2015 IEEE 18th International Symposium on Real-Time Distributed Computing

Seeking Time-Composable Partitions of Tasks for COTS Multicore Processors

Gabriel Fernandez*†, Jaume Abella†, Eduardo Quiñones†,
Luca Fossati§, Marco Zulianello§, Tullio Vardanega¶, Francisco J. Cazorla†‡
* Universitat Politècnica de Catalunya, Spain
†Barcelona Supercomputing Center, Spain
‡Spanish National Research Council (IIIA-CSIC), Spain
§European Space Agency, The Netherlands
¶University of Padova, Italy

***Abstract*—The timing verification of real-time singlecore systems involves a timing analysis step that yields an Execution Time Bound (ETB) for each task, followed by a schedulability analysis step, where the scheduling attributes of the individual tasks, including the ETB, are studied from the system level perspective. The transition between those two steps involves accounting for the interference effects that arise when tasks contend for access to shared resource. The advent of multicore processors challenges the viability of this two-step approach because several complex contention effects at the processor level arise that cause tasks to be unable to make progress while actually holding the CPU, which are very difficult to tightly capture by simply inflating the tasks' ETB. In this paper we show how contention on access to hardware shared resources creates a circular dependence between the determination of tasks' ETB and their scheduling at run time. To help loosen this knot we present an approach that acknowledges different flavors of time composability, examining in detail the variant intended for partitioned scheduling, which we evaluate on two real processor boards used in the space domain.**

I. Introduction

Research on timing analysis for multicore processors is still in its infancy. This is particularly the case for COTS hardware, which to date, to the best of our knowledge, has been addressed with some success only for well-behaved – hence not especially realistic – variants. The difficulty with the timing analysis of software programs running on multicore processors thus is a serious impediment to their adoption in real-time systems industry. As asserting a single, tight, safe and absolute worst-case execution-time (WCET) bound for COTS multicore processors is still an open problem, which far exceeds the state of the art in timing analysis, in this paper we use the term Execution Time Boud, or *ETB*, in place of the more common WCET.

Transposing to multicore processors the practice in place for traditional singlecore processors, we note that the timing analysis of a real-time system involves essentially two steps. The first step derives for each application program an ETB, computed assuming that the program runs in isolation. In the second step, all system-level overheads such as interrupts, blocking times when arbitrating access to shared resources in the program, and the preemptions that result from dynamic scheduling are taken into account, compositionally (that is to say, by capturing holistically all interference effects that result from composing software programs into the final system). Response-time analysis [19], [7] is one of the techniques typically used in the latter step, which is fed with the ETB of individual tasks, which is assumed given. In that two-step process, the ETB computed considering tasks in isolation is augmented with the time duration in which the task, though notionally holding the CPU, cannot actually progress. This timing approach in which ETB is inflated to account for system-level effects has been shown to work sufficiently well in singlecore architectures.

Task scheduling is one of the system functions that affects the mosts tasks' timing behavior. This is so because task scheduling determines when and how tasks may pre-emptively interleave, hence the extent of pre-emption effects that each task may suffer. In COTS multicore systems, task scheduling affects task pre-emption much like in singlecore systems. Unlike singlecore systems, however, *scheduling in multicore systems also determines the tasks that may potentially run at the same time in the processor and hence, can conflict in the access to hardware shared resources. The latter factor, which has a massive impact on the tasks' timing behavior, creates a nasty circular dependence between task scheduling and ETB determination in the timing analysis for COTS multicores.*

Attempting to compositionally extend in response-time analysis the magnitude of the ETB obtained for individual programs in isolation, with the effects of inter-task conflicts on access to processor resources is impractical in the general case. This is so because tasks may conflict so frequently (e.g. on every access to the on-chip bus) and for so many resources (caches, buses, queuing buffers, etc.) that the amount of knowledge to characterize the application usage of hardware resources required rapidly gets out of control. This calls for an alternative solution, in which on-chip contention effects are accounted for as part of the ETB determination.

The much-sought property of time composability [28], [15], when transposed to a multicore processor, stipulates that the timing behavior of an individual task is not affected by the activity of its co-runners. Time composability is generally considered as an all-or-nothing metric: if the ETB of a task varies depending on the tasks' actual co-runners, then that ETB is non-time-composable. This full extent of time composability (in force of which we have *fully time composable*, *fTC*, ETB) can be obtained by forcing the timing analysis tool to contemplate a-priori *all* sources of worst-case contention that a task may possibly experience on access to hardware shared resources. As we discuss in this paper, however, this provision may be overly pessimistic and defeat the purpose of transitioning the system from singlecore to a multicore processor architecture.

