapping to achieve better performance. Tasks response time is the performance metric used to compare configurations having different scheduling policies. Our work aligns with this approach [30], but it differs by using system benchmarks rather than code annotations. Moreover, by adopting the AMP scheduling strategy we assume that identifying the optimal task-to-core mapping is beyond the scope of our paper.

Diamond et al. [12] study the performance of multicore systems and discuss the challenges of measuring the performance. Due to the strong influence of data access to the shared memories (interference) on the system execution, shared cache capacity, shared memory bandwidth and DRAM page conflicts are identified as performance determining factors. A system performance is relatively *good* if the flops, instructions, and loads per cycle, which describe how hard cores and memories are working, approach maximum values.

Subramanian et al. [28], identify memory interference-induced slowdown as a performance metric. Slowdown is the delay experienced by the application, due to waiting for access to shared resources, compared to the case when the application runs alone on the platform. The authors build a simple model to capture and analyze such a performance metric and develop a scheduling scheme (MISE-Fair) to minimize the maximum slowdown. In essence, MISE-Fair estimates the slowdown of each application and redistributes the memory bandwidth to reduce the slowdown of the most slowed-down application. However, minimizing the DRAM-related slowdown of tasks may impact scheduling at the level of cores.

Yao et al. [32] motivate the use of memory-centric scheduling for multicore systems as processor-centric scheduling policies introduce unacceptable pessimism. The authors propose a *global* memory-centric scheduling algorithm and study the execution slowdown while varying the number of cores. A conclusion is that the memory-centric policy can schedule twice as many tasks compared to processor-centric scheduling. Our work aligns with this approach [32], but we adopt *local* scheduling (AMP), and arbitrary memory access patterns during task executions rather than dedicated access patterns.

Compared to the state of the art [32], [30], [34], we use more detailed models to capture multicore platform architectures and fine-grained application descriptions. Platforms include a hierarchy of shared memories, and application tasks have explicit numbers of read and write requests to each shared memory. We use model checking to analyze schedulability and performance while varying some design parameters (scheduling policies). To make the system behavior more realistic by spreading out the access requests to L2 and DRAM non-deterministically during task execution rather than using dedicated phases [32]. Our model-based framework supports automatic configuration creation. In fact, by feeding our framework with a single set of parameters at least 5 different configurations can be created. The difference from one configuration

to another is the scheduling policies of the individual cores. The scheduling policies currently included are: First-In-First-Out (FIFO), Fixed Priority Scheduling (FPS), Rate-Monotonic (RM), Earliest Deadline First (EDF) and memory-centric. The performance of each configuration is automatically captured using model checking, and evaluated with processor-related (utilization) or memory-related (access request delay) metric.

III. Background

This section introduces benchmarking, as a technique to measure tasks execution time (WCET) and maximum number of memory accesses (WCRA), as well as the background of our work.

In our work, we use the cache coloring policy (CC) [17] to arbitrate concurrent access requests to the shared L2 cache . In addition, we adopt the policy First Ready-First Come First Serve (FR-FCFS) [25], [16] used by modern COTS-based memory controllers to schedule the DRAM access requests. The system application is given by a set of task sets, each of which is statically assigned to a given core. Besides WCET and deadlines, tasks have explicit read and write access numbers for shared cache L2 and DRAM. We distinguish between read and write access requests to shared memories as read actions make cores stall, while write actions are not blocking and can be performed using dedicated buffers. Tasks assigned to the same core will be scheduled using either memory-centric or processor-centric scheduling.

A. Systems Benchmarking

Throughout this section we describe how the inputs required by our model-based framework are obtained from an actual system, in particular WCETs and WCRAs. Putrycz et al [24] propose benchmarking to measure a set of execution attributes and performance characteristics of software systems. The output of a system benchmarking will be used to reconstruct a behavioral model of the system, which can be used in turn for simulation and analysis purposes. Benchmarking is known as an expensive operation that involves several iterative rounds in order to reach conclusive decisions.

Flow analysis [29], [8] is a technique to estimate the WCET of a program. It consists of simulating, or concretely running, a program in isolation and calculating WCET. Technically, static analysis tools use symbolic execution engines to identify potential execution paths. Such representations can be structured in terms of control flow graph (CFG). WCET is then the time elapsed when executing the longest path of the CFG. Static analysis has been applied to the analysis of software systems via different analysis tools, e.g. SWEET [29].

System profiling [35], [16], [20] is a measurement-based approach to estimate the execution time of a process and how many times it 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 measure WCET and track how many times the process accesses a given shared memory. The number of accesses can be obtained using Performance Monitor Counters (PMCs) [20], [35], available in certain multicore platforms. PMCs provide the ability to count L2 misses for both data and instructions per core.

Profiling and static analysis techniques are not mutually excluded. Both techniques can be used to measure WCET and WCRA.

