Performance Driven Scheduling of Multicore Time-critical Systems

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

***Abstract*—Multicore platforms are getting widely used in the deployment of embedded systems due to their computing capacity. Memory interference represents a challenge in leveraging the performance of multicore platforms and may lead to drastic degradations of the platform performance, in particular for memory-intensive systems. This paper explores performance-driven schedulability of multicore systems by evaluating the performance when changing some design parameters (scheduling policies). The model-based framework we build enables to analyze the performance of multicore time-critical systems running core-centric and memory-centric scheduling policies. The system architecture we consider consists of a set of cores having each a local cache and sharing the cache level L2 and DRAM. The metrics we use to compare the performance achieved by different configurations of a system are: 1) cores utilization; 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 achieving better performance for a given system while maintaining the system schedulability. As a proof of concept, we analyze 2 different cases studies and compare the outcomes.**

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. Such multicore avionic systems are usually built by integrating different subsystems, potentially provided by different vendors, to enable incremental Design and Certification (iD&C) [34], recommended by the standard Integrated Modular Avionics (IMA) architecture [28]. However, due to the safety guarantees and hard critical requirements performance can be sacrificed, up to a certain degradation level, if the safety and reliability are in peril. Performance is then considered as a second class property that 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 with an efficient use of the hardware. 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 the access to the 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 [15].

Strong efforts have been devoted to the analysis of safety and predictability of multicore systems [18], [22], [3], [8], [23], however less studies focusing on the compensation of performance while maintaining the timing predictability guarantees have been achieved [33], [26], [29], [24], [31]. When designing a software system, there is a potential margin where a drift configuration, obtained by slightly tweaking the original system e.g. by assigning tasks differently to cores or using much optimal scheduling policies, can functionally behave in the same way while achieving better performance and satisfying the predictability requirements [30], [29], [23].

This paper represents a performance-driven schedulability framework for multicore systems. Such a model-based framework enables to analyze and improve the performance of multicore systems when changing cores scheduling policies (design parameters), while maintaining systems schedulable so that system configurations achieving better performance can be identified and planned for deployment. We adopt the Asymmetric Multi-Processing (AMP) [14] avionic scheduling architecture, where partitions are statically allocated to specific cores. Moreover, the processor scheduling alternatives we consider are memory-centric and core-centric. The platform architecture consists of a set of cores having each a local cache, and sharing the cache level L2 and DRAM. We use the cache coloring policy [16] to arbitrate concurrent access requests to the shared cache at level 2 (L2). In addition, we adopt the policy First Ready-First Come First Serve (FR-FCFS) [27], [15] commonly used by modern COTS-based memory controllers to schedule the DRAM access requests. The application model is given by a set of periodic task sets, each of which is assigned to a given core and scheduled using either memory-centric or core-centric policies. Moreover, application tasks have explicit read and write access numbers for shared caches and DRAM. We distinguish between read and write access requests to shared memories as read actions are blocking for cores, while write actions are not blocking

This work has been supported by the NFFP6 Project.

and can be performed using dedicated buffers.

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.

II. Related Work

[24] performance analysis techniques
[9] performance measurement of multicore systems
[33] performance analysis of multicore systems
[12] evaluation and optimization of the performance
[31] Providing performance predictability and improving memory interference

[30] The tool also provides the possibility of implementing different scheduling and mapping algorithms for multi-core process systems and evaluating the performance when changing various design parameters.

The analysis of multicore real-time systems has attracted a lot of attention recently.