This paper aims to break the circular dependence between the timing analysis and scheduling while ensuring effective containment of pessimism. To that end, this paper makes a threefold contribution, as follows:

1) We show that the traditional *fTC*-centric approach is acceptable only if the interference effects of contention for processor resource sharing are low, which typically is the case when the core count is low (≤ 4) and the hardware resources being shared cause modest contention impacts. We present realistic scenarios where these conditions do not hold and the provisions of *fTC* ETB are exceedingly pessimistic.

2) We consider the flavors of time composability first introduced in [12]. We develop the notion of sufficient time composability, and of *partially time-composable (pTC) ETB*, which proceeds from it and solely considers the *contention conditions* that the task of interest can effectively incur at run time as a result of actual scheduling decisions. In that respect, *pTC* trades tightness in the resulting upper bounds for ease of determination. In particular we introduce two variants of *pTC* , called allocation-aware time-composability (*aTC*) and task set-aware time-composability (*tsTC*), respectively devised for partitioned and global scheduling.

3) We develop an aTC-aware allocation algorithm, called aTC-allocator, which performs timing analysis in an iterative fashion, integrated with reasoning on task scheduling. For an m-core processor, aTC-allocator starts by assuming that any given set of m tasks in the task set of interest can simultaneously run. aTC-allocator considers all the contention conditions that may occur among those m co-runners. As the allocation progresses, at every step of task-to-core assignment, aTC-allocator discards the contention conditions generated by the tasks that have been assigned to the same core, since they no longer can be co-runners of each other. This procedure progressively reduces the inflation effects in the ETB computed for each individual task, and culminates with finding a feasible assignment, if one exists. We compare *aTC* and *fTC* on two real COTS processors used in the space domain, a dual-core LEON3-based GR712RC board, and a quad-core LEON4 ML510 board. Our results show that for the GR712RC *fTC* ETB incur acceptable pessimism, which makes *fTC* ETB usable on that processor. Conversely, we show that, while the *fTC* ETB on the ML510 are too pessimistically inflated, *aTC* considerably reduces the pessimism by tuning the ETB of individual tasks to the contention conditions determined by the chosen task-to-core assignment.

The remainder of this paper is organized as follows: Section II discusses some related work. Section III presents our approaches based on *pTC* to break the circular dependence between timing analysis and task scheduling in COTS multicores. We present our aTC-aware allocator in Section IV. Section V presents the difficulties in deriving ETB under each *pTC* approach. Section VI presents results for two real COTS multicore processors: the GR712RC and the NGMP; Finally, Section VII summarizes the main conclusions of this study.

II. Related Work

Contention on access to hardware shared resources at the processor level is a much studied topic in the state of the art. [10] provides a taxonomic summary of relevant works.

Contention for off-chip resources such as the bus is addressed predominantly with TDMA buses [32], for which the worst case of interest for timing analysis is the worst possible alignment of task requests to their TDMA slots. The works that assume dynamic arbiters instead (cf. e.g., [31]) consider the particular pattern of accesses that each contending task may make to the bus. Notably however, several of the latter type of works [33], [32] make assumptions that prevent their use in the two COTS processors considered in this paper. In particular, they model just one off-chip shared resource that can process one request at a time only and in which requests are synchronous (so that the contending task is stalled) and cannot be split into several asynchronous requests. They further assume that on-chip shared resources (e.g. core-to-cache bus, caches, etc.) are replicated per core or partitioned across tasks, so that tasks incur no contention effects when accessing on-chip resources. These works study specific task models in which programs can be divided into superblocks for which both maximum and minimum access bounds and ETB can be derived.