B. Processor-centric Scheduling

To leverage the processing performance of a computing system, a processing unit (core in our context) can be assigned more than one task, however only one task can execute effectively at any point in time. The arbitration between the execution of different tasks 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. Such a ready queue can be either local for a given core (local scheduling) or common for a set of cores (global scheduling). The key factor in selecting a ready task can be assigned to the priority, remaining execution time, etc.

C. Memory-centric Scheduling

With the goal to reduce memory interference in multicore systems, a recent alternative to schedule memory-intensive application tasks is the use of a memory-centric policy [32]. Tasks assigned to the same core are sorted in the ready queue according to a decreasing order of their WCRA (numbers of memory accesses).

In our framework, we distinguish between the access numbers to L2 and DRAM. Hence, when comparing 2 tasks we prioritize first the task having more DRAM access requests as DRAM access is more expensive than accessing shared cache L2. If both tasks have the same DRAM access number of requests we refer then to their L2 access numbers where task having higher number will be scheduled first.

D. Shared Memory Access Scheduling

In order to enhance the processing performance of multicore platforms, some modern multicore processors [1] consider a shared L2 cache besides private caches (L1-cache). The primary reason of sharing a cache between different cores is to reduce the access requests to the main memory DRAM [22].

Cache coloring policy [17], [33] is an algorithm to control the access to the shared cache L2. 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 entails avoiding the 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.

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

Similarly, DRAM controllers in modern COTS-based systems use First Ready-First Come First Serve (FR-FCFS) policy [25], [16] to schedule accesses. FR-FCFS considers a detailed DRAM architecture structured in terms of banks, rows and columns. The access requests can target different banks separately, where they will be queued in the corresponding bank queue with a special preference to read requests since they cause the processor to stall. Access requests will be sorted at each bank queue first according to their readiness. Thereafter, 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. A request hits a row buffer if it has high row buffer locality, i.e it targets a row being recently accessed.

In our framework, we do not consider the detailed internal architecture and size of DRAM and shared cache. We focus instead on measuring the delays caused by the concurrent accesses. Cache recoloring and estimation of the optimal cache size for each application are beyond the scope of this paper.

E. Statistical Model Checking

We use UPPAAL [10] to perform a formalized statistical simulation of our models, known as Statistical Model Checking (SMC). SMC enables quantitative performance measurements instead of the Boolean (true, false) evaluation that symbolic model checking techniques provide. UPPAAL enables describing systems as a set of concurrent processes called *network of timed automata*. Each timed automaton is represented by a behavioral component called *template*. Processes are statically created by instantiating the underlying templates with actual parameters. Basically, a timed automaton is a state-transition system where transitions are labeled with potential synchronization events (described with cyan statements in our models). The execution of a transition is guarded by potential time and data constraints (described with green statements in our models), and may lead to update variables and control flow (described with blue statements in our models). Moreover, the stay duration of a timed automaton at a given location can be subject to a time constraint called *invariant* (described with pink statements in our models) where it is allowed to wait only while the invariant constraint is satisfied. The system safety and liveness properties can be expressed using Computation Tree Logic (CTL). A variation of UPPAAL called UPPAAL SMC enables performing statistical and quantitative analysis. We can summarize the main features of UPPAAL SMC as follows:

- Stopwatches [7] are clocks that can be stopped and resumed without a reset. They are very practical to measure the execution time of preemptive tasks.
- Probability evaluation `Pr[bound] (P)` for a property `P` to be satisfied within a given simulation time and/or number of runs specified by `bound`.
- Simulation and estimation of the expected minimum or maximum value of expressions over a set of runs, `E[bound](min:expr)` and `E[bound](max:expr)`, for a given simulation time and/or number of runs specified by `bound`.

Statistical model checking does not provide complete certainty that a property is satisfied, but only verifies it up to a specific confidence level [11], given as an analysis parameter.

IV. PERFORMANCE METRICS

This section defines the performance metrics we have chosen as a basis to compare memory-centric and processor-centric scheduling policies. We analyze the performance on a set of independent runs $Runs = \{\pi_1, \pi_2, \ldots\}$, randomly generated according to a probabilistic semantics [11]. A run π of a system S is an infinite sequence:

$$\pi = s_0(t_0, e_0)s_1(t_1, e_1)\ldots s_n(t_n, e_n)\ldots$$

where each state s_i gives information about a system configuration including the state of each task (e.g. idle, ready, running, blocked) and resource (e.g. idle, occupied) at state i; s_0 is the initial state. Each e_i indicates an event (triggering, queued or completing) signifying a transition from state s_i to s_{i+1}. Timestamp t_0 indicates the time from system initiation until event e_0. Every subsequent timestamp t_i (with $i \geq 1$) indicates the duration between events e_{i-1} and e_i. Moreover, for a given run π we introduce the following:

- $InUse^s_\pi(C) \in \{0, 1\}$ states whether the core C is in use in state s of π or not [2]. A core is in use if it is either effectively executing a workload or stalling due read access requests to L2 and DRAM.
- $Issued^C_\pi(req)$ indicates at which state of π the access request *req* performed by core C has been issued.
- $Granted^C_\pi(req)$ states the earliest state at which the request *req* performed by core C has been granted.