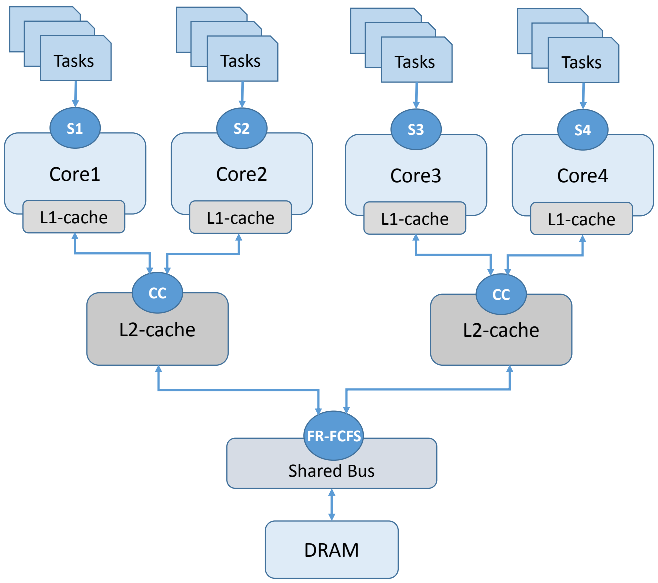
In this work, we combine the analysis processes so that for a given platform we check whether the application tasks will be schedulable or not, given the task WCETs estimated in isolation. Our framework provides also the ability to calculate the isWCET and a method for reallocation of tasks to cores upon non-schedulability. The novelty of our modeling approach is the inclusion of cache coloring policy [16] to arbitrate concurrent access requests to the shared cache, FR-FCFS policy [27], [21], [15] commonly used by modern memory controllers to schedule the DRAM accesses, while distinguishing between read and write requests.

III. Background

In multicore settings, assessement of the execution time and memory interference 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 [8]. 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. This section introduces benchmarking, as a technique to measure WCET and WCRA, as well as the fundamental background of our work.

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

Fig. 1. Simplified multicore system architecture.



A. Systems Benchmarking

Throughout this section we describe how the inputs required by our model-based framework are obtained form an actual system, in particular WCETs and WCRAs. In software engineering, benchmarking [25] aims at measuring a set of execution attributes and performance characteristics, e.g. WCET, of a given software system. The output of a system benchmarking (shortly known by benchmark) 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. Roughly speaking, flow analysis and profiling are the most commonly used techniques to measure the WCET and WCRA of application tasks.

Flow analysis [32], [8] 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. 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). WCET is then the time spent when executing the longest path of the CFG. Static analysis has been practiced for the analysis of software systems via different analysis tools, e.g. SWEET [32].

System profiling [39], [15] 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 technology 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 to the process being run. The number of accesses for each core can be obtained using Performance Monitor Counters (PMCs) [39], equipping certain multicore platforms.

In literature [15], [39], profiling and static analysis techniques are not completely differentiated. Both techniques can be used to measure WCET and WCRA.

B. Core-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 different tasks execution 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 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 the ready queue can be assigned to the priority, remaining execution time, etc.

In our setting, we adopt FIFO, FPS and EDF as scheduling policies for the individual cores.

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 [36], [35]. Tasks assigned to the same core are sorted in the ready queue according to a decreasing order of their WCRA (numbers of memory accesses).

We distinguish between the access numbers to L2 and DRAM, 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 number where task having higher number is scheduled first.

D. Shared Memory Access Scheduling

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

Cache coloring policy [16], [37] is an algorithm to control the access to the shared cache level 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.

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 [27], [15]. 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.

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. Hence, estimating the optimal cache size for each application and cache recoloring are beyond the scope of this paper.

E. Statistical Model Checking

We are using UPPAAL SMC [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. We can summarize the main features of UPPAAL SMC in the following:

- 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.
- 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`.
- 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`. `P` is specified using either LTL or tMITL logic.

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 from a standard cylinder construction according to

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

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 states 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 stage 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 separation between events e_{i-1} and e_i.

Moreover, for a given run π we introduce the following the obvious predicates:

- $\mathit{InUse}^s_\pi(C) \in \mathbb{N}$ states whether the core C is in use in state s of π or not. In fact, a core is in use if it is either effectively executing a workload or stalling due read access requests to L2 and DRAM.
- $\mathit{Issued}^C_\pi(\mathit{req})$ indicates at which state of π the access request *req* performed by core C has been issued.
- $\mathit{Granted}^C_\pi(\mathit{req})$ 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) L is given by the following:

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

The average utilization of a core C over the set of runs *Runs*, analyzed for the same time interval length L, will be the sum of the individual runs utilization on the number of runs:

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

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

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

$$\mathit{ReqDelay}^C_\pi = \max(\mathit{ReqDelay}^C_\pi(\mathit{req} \mid \mathit{req} \in \mathit{Req}_C))$$