Hardware support has been proposed to either eliminate or control contention on access to hardware resources, e.g.: TDMA or UBD in buses [23]; partitioning for caches [23]; real-time aware controllers for memory [3], [25]. Those solutions simplify timing analysis as contention effects are nullified when the hardware design eliminates contention interference by construction or easily determined when hardware features bound contention interference [22]. Solutions of this kind ultimately enable the use of the two-step analysis process inherited from the singlecore practice. However, to the best of our knowledge, no current commercial multicore processor provides complete isolation from contention interference or full control of it. Although some multicore processors do implement cache partitioning [2], [5], the above techniques can only be sensibly used if *all* hardware shared resources (which, in the case of the NGMP, means on-chip interconnection, shared cache, and core to memory bus) are suitably controlled. It is worth noting in this regard that several works [26], [25] show how, in real processors, contention on access to memory alone may more than double the execution time of tasks considered in isolation. It therefore follows that controlling contention interference from cache effects alone in real-world multicore processors is most evidently not enough to attain isolation in the time domain among tasks. The problem is that the execution time of a software program running on COTS multicore processors may be inordinately affected by the contention effects that the co-runners of that task may cause on the hardware shared resources.

One solution to this predicament might be to use an analysis approach that, for all the tasks that may run in parallel, studies statically, at a very fine grain of detail on an abstract model of the processor, the accesses that they may make to hardware shared resources and how they might contend with one another [18], [16]. This technique may lead to determine ETB that are considerably tighter than what we can arrive at, but at the cost of a much more onerous and complex effort, which trades time composability for tightness, since its results can only apply to a given task configuration that cannot be varied without invalidating the analysis results.

Finally, the authors of [9] propose design principles to make multicore processors more predictable in the time dimension. For as interesting as it may be, however, this approach cannot be applied to existing COTS technology and therefore does not have the same goals as we do.

III. Breaking the Timing Analysis - Scheduling Dependence

Time composability is generally regarded as all-or-nothing binary metric. When applied to the ETB for a task, therefore, any variability in the computed bound that results from the determination of the actual co-runners of the task of interest, is deemed a fatal disruption of time composability. The implication is that the ETB for a task can be said to be *fTC* if it accounts for the worst-case *contention conditions* that can be generated by any other potential co-runner of that task in the system. As we show in this work, although using *fTC* ETB does indeed break the circular dependence between timing analysis and scheduling, in many cases the resulting values are too pessimistic to be of any practical use.

A. Generalizing Time Composability

Our proposal to attack this dependence consists in determining *Partially Time Composable* (*pTC*) ETB for each task. The idea behind this approach is to reduce the search space for the contention conditions to consider in the computation of *pTC* ETB. The resulting set of *pTC* ETB remains valid as long as the contention conditions for which they have been derived are maintained throughout system operation. The contention conditions for a task are meant to cover all hardware factors of influence on its timing behavior. In the case of multicore processors, the contention conditions for task τ_i are determined by the load that the task's co-runners may place on the hardware shared resources, which ultimately determines the extent of hardware contention that τ_i may incur at run time.

Under our *pTC* approach, each task τ_i is associated multiple ETB, one for each of the possible set of contention conditions that need to be considered. For that approach to be usable, the scheduling of tasks at run time must ensure that the contention conditions that every task encounters at run time stay within the boundary of those that were considered when the corresponding ETB was derived, so that the results of schedulability analysis can be safely asserted.

We assume a task set $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$. For partitioned scheduling, which is the main focus of this paper, individual tasks are statically assigned to one of m groups[1], $\Phi = (\varphi_1, \varphi_2, \cdots, \varphi_m)$, where m stands for the number of cores in the processor. In presenting our approach we use the following terms, which are specific to partitioned scheduling.

• We call *group mates (gm_i)* of a given task τ_i, those tasks allocated to the same core that τ_i is assigned to, called φ_i. That is, $gm_i = \{\tau_j\}, \tau_j \in \varphi_i \wedge \tau_j \neq \tau_i$.

• Similarly, we call *siblings (sb_i)* of a given task τ_i, those tasks allocated to a different group from the one to which τ_i is assigned. That is, $sb_i = \{\tau_k\}, \tau_k \notin \varphi_i$. It follows that the union of a task, its group-mates and its siblings forms the whole task set $\mathcal{T} = \{\tau_i \cup gm_i \cup sb_i\}, \forall \tau_i \in \mathcal{T}$.

We also use other terms that equally apply to partitioned and global scheduling:

• We call *co-runner tasks (cr_i)* of a given task τ_i at a given instant t in which task τ_i is being executed, the set of $m-1$ tasks running in parallel with τ_i at t.