Definition 1 (Core utilization): The utilization $U^{\mathcal{L}}_\pi(C)$ of a core C for a given run π up to the time bound (simulation length) $\mathcal{L}$ is given by the following:

$$U^{\mathcal{L}}_\pi(C) = (\limsup_{t\to\mathcal{L}} \frac{\sum_{t_{i+1}\leq\mathcal{L}} ((t_{i+1} - t_i) \times InUse^{s_{i+1}}_\pi(C))}{\mathcal{L}}) \times 100$$

The average utilization of a core C over the set of runs *Runs*, analyzed for the same time interval length $\mathcal{L}$, will be the sum of core utilizations of the individual runs over the number of runs:

$$U^{\mathcal{L}}_{Runs}(C) = \frac{\sum_{j=1}^{j=|Runs|} U^{\mathcal{L}}_{\pi_j}(C)}{|\ Runs\ |}$$

Definition 2 (Interference of access requests): We define the interference delay of an access request *req* to a shared memory performed by a core C in a run π as follows: $ReqDelay^C_\pi(req) = Granted^C_\pi(req) - Issued^C_\pi(req)$.

In a similar way, we define the maximum delay of the requests Req_C performed by a core C over a run π by:

$$ReqDelay^C_\pi = \max(ReqDelay^C_\pi(req \mid req \in Req_C))$$

[2] In fact, we handle Boolean values as either 0 or 1.

The average of maximum interference delays performed by a core C over the set of runs *Runs* can be calculated as follows:

$$ReqDelay^{C}_{Runs} = \frac{\sum_{j=1}^{j=|Runs|} ReqDelay^{C}_{\pi_j}}{|\ Runs\ |}$$

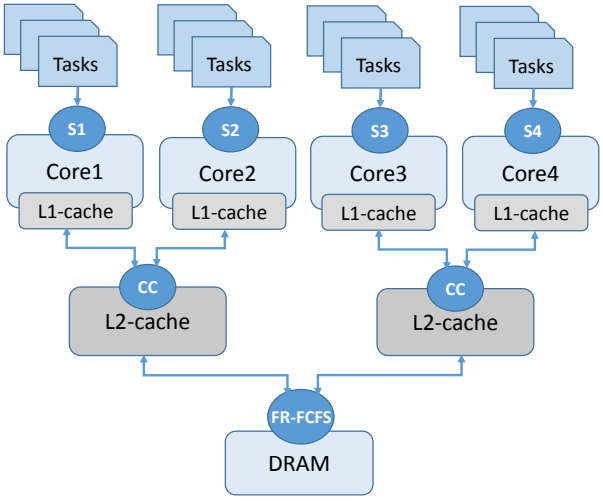
Interference delays are important factors for the tasks' response time and schedulability, i.e. the longer interference delays are the longer response time will be. As access requests to L2 and to DRAM do not have the same interference delays and impact on the system execution, later on we distinguish between $L2\_ReqDelay^{C}_{Runs}$ and $DRAM\_ReqDelay^{C}_{Runs}$ as the interference delays of a given core C to access L2 and DRAM respectively.

V. System Model Description

The platform we consider consists of a set of cores (processing elements), one shared cache level and a shared DRAM. The application is given by a set of periodic tasks mapped to different cores. However, sporadic tasks can be considered as well without any change in the system models. Mainly, application tasks are characterized by their computation times WCET (measured when each task runs in isolation), and the worst case resource access numbers (WCRA) [22] to DRAM and shared cache L2 for both read and write patterns. Access requests to shared memory are non-deterministically spread out along the task execution. The reason behind this is that the task execution, and thereby the issue time of data requests, may vary from one period to another following the changes in the computation environment.

We use $\mathcal{T}$ for the set of tasks and $\mathcal{C}$ for the set of cores. The overall system architecture we consider is depicted in Fig. 1, where $S1$, $S2$, $S3$ and $S4$ are scheduling policies.

Fig. 1. Simplified multicore system architecture.



The assumptions about the systems we consider are:

- Tasks are periodic, and consume the entire budget WCET for each period.
- Tasks assigned to the same core are arbitrated using a local scheduler.
- We consider a local cache (L1) for each core, only one shared cache (L2) and one DRAM for all cores.

A. 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 functionality is abstracted at task level using WCET, and WCRA for both L2 and DRAM. WCET is the pure execution time, captured in isolation when a task runs alone on a single core. 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 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. $WCRA^{r}_{c}$, $WCRA^{w}_{c}$, $WCRA^{r}_{m}$ and $WCRA^{w}_{m}$. Such attributes are identifiable using program/cache analysis tools [14], e.g. PAG tool [2], on a given platform architecture.