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

$$\mathit{ReqDelay}^C_{\mathit{Runs}} = \frac{\sum\limits_{j=1}^{j=|\mathit{Runs}|} \mathit{ReqDelay}^C_{\pi_j}}{|\ \mathit{Runs}\ |}$$

As access requests to L2 and to DRAM do not have the same impact on the interference, later on we distinguish between $\mathit{L2\_ReqDelay}^C_{\mathit{Runs}}$ and $\mathit{DRAM\_ReqDelay}^C_{\mathit{Runs}}$ 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 tasks mapped to different cores. 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 actions. Access requests to shared memory are non-deterministically spread out along the task execution. The reason behind that is that tasks 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 assumptions about the systems we consider are the following:

- Tasks are periodic and non-preemptible during CPU use.
- 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.

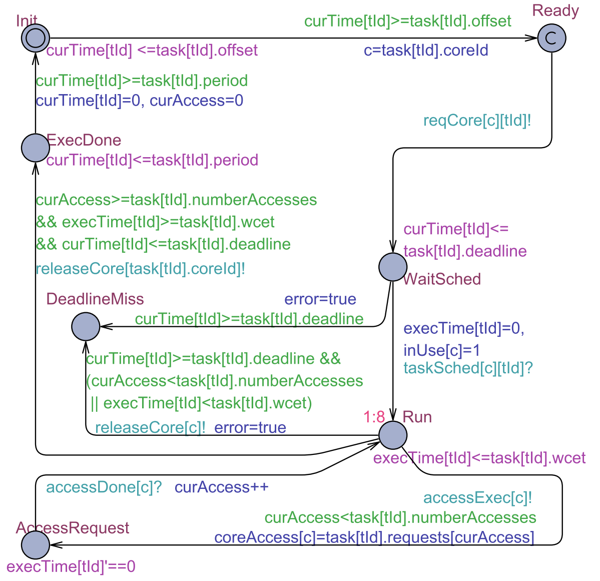
A. Application Model

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 behavior is abstracted at task level using WCET and WCRA for both shared cache and DRAM. WCET is the pure execution time, calculated 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 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. $\mathit{WCRA}^r_c, \mathit{WCRA}^w_c, \mathit{WCRA}^r_m$ and WCRA^w_m. 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. Such patterns are identifiable using abstract interpretation [20] and program/cache analysis [13], e.g. the PAG tool [2], on a given platform architecture.

An access request to a shared memory is given by a $\mathit{pattern} \in \{L2, DRAM\}$ 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 *issueTime*. In fact, *issueTime* is initially empty and will be initialized by a core when the access request is triggered. Accordingly, an access request is formally given by $\mathit{req} = \langle \mathit{pattern}, RW, \mathit{issueTime} \rangle$. WCRA^r_c and WCRA^w_c, respectively WCRA^r_m and WCRA^w_m, of a given task are then the numbers of read and write accesses to L2, respectively DRAM.

Definition 3 (Task structure): A task T is given by $\langle \mathit{Prd}, \mathit{Offset}, \mathit{WCET}, \mathit{WCRA}^r_c, \mathit{WCRA}^w_c, \mathit{WCRA}^r_m, \mathit{WCRA}^w_m, \mathit{Dln}, \mathit{Pri} \rangle$ where *Prd* is the task period, *Offset* is the periodic offset, *WCET* is the pure execution time, *Dln* is the relative