• We call *workload* at time t, the set of tasks that simultaneously run at t. Hence, the co-runners of a task τ_i at t, together with τ_i itself form the *workload* at time t. With partitioned scheduling the group mates and siblings of any task τ_i are determined by the task-to-core assignment algorithm, while τ_i's co-runners are determined by the scheduling algorithm used on individual cores.

• The *potential co-runners* of τ_i are all the tasks that may run in parallel with τ_i at any point in time during system execution. With partitioned scheduling, the potential co-runners of τ_i's are only its siblings sb_i. With global scheduling the potential co-runners of τ_i are all the tasks in the task set since any subset of $m-1$ of them can run simultaneously at a given point in time with τ_i.

In this work we consider four flavors of time composability with decreasing resilience of the computed ETB to variations in the contention conditions captured by the analysis. We introduce them next and develop them in the subsequent sections.

[1]We use the term group, allocation and core indistinctly to refer to the partition of interest.

• *Full TC (fTC) ETB*. *fTC* considers the worst possible contention effects that a task may suffer, in the general case (thus owing to the nature of the hardware, not contingent on the particular activity of the tasks in the task set), when attempting to access hardware shared resources. This definition, which may sound inflationary, serves the purpose of allowing *fTC* ETB values to be independent of the nature of the task's co-runners as well as of their scheduling at run time.

• *Task-set TC (tsTC) ETB* apply to global scheduling when no task-to-core assignment is predetermined. With *tsTC*, the ETB estimate C_i derived for τ_i has to be time-composable with respect to the contention conditions that can occur at run time for any potential workload of tasks in the task set.

• *Allocation-based TC (aTC) ETB*. This formulation, which applies to partitioned scheduling when some task-to-core assignment Φ is given, has the ETB estimate C_i, derived for τ_i to be time-composable for the contention conditions that can occur for Φ, is guaranteed to be higher than execution time that τ_i's may incur under any scheduling of its siblings in Φ. That is to say, C_i covers all the contention conditions that τ_i's siblings may generate.

• *Non time composable (nTC) ETB*. This is the weakest form of time composability, and we report it here only for the sake of taxonomic completeness, without discussing it further in this paper. An nTC ETB computed for task τ_i, in fact, is determined for the specific contention conditions that can be generated from a given set of co-runners with given characteristics (for execution time, access to hardware shared resources, and competing alignments at run time), and it is only valid for that particular case. This variant is too feeble to be used in practice because any, however minor, variation in the assumed characteristics varies the contention conditions for which the *ETB* was determined and invalidates it.

On the basis of the above notions, we can now study the degree of pessimism incurred by each TC approach of interest, and how that affects incremental verification.

Fully Time-Composable ETB. The use of *fTC* ETB in the development of multicore real-time systems presents obvious benefits since the bounds computed in that manner for a task always upper bound the contention effects that the task's co-runners may cause on hardware shared resources present on the target processor. The primary benefit is that individual subsystems can be independently developed and incrementally integrated and qualified without risks of timing-related regression at system level. Furthermore, the use of *fTC* ETB *breaks the circular dependence between timing analysis and scheduling caused by the contention effects arising upon system integration* and therefore allows using the Execution Time Analysis and the Response Time Analysis techniques much like done for singlecore systems.

Section V discusses how to determine the *fTC* ETB for a given task τ_i for the COTS processors considered this work.

The dark side of *fTC* ETB is that the upper-bound values computed with it may be *overly pessimistic*. The *fTC* ETB we obtained for some EEMBC benchmarks [27] on the NGMP processor (cf. Section VI for details) are up to 5.8x higher than the ETB obtained in configurations where no contention arises. The main corollary of this result is that *the price in overestimation paid to enable incremental verification* – which is the common way to approximate time composability – *may defy the whole point of using multicores in real-time systems*.

Task-set Time Composable ETB. Under task-set time composability the contention conditions of interest for each task are narrowed down to those that can actually occur considering the execution of the specific task set and its specific scheduling regime. Task-set time composability is designed for use with global scheduling in that when partitioned scheduling is used, further reductions in pessimism can easily be obtained by canceling contention effects ruled out by the task-to-core assignment decisions.

Once *tsTC* ETB are derived for each task, the scheduler has full freedom to schedule tasks on any core, similarly to the case of *fTC* ETB. However, unlike for *fTC*, any change in the workload invalidates all results and requires a full repeat of the timing analysis stage for all tasks in the task set.