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 indicating whether it is a read (r) or write (w) action. As we need to keep track of when the access requests are issued, so that FR-FCFS algorithm determines the priorities of requests targeting DRAM, we introduce the attribute *issueTime* which is initially empty and will be initialized by a core when the access request is effectively triggered. Accordingly, an access request is given by $req = \langle pattern, RW, issueTime \rangle$. $WCRA^{r}_{c}$ and $WCRA^{w}_{c}$, respectively $WCRA^{r}_{m}$ and $WCRA^{w}_{m}$, of a task are then the numbers of read and write accesses to L2, respectively DRAM.

Definition 3 (Task structure): A task T is given by $\langle Prd, Offset, WCET, WCRA^{r}_{c}, WCRA^{w}_{c}, WCRA^{r}_{m}, WCRA^{w}_{m}, Dln, Pri, C \rangle$ where *Prd* is the task period, *Offset* is the periodic offset, *WCET* is the pure execution time, *Dln* is the relative deadline, and *Pri* is the priority level associated to T. $WCRA^{r}_{c}, WCRA^{w}_{c}, WCRA^{r}_{m}$ and $WCRA^{w}_{m}$ are described above. C is the core identifier to which the task will mapped.

Fig. 2. Task template model

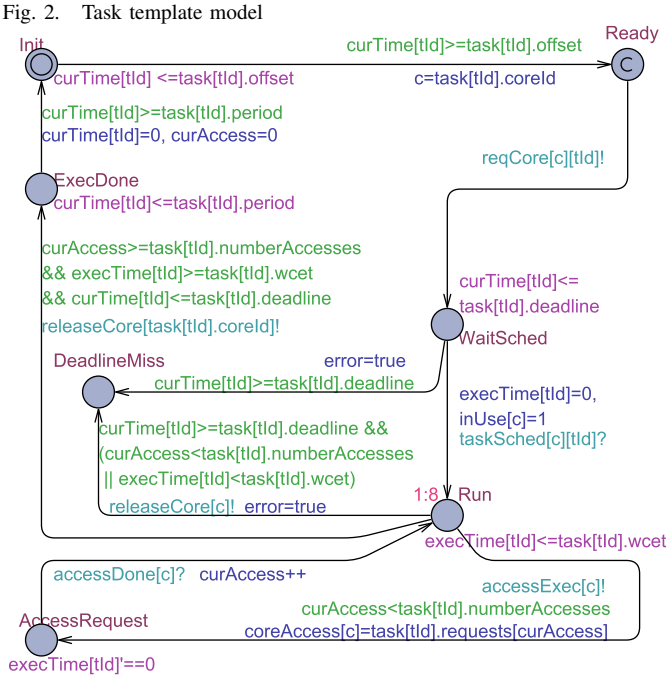
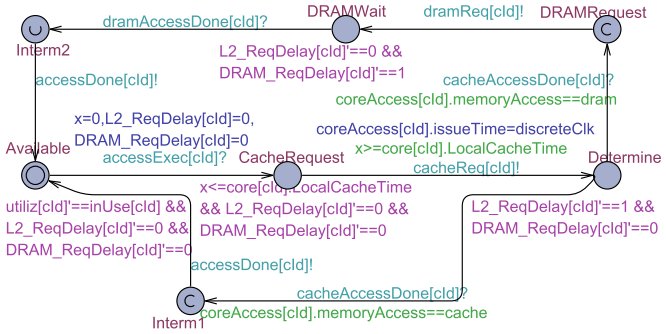


Fig. 3. Core template model



Our UPPAAL task template model is depicted in Fig. 2. We associate to each task an identifier `tId` as a template parameter to distinguish between different tasks. The task starts at location Init, and once the offset expires it moves to location Ready to request the core (C) it is mapped to, through a synchronization 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 moves to location Run to execute and the status of the corresponding core gets updated *inUse[c]=1* to start accumulating the core utilization. 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 the expiry of *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]*. 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 achieved before deadline, or it moves to location DeadlineMiss otherwise. The task template can be instantiated for different tasks by just providing the aforementioned parameters.

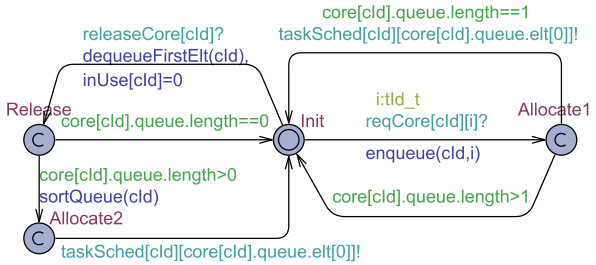
B. Platform Model

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

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 Fig. 3.