Fig. 2. Task template model

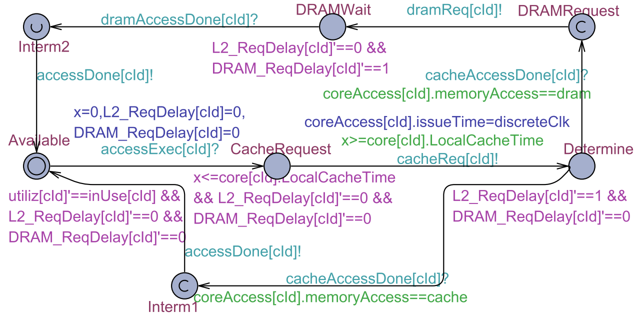

Init
curTime[tId]>=task[tId].offset
Ready
curTime[tId] <=task[tId].offset
c=task[tId].coreId
curTime[tId]>=task[tId].period
curTime[tId]=0, curAccess=0
reqCore[c][tId]!
ExecDone
curTime[tId]<=task[tId].period
curAccess>=task[tId].numberAccesses
&& execTime[tId]>=task[tId].wcet
&& curTime[tId]<=task[tId].deadline
releaseCore[task[tId].coreId]!
curTime[tId]<=
task[tId].deadline
WaitSched
DeadlineMiss
error=true
curTime[tId]>=task[tId].deadline
execTime[tId]=0,
inUse[c]=1
taskSched[c][tId]?
curTime[tId]>=task[tId].deadline &&
(curAccess<task[tId].numberAccesses
|| execTime[tId]<task[tId].wcet)
releaseCore[c]! error=true
1:8
Run
execTime[tId]<=task[tId].wcet
accessDone[c]? curAccess++
accessExec[c]!
curAccess<task[tId].numberAccesses
AccessRequest
coreAccess[c]=task[tId].requests[curAccess]
execTime[tId]'==0

deadline whereas *Pri* is the priority level associated to T. $WCRA_c^r$, $WCRA_c^w$, $WCRA_m^r$ and $WCRA_m^w$ are described above.

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.

Our Uppaal task template model is depicted in Fig. 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 (c) it is mapped to, 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[c]=1*, and moves to location Run to execute. The status update of variable *inUse[c]* leads the clock accumulating the core utilization to resume. 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 location 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 task template can be instantiated for different tasks by just providing the aforementioned parameters.

Fig. 3. Core template model



B. Platform Model

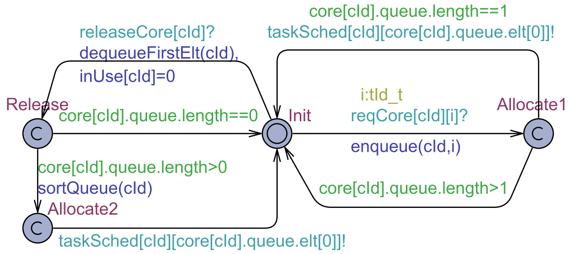
A platform is given by a set of processing elements PE, sharing a cache level L2 and DRAM memory, and schedulers to manage the access to L2 and DRAM. Each processing element PE is given by a computation resource (core), a local cache memory and a scheduler to dispatch tasks to run on that core. The access time for local caches may vary from one PE to another. In fact, such a number corresponds to a cache hit. In case of a cache miss, extra delay will result from accessing the shared memory.

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 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 *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 preempted (*L2_ReqDelay[cId]'==0*) and 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, 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 preempted 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

Fig. 4. Scheduler template model



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 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).

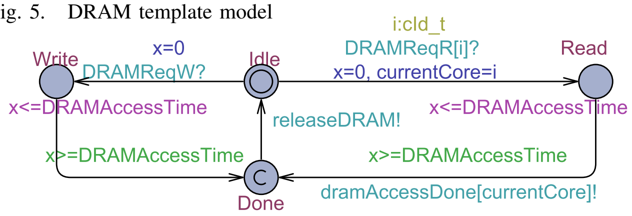
Fig. 4 depicts the core scheduler model. Initially at location Init waiting for a ready task request, the core moves to location Allocate1 while queuing the identifier of the requesting task. If the core queue contains only one element (*queue.length==1*), which is the identifier of the newly added task, 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 that is shared between all cores, but it can easily be extended for several banks by just duplicating the DRAM structure and assigns each to one core only [38]. 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 such latency, opening and closing durations of banks.

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 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, so that other requests can be performed.