The challenge is to determine, for every individual task, an ETB that upper bounds its execution time for any scheduling scenario across all cores. In this regard, in the general case, for any task τ_i in the task set $\mathcal{T}$, all tasks $\tau_{j_{\{j \neq i\}}}$ are siblings of τ_i and its potential co-runners. To attain *tsTC* the contention conditions of interest must therefore cover all possible scheduling scenarios that may occur for $\mathcal{T}$ across all cores, with *tsTC* ETB for τ_i capturing the resulting worst-case contention situation. Section V discusses how to compute *tsTC* ETB.

Allocation-based Time Composable ETB. This variant of time composability is designed for use with partitioned scheduling. Below we identify two partition scheduling scenarios in which *aTC* may be applied.

Task-to-core assignment is given. In this case, which transposes to multicore processors the practice in use with singlecore processor systems in several application domains, where the system integrator assigns individual scheduling partitions to individual subsystem suppliers. For the purposes of this paper and equally applicable to singlecore and multicore processors, a scheduling partition corresponds to a slice of CPU time, regardless of whether statically or dynamically assigned. The supplier develops the contracted software, normally organized in a schedule of tasks, to fit in the assigned partition. As in the multicore processor scenario the partition is pre-allocated to a core, the task-to-core assignment is given. From the timing analysis perspective, the problem reduces to determining an ETB for each task in each partition, knowing their respective assignment.

Task-to-core allocation is not given. In this case the problem is to determine how to assign tasks to cores and then derive a ETB for each task such that the final allocation is feasible.

TABLE I. TAXONOMY OF TIME COMPOSABILITY APPROACHES. LEGEND: +++ (HIGH), ++ (MODERATE), + (LOW) AND − VERY LOW.

	global sched.	partit. sched.	overestimation	increment. verific.	Execution Conditions
fTC	✓	✓	+++	+++	All possible tasks
tsTC	✓	✓	++	++	Tasks in the task set
aTC		✓	+	+	Sibling tasks
nTC			-	none	Particular workload

We need an approach that associates several ETB to each task τ_i, one for every possible task-to-core assignment, such that the task's *aTC* ETB$_{ai}$ is time composable for a particular allocation a_i, of the task set. In Section IV we develop an allocation algorithm that is aware of time composability.

B. Putting it all together

Table I presents the time composability approaches discussed in this work. *fTC* can be used with global and partitioned scheduling. It cleanly breaks the dependence among timing analysis and scheduling. Hence, at the cost of some pessimism in the computation of ETB, it allows good control of feasibility in the face of incremental system integration. *tsTC* works with global scheduling but is vulnerable to changes in the behavior and characteristics of the tasks in the system. This drawback makes this approach less apt for incremental development. *aTC* is fit for partitioned scheduling and, in general, arrives at tighter ETB than *tsTC*, although only for the contention conditions determined for the considered task-to-core assignment and for the assumed per-core scheduling.

It is worth noting that the contention conditions captured for *fTC* are more conservative than those considered in *tsTC*, which in turn, are more pessimistic than those accounted for *aTC*. Owing to their conservatism, the ETBs derived for *fTC* and for *tsTC* can in principle be used with any scheduling regime. In that manner, theoretically greater schedulable utilization may possibly be sought (as offered for example by certain global scheduling algorithms [30], [8]) to compensate for the increase in pessimism in the assumed ETB load.

The particular time composability approach to use ultimately depends on the development needs and constraints.

IV. aTC-AWARE ALLOCATION ALGORITHM

From the taxonomy of Time Composability approaches presented in the previous section, we focus on *aTC* for the more general case in which the task-to-core allocation is not given. To support *aTC* we have developed an allocation algorithm, called aTC-allocator, that is aware of *aTC* ETB. In particular, the goal of an aTC-based allocation algorithm is to reduce the wasted capacity in the task set ($tswc$) as a way to tighten the ETB derived for each task.

Our aTC allocator associates several ETB to each task τ_i, one for every possible task-to-core assignment, such that the task's *aTC* ETB$_{ai}$ is time composable for a particular allocation a_i, of the task set. For every task τ_i in the system, this requires: (1) understanding which tasks are group-mates to τ_i, as they are assigned to the same partition as τ_i; (2) understanding which tasks are siblings to τ_i, as they are assigned to other partitions than that of τ_i; and (3) obtaining an *aTC* ETB that upper bounds τ_i's execution time against a specific set of co-runners. This limits the scope of the analysis to consider that particular set of co-runners and the load that they can place on hardware shared resources. Section V discusses how to compute *aTC* ETB.