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

Fig. 4. Scheduler template model



a clock can stop and resume according to the core status *inUse[cId]*, i.e. manipulated by the task template. 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 *issueTime=discreteClk*. At that location, the clock measuring the core delay for the current access to L2 starts (*L2_ReqDelay[cId]'==1*). Once the access to L2 hits (*memoryAccess==cache*) and terminates, the clock *L2_ReqDelay[cId]* gets stopped (*L2_ReqDelay[cId]'==0*) and the core moves back immediately to location Available to continue executing the assigned workload. Otherwise, once the L2 access terminates and misses (*memoryAccess==dram*) the core requests access to DRAM and joins immediately the location DRAMWait, whereby the clock measuring the core delay for the current access request is released (*DRAM_ReqDelay[cId]'==1*). Once the current access is completed, clock *DRAM_ReqDelay[cId]* gets stopped while holding the measured delay. Upon the release of new access requests, from location Available, clocks *L2_ReqDelay[cId]* and *DRAM_ReqDelay[cId]* are reset.

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 request completes. 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, so that the task moves back as well from location AccessRequest to Run and accounts a granted access (*curAccess++*). As it is not possible to entitle a transition with two synchronization events in UPPAAL, we introduce two intermediate locations Interm1 and Interm2. Thus, we create a sequence of 2 synchronizations without any delay between them. We use *urgentness* and *committedness* of UPPAAL to enforce time to not elapse at a given location (locations marked with `U` and `C`).

Fig. 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. Once the core is released by the current running

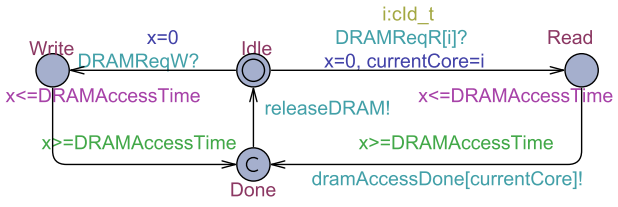
task, through a signal on channel *releaseCore*, the scheduler moves to Release while removing the first element of the queue. If the queue is still not empty, the scheduler calls the adopted scheduling policy *sortQueue()* of core cId to sort the queue and moves to location Allocate2, whereafter it schedules the task corresponding to the new 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, EDF or memory-centric.

2) Modeling of Shared Memories: This section describes the modeling of L2 cache, DRAM and their schedulers. A DRAM=⟨*DRAMStruct, DRAMSched, DRAMAccessTime*⟩ is given by a structure *DRAMStruct* (Fig. 5), a scheduler *DRAMSched* (Fig. 6) and the time duration 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 to capture the delay for accessing a DRAM bank/row. Our DRAM model can be viewed as a one-bank memory which 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 [34]. Moreover, we assume that the instantiation of our L2 and DRAM models satisfies the JEDEC standard [1], which dictates the operation and timing constraints of memory devices.

The DRAM structure model (Fig. 5) is initially waiting at location Idle for an access request, either read or write. 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 completed. 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.

Fig. 5. DRAM template model

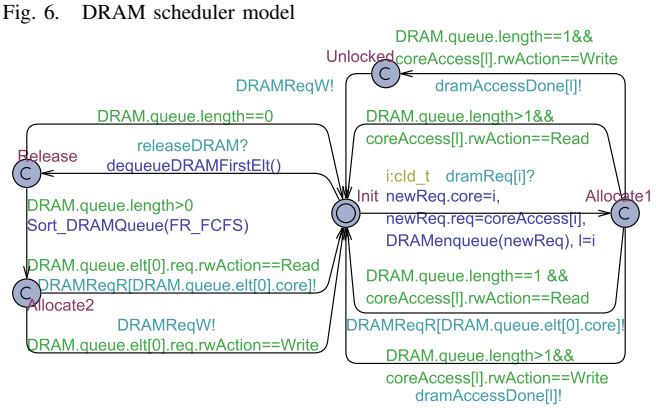


We adopt the FR-FCFS policy to arbitrate accesses to DRAM. We assume that row opening and reload actions are instantaneous, so that we do not need to consider any preference based on the *already open row* policy [16]. This leads us to consider the attribute *issueTime* of each request as a *readiness*. Hence, we characterize each request to DRAM with another new attribute *arrivalT*, besides to *issueTime*. In fact, *issueTime* 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*.

The DRAM scheduler is depicted in Fig. 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[1]!*) 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, but it just schedules the access (*DRAMReqR[DRAM.queue.elt[0].core]!*) if the current request is alone in the queue, i.e. DRAM is not occupied. In any case, the scheduler moves back to Init.

Once an access request is finished, 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 join during the execution of the last access. At location Allocate2, the scheduler schedules the request in the head of the queue using appropriate channels (*DRAMReqR* or *DRAMReqW*) according to the request pattern; read or write.

Fig. 6. DRAM scheduler model



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 scheduler 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 we adopt behaves in similar way to FCFS policy.

Finally, a platform *P* is given by $\langle\langle PE_1, .., PE_m\rangle, DRAM, L2\rangle$. One can see that updating the specification of one platform ingredient does not necessarily need to update the others.