Fig. 5. DRAM template model



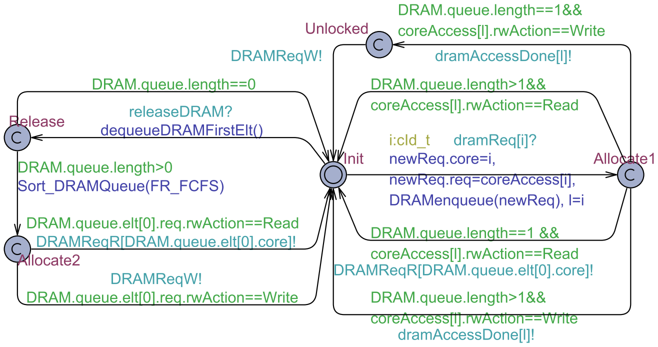
We adopt the FR-FCFS policy to arbitrate accesses to DRAM. We assume that row opening and reload actions are instantaneous, so that we do not need to consider any preference based on the *already open row* policy [15]. This leads us to consider the attribute *issueTime* of each request as a *readiness*. 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[l]!*) as location is committed Allocate1. Moreover, 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. In all of the four scenarios, the scheduler moves back to location Init.

Once an access request is performed, the scheduler is notified by the DRAM through a synchronization event *re-*

leaseDRAM? 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 first element of the queue *queue.elt[0]* using the appropriate channel (*DRAMReqR* or *DRAMReqW*) according to the request nature; 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 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 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 affect 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

This section gives a technical description of the performance and schedulability analysis, as well as a discussion about how the performance analysis outcomes are interpreted and compared.

A. Schedulability Analysis

In our framework, the system schedulability is analyzed as a reachability property using symbolic model checking [5]. 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 query supported by Uppaal:

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

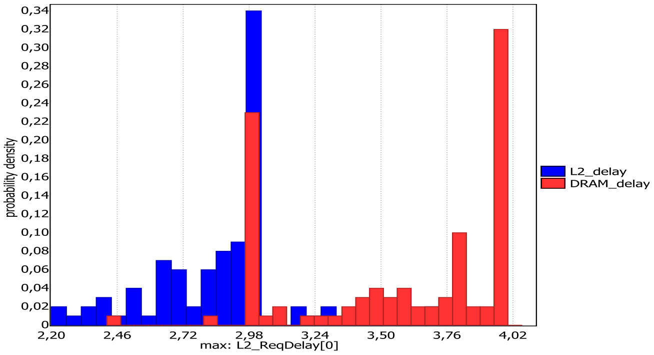
B. Memory Interference Analysis

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 average delays of a core C, to access L2 and DRAM respectively, in terms of probability distributions we use the following SMC queries:

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

Fig. 7 shows the probability distributions of the request delays to L2 and DRAM of a core C where $X = 10^3$ and $Y = 10^4$. C runs in parallel with another core, each of which serves 2 tasks. 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.

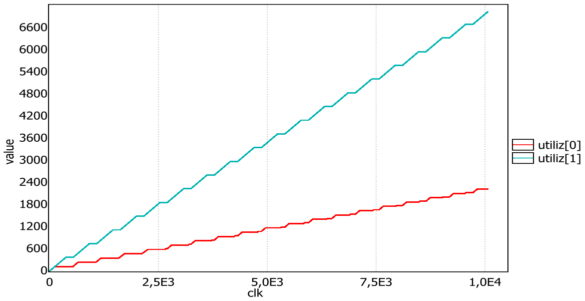


C. Utilization Analysis

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:

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

Fig. 8. Simulation of cores utilization.



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.

VII. Case Study

To illustrate the applicability of our performance-driven scheduling framework, we consider two case studies: a Mission Control Computer avionic system MCC [19] and an Autonomous Vehicle Component AVC [17]. MCC is CPU-intensive whereas SVC is memory-intensive. We analyze and compare the performance achieved by each case study when running core-centric and memory centric. 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 high performance.

The time durations for effective accesses to L2 ($L2AccessTime$) and DRAM ($DRAMAccessTime$) are $1/10^3$ and $1/10^2$ milliseconds respectively. Moreover, all time units are given in milliseconds.

A. MCC Example

MCC is a partial specification for a hypothetical avionics mission control computer system dedicated to combat and attack aircrafts. The application we consider is a composition of 13 tasks having the characteristics shown in Table I, where *WCRA*$_c$ and *WCRA*$_m$ are given in terms of (reads; writes). Tasks are grouped in 4 components $\{$*Display*=$(T_1, T_2, T_3, T_4, T_5)$, *Radar*=$(T_6, T_7)$, *NAV*=$(T_8, T_9, T_{10})$, *Weapon*=$(T_{11}, T_{12}, T_{13})\}$ running each on one core [6].