C. 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 $PE_i \in \mathcal{PE}$.

VI. Performance Analysis

A. Schedulability Analysis

In our framework, the system schedulability is analyzed as a reachability property using symbolic model checking [5]. Whenever a process misses its deadline it joins immediately the location DeadlineMiss (where the global variable *error* is updated to true). Thus, the schedulability analysis simply checks whether any task can reach its own DeadlineMiss location. To quantify on all tasks regardless of their identifiers we use the following CTL query supported by Uppaal:

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

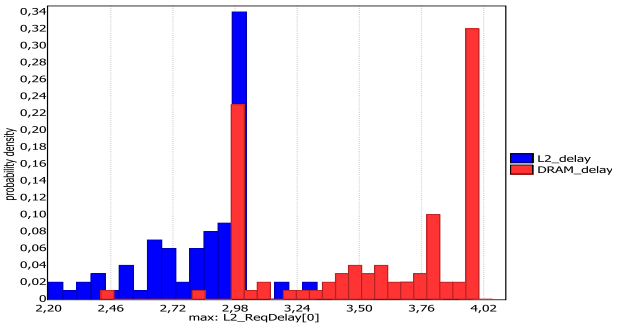
B. Memory Interference Analysis

To analyze the delays of access requests performed by a given core, we need to run the execution simulation several times (X) each of which lasts for Y time units. The simulation time Y should be greater than the least common multiplier of periods of the tasks mapped to such a core. In fact, the larger X and Y are the more accurate the results will be. To display the maximum interference delays of a core C, to access L2 and DRAM respectively, in terms of probability distributions we use the following SMC queries:

$$E[clk <= Y; X](max : \textit{L2\_ReqDelay}[C]) \quad (2)$$
$$E[clk <= Y; X](max : \textit{DRAM\_ReqDelay}[C]) \quad (3)$$

Fig. 7 shows the probability distributions of the request delays to L2 and DRAM of a core C where $X = 10^3$ and $Y = 10^4$. Values 2.96 and 4 are the most likely L2 and DRAM access delays respectively because they have the highest probabilities.

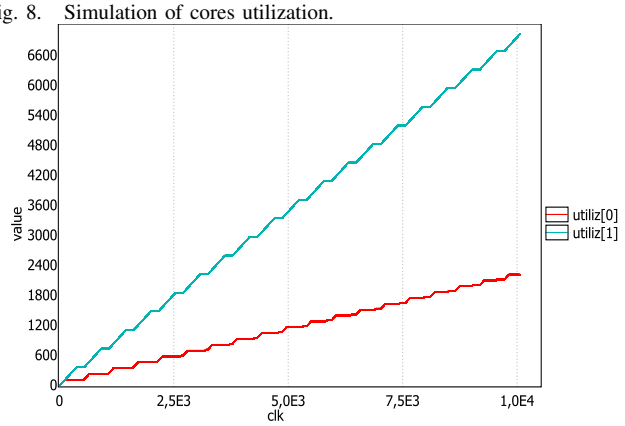
Fig. 7. L2 and DRAM Request delays.



C. Utilization Analysis

To analyze 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]*, we consider then the maximum value using the following SMC query:

Fig. 8. Simulation of cores utilization.



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

The utilization degree of a given core is then obtained by dividing the accumulated utilization time over the total simulation time Y. Fig. 8 shows the average accumulated utilization time of 2 individual cores (C_0 and C_1) for 1000 simulations. Each simulation runs for 10000 clock ticks (query (4)). Thus, the utilization of core C_0 is $2223/10^4 * 100 = 22.2\%$.

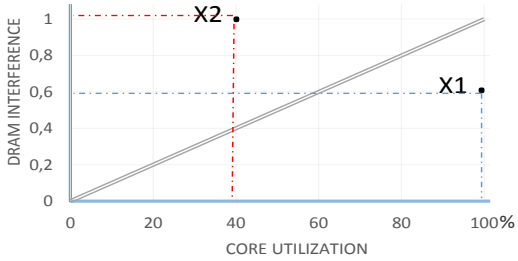
The fact that read requests are blocking, they contribute majorly in the cores utilization by making cores stalling. On the other hand write accesses are not blocking and have a less important impact on the utilization, even though write accesses make the waiting queue longer, which some how might delay other read accesses.

D. Performance Comparison

We use the multi-objective Pareto frontier to compare the performance of system configurations having different scheduling policies. We emphasize that we compare different configurations of the *same* system, where only scheduling policies vary from a configuration to another. Hence, our framework cannot be used to compare unrelated systems. To perform comparison, one can keep varying scheduling policies, while schedulability and functional requirements are satisfied, until better performance is achieved.

When comparing two configurations $X1$ and $X2$, if configuration $X1$ achieves smaller numbers for all metrics (core utilization, L2 and DRAM interference) compared to X_2 then X_1 is identified to be achieving better performance than X_2. However, if the performance values achieved by both configurations overlap, i.e. a configuration has a higher value in one metric and a lower value in another, we compare the differences obtained for individual metrics: *how large the difference between the utilization of $X1$ and $X2$ is compared to the difference between the interferences achieved by $X1$ and $X2$*. To illustrate the comparison, we use Fig. 9. After normalizing the metric values (using the highest obtained value as reference), we compare the difference between the utilization of $X1$ and $X2$ to the difference between the interference of $X1$ and $X2$.

Fig. 9. Performance comparison using the Pareto frontier.



The core utilization of configuration $X2$ is 60% less than that of $X1$ while the DRAM interference achieved by $X1$ is 40% less than that of $X2$. The performance gain achieved by $X2$ (60% for utilization) is greater than that achieved by $X1$ (40% for DRAM interference), which could qualify $X2$ to be better than $X1$ in terms of performance. In the general case, we need to consider the L2 interference as another dimension in the comparison. For more realistic comparisons, engineers can associate to each resource a utilization cost and a priority/criticality so that the comparison will rely on the utilization cost of each resource rather than the utilization time. Accordingly, the configuration leading to the cheapest cost (accumulated for all resources) will be the most suitable.

VII. Case Study

In this section, we consider two case studies: a Mission Control Computer avionic system (MCC) [13] and an Autonomous Vehicle Component (AVC) [18]. We analyze and compare the performance achieved by each case study when running processor-centric and memory centric scheduling. The diversity of the case studies gives a practical experience on what is the appropriate scheduling policy to be used for each application category to achieve higher performance.

As WCRAs are not provided in the original MCC benchmark, we calculate them by dividing the time spent by each task to fetch data over the average duration for one access. Moreover, as the numbers of access requests to shared cache and DRAM are not explicitly distinguishable in both case studies, we rely on the analysis results obtained by Ye *et al.* [33] where only 22.2% of the access requests hit the L2 cache.

A. MCC Example

MCC is a partial specification for a hypothetical avionic mission control computer system dedicated to combat and attack aircrafts. The application we consider is a composition of 15 tasks having the characteristics shown in Table I, where $WCRA_c$ and $WCRA_m$ are given in terms of (reads; writes), and all time values are in milliseconds. Tasks are grouped in 4 components running on four identical core [6]. Each component is statically mapped to one core.

The performance achieved by the system configurations where all cores run either FPS or memory-centric scheduling is depicted in Table II, in terms of μs for memory interference.

The performance achieved when running FPS scheduling is absolutely better than that of memory-centric scheduling. While having almost the same cores utilization, the memory interference delays obtained with memory-centric scheduling

TABLE I
Attributes of the MCC tasks

Task	Prd	Offset	WCET	$WCRA_c$	$WCRA_m$	Dln
T_1	10	0	1	(1; 0)	(3; 1)	5
T_2	40	0	2	(5; 1)	(20; 2)	40
T_3	40	10	4	(2; 2)	(7; 7)	40
T_4	40	20	2	(0; 0)	(1; 0)	40
T_5	40	30	1	(1; 0)	(3; 1)	40
T_6	55	0	8	(4; 13)	(14; 51)	55
T_7	52	16	6	(5; 2)	(20; 5)	52
T_8	52	32	8	(12; 1)	(42; 4)	52
T_9	80	20	6	(3; 1)	(13; 2)	80
T_{10}	100	0	7	(5; 1)	(18; 5)	100
T_{11}	100	50	3	(0; 0)	(0; 2)	100
T_{12}	200	0	1	(1; 2)	(3; 9)	200
T_{13}	200	100	2	(5; 0)	(15; 2)	200
T_{14}	400	0	6	(5; 1)	(18; 5)	400
T_{15}	1000	0	5	(1; 0)	(5; 2)	400

TABLE II
Performance results of the MCC system

Sched policy	Core	L2Req (most probable)	DramReq (most probable)	Utilization (%)
FPS	C0	4.0	15.3	10.1
	C1	5.0	15.25	12
	C2	5.26	15.25	11.8
	C3	6.05	15.0	34.9
MCentric	C0	5.1	15.19	10
	C1	6.2	15.25	11.5
	C2	6.05	16.3	11.8
	C3	6.1	15.33	35

are mostly larger than that of FPS, with a difference up to 27% for L2 and 7% for DRAM. The analysis results are easily distinguishable, without using Pareto frontier, and identify FPS as the more efficient scheduling policy for the MCC system.

B. AVC Example

AVC is a component of an autonomous vehicle system [18]. 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 autonomous vehicle systems.

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. Similarly to MCC system, the characteristics of the task set are shown in Table III where all time values are in milliseconds. We analyze such a task set with both cores running either EDF and memory-centric scheduling policies. The performance achieved by both configurations is depicted in Table IV, where all units are in μs. One can observe that there is no difference in terms of cores utilization between the AVC configurations running EDF and memory-centric scheduling. However, memory-centric scheduling achieves better performance regarding the delays to access L2 and

TABLE III
Attributes of the AVC tasks

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

TABLE IV
PERFORMANCE RESULTS OF THE AVC COMPONENT

Sched policy	Core	L2Req (most probable)	DramReq (most probable)	Utilization (%)
EDF	C0	1.94	4.16	45.1
	C1	2.01	4.12	44.8
MCentric	C0	1.90	2.99	45.1
	C1	1.93	4.08	44.9

DRAM than EDF's. The performance gain obtained when using memory-centric scheduling varies from 1.5% to 39%. This could be related to the fact that the AVC component is memory-intensive.