****provide wcra for each task***

***Analysis results

B. AVC Example

AVC is a component of an autonomous vehicle system [17]. 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. The characteristics of the task set are shown in Table II.

TABLE I
Attributes of the MCC tasks

Task	Prd	Offset	WCET	WCRA$_c$	WCRA$_m$	Dln
T_1	200	0	3	(X; X)	(Y; Y)	200
T_2	200	0	1	(X; X)	(Y; Y)	200
T_3	80	0	2	(X; X)	(Y; Y)	80
T_4	80	0	9	(X; X)	(Y; Y)	80
T_5	200	0	1	(X; X)	(Y; Y)	200
T_6	50	0	5	(X; X)	(Y; Y)	50
T_7	25	0	2	(X; X)	(Y; Y)	25
T_8	59	0	8	(X; X)	(Y; Y)	59
T_9	200	0	3	(X; X)	(Y; Y)	200
T_{10}	1000	0	1	(X; X)	(Y; Y)	1000
T_{11}	200	0	1	(X; X)	(Y; Y)	200
T_{12}	200	0	3	(X; X)	(Y; Y)	200
T_{13}	50	0	3	(X; X)	(Y; Y)	50
T_{14}	200	0	3	(X; X)	(Y; Y)	200
T_{15}	50	0	3	(X; X)	(Y; Y)	50

TABLE II
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

***Analysis results

Both cores run EDF schedulers (core-centric), while the access time to the local cache lasts for $1/10^3$ millisecond.

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 Fig. **??**. 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.

Add analysis of average delay to access L2 and DRAM *****

Second alternative: when using a memory-centric scheduling policy****

case study2*

****Discussion of the performance results and how one would choose the appropriate scheduling policy.

VIII. Conclusion

***Summary of the paper

As future work, we plan to compare the performance results obtained using our framework with the state of the art tools [26], [24]. We intend also to incorporate detailed architectures of L2 and DRAM, consider other factors in the derivation of configurations such as task-to-core mapping where we reallocate tasks to cores differently and the use of other performance metrics such as energy consumption.

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 C. Artho and C. P. Ölveczky, editors, *FTSCS'15*, pages 69–84, Cham, 2015. Springer International Publishing.

[7] F. Cassez and K. G. Larsen. The impressive power of stopwatches. In C. Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 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 U. Fahrenberg and S. Tripakis, editors, *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 *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 32–43, 2011.

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

[14] P. Huyck. Arinc 653 and multi-core microprocessors x2014; considerations and potential impacts. In *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pages 6B4–1–6B4–7, 2012.

[15] 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.

[16] 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.

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

[18] B. Lisper. SWEET - A tool for WCET flow analysis (extended abstract). In *Leveraging Applications of Formal Methods, Verification and Validation. 6th International Symposium, ISoLA'14*, volume 8803 of *Lecture Notes in Computer Science*, pages 482–485. Springer, 2014.

[19] C. D. Locke, D. R. Vogel, and T. J. Mesler. Building a predictable avionics platform in ada: a case study. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 181–189, Dec 1991.

[20] 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, 2010.

[21] 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.

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

[26] F. F. Rivera, R. Iglesias, J. A. Lorenzo, J. C. Pichel, T. F. Pena, and J. C. Cabaleiro. A graphical tool for performance analysis of multicore systems based on the roofline model. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 847–848, 2012.

[27] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM.

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

[29] A. Sarkar, F. Mueller, and H. Ramaprasad. Predictable task migration for locked caches in multi-core systems. In *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, volume 46 of *LCTES '11*, pages 131–140. ACM, 2011.

[30] E. M. Shakshuki, M. Sharma, H. Elmiligi, and F. Gebali. SMARTs: A tool to simulate and analyze the performance of real-time multi-core systems. *Procedia Computer Science*, 34:544 – 551, 2014.

[31] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 639–650, 2013.

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

[33] 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.

[34] 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.

[35] 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.

[36] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE TRANSACTIONS ON COMPUTERS*, 2015.

[37] 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.

[38] 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.

[39] 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.