The longest analysis of the aforementioned case studies lasts for 730.77 seconds, whereas the used memory space is about 46.5MB. To study the scalability, we created a synthetic set of 30 tasks running on 8 cores, by just duplicating the MCC description. The longest analysis time, obtained when analyzing the utilization, is 1067.1 seconds whereas the largest used memory space is 223.7MB. The analysis time and used memory space are encouraging and suggest that our framework might scale to larger systems.

VIII. CONCLUSION

This paper presents a methodology for formal analysis of schedulability of multicore real-time systems. It supports making engineering decisions at design stage based on a platform model with a shared memory hierarchy, and combines it with alternative scheduling policies (processor-centric or memory centric). For each schedulable configuration, the framework automatically computes performance metrics related to different resources. Thus, trade-offs between alternative platforms for a given task characterization are quantifiable.

Among interesting directions for future work we envision more extensive scalability studies, variations in the memory intensiveness of applications, adding other resource dimensions (e.g. energy), and application to more realistic case studies including systems with networked multicore platforms.

REFERENCES

[1] JEDEC: DDR3 SDRAM Standard JESD79-3F. 2012.

[2] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with pag. In A. Mycroft, editor, *Static Analysis*, volume 983 of *LNCS*, pages 33–50. Springer Berlin Heidelberg, 1995.

[3] 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.

[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. *Sci. Comput. Program.*, 113:236–260, 2015.

[6] A. Boudjadar, J. Dingel, B. Madzar, and J. H. Kim. Compositional predictability analysis of mixed critical real time systems. In *FTSCS'15*, pages 69–84. Springer International Publishing, 2015.

[7] F. Cassez and K. G. Larsen. The impressive power of stopwatches. In *CONCUR'00*, volume 1877 of *LNCS*, pages 138–152, 2000.

[8] 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, 2012.

[9] R. Chhibber and R. Garg. Multicore processor, parallelism and their performance analysis. *IJARCST*, 2:31–37, 2014.

[10] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

[11] A. David, K. G. Larsen, A. Legay, M. Mikucionis, D. B. Poulsen, J. van Vliet, and Z. Wang. Statistical model checking for networks of priced timed automata. In *FORMATS*, volume 6919 of *LNCS*, pages 80–96. Springer, 2011.

[12] J. Diamond, M. Burtscher, J. D. McCalpin, B. D. Kim, S. W. Keckler, and J. C. Browne. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In *ISPASS'11*, pages 32–43, 2011.

[13] R. Dodd. Coloured Petri net modelling of a generic avionics missions computer. Technical report, Defence Science and Technology Organisation, Department of Defence. Australia, 2006.

[14] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Syst*, 17(2-3):131–181, 1999.

[15] P. Huyck. Arinc 653 and multi-core microprocessors; considerations and potential impacts. In *DASC'12*, pages 6B4–1–6B4–7, 2012.

[16] 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, 2014.

[17] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *ECRTS13*, pages 80–89, 2013.

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

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

[20] A. Löfwenmark and S. Nadjm-Tehrani. Experience report: Memory accesses for avionic applications and operating systems on a multi-core platform. In *ISSRE'15*, 2015.

[21] A. Löfwenmark and S. Nadjm-Tehrani. Understanding shared memory bank access interference in multi-core avionics. In *Proceedings of WCET'16*, OpenAccess Series in Informatics (OASIcs), 2016.

[22] 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, 2014.

[23] M. Paolieri, E. Quinones, F. J. Cazorla, J. Wolf, T. Ungerer, S. Uhrig, and Z. Petrov. A software-pipelined approach to multicore execution of timing predictable multi-threaded hard real-time tasks. In *ISORC'11*, pages 233–240, 2011.

[24] E. Putrycz, M. Woodside, and X. Wu. Performance techniques for COTS systems. *IEEE Software*, 22(4):36–44, 2005.

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

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

[27] A. Sarkar, F. Mueller, and H. Ramaprasad. Predictable task migration for locked caches in multi-core systems. In *LCTES '11*, volume 46, pages 131–140. ACM, 2011.

[28] 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, 2013.

[29] L. Tan. The worst-case execution time tool challenge 2006. *Intl Journal on Software Tools for Technology Transfer*, 11(2):133–152, 2009.

[30] G. Teodoro, T. M. Kurç, G. Andrade, J. Kong, R. Ferreira, and J. H. Saltz. Performance analysis and efficient execution on systems with multi-core CPUs, GPUs and MICs. *CoRR*, abs/1505.03819, 2015.

[31] A. Wilson and T. Preyssler. Incremental certification and integrated modular avionics. In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 1.E.3–1–1.E.3–8, 2008.

[32] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE Trans. Computers*, 65(9):2739–2751, 2016.

[33] 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.

[34] 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.

[35] 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.