We use the following terms in defining aTC allocator.

• *Task wasted (CPU) capacity (twc).* We define *twc* for a task τ_i that runs in a workload, as the CPU time budget allocated to τ_i to account for the contention interference it can suffer from the contention caused by tasks running on other cores. Let C_i^{isol} denote the ETB for τ_i when run in total isolation, hence suffering no contention interference. Let $C_i^{wld_j}$ denote the ETB estimate for τ_i when run as part of workload wld_j. The latter bound accounts for the CPU budget inflation incurred by τ_i owing to the suffered contention interference. The *task wasted capacity* is defined as: $twc_i^{wld_j} = C_i^{wld_j} - C_i^{isol}$.

• *Workload wasted (CPU) capacity (wwc).* We define *wwc* as the addition of the *twc* of all tasks in a workload. That is, $wwc_j = \sum_{i=1}^{i=k_j} twc_i^{wld_j}$, where k_j is the number of tasks in workload wld_j.

• *Allocated task wasted capacity (atwc).* Under partitioned scheduling, for a given τ_i in a partition Φ, we define $atwc_i$ as the maximum twc_i in any of the workloads that can be constructed that include τ_i: $atwc_i = \max_{wld_j \in combs(\Phi)} twc_i^{wld_j}$, where $combs(\Phi)$ stands for all potential schedules wld_j of the task set containing τ_i, given partition Φ.

• *Task-set wasted CPU (tswc).* Finally, once a feasible assignment has been set for each individual task in the task set, we define *task set wasted capacity* as the sum of $atwc_i$ for all tasks in the task set $\forall \tau_i \in \mathcal{T}$.

aTC-allocator accounts as wasted capacity all duration of CPU time that a task is stalled due to contention interference. Hence, the zero-waste case occurs when all forms of contention interference that stem from contention on access to hardware shared resources are avoided. While this is not achievable in practice in a real-world multicore processor, using the zero-waste case as the ideal target for algorithm 1 helps reducing the wasted capacity per task ($atwc_i$ $\forall \tau_i \in \mathcal{T}$) which in turn reduces the tasks' ETB and increases the useful CPU utilization attained by the system.

Interestingly, before any task τ_i is allocated, the ETB assumed for every task in the task set is the *tsTC ETB*. That is, given that initially (when no partition yet exists) the task under study can be grouped with any other task, it must be assumed that all other tasks in the workload can be siblings and hence potential co-runners of τ_i. In subsequent iterations this approach progressively assigns tasks to cores. If a new task τ_j is assigned to a group where τ_i is, then τ_j cannot be a co-runner of τ_i, which causes a potential reduction in the

Algorithm 1 aTC_allocator ($\mathcal{T}$, $\mathcal{W}$, $lscht()$, $\mathcal{C}$)

Input: The task set($\mathcal{T}$), the ETB for each task under any configuration of siblings($\mathcal{W}$), local schedulability test $lscht()$ and the list of cores $\mathcal{C}$)

Output: A task allocation (Φ).

```
1:  $\mathcal{W} \leftarrow$ sort_by_wwc($\mathcal{W}$)
2:  while $\mathcal{W}$ not empty do
3:      $\mathcal{C} \leftarrow$ sort_by_spare_capacity($\mathcal{C}$)
4:      $c_j \leftarrow first(\mathcal{C})$
5:      while $c_j \in \mathcal{C}$ do
6:          $tuple_i \leftarrow find\_tuple(\mathcal{W}, lscht(), c_j)$
7:          if $tuple_i \neq \emptyset$ then
8:              break
9:          end if
10:     end while
11:     if $c_j \notin \mathcal{C}$ then
12:         return nonschedulable
13:     end if
14:     alloc($tuple_i$, $c_j$)
15:     update_capacity($c_j$)
16:     remove_overlapping_tuples($\mathcal{W}$)
17: end while
18: return $\mathcal{C}$
```

ETB estimate for τ_i. At any given step therefore, the ETB of any task τ_i is time composable with its siblings and with the not-yet assigned tasks assumed to be siblings of tasks in all groups. In this way, at every step when a task is assigned to a group φ_j, the ETB of all tasks in φ_j is reduced since the just-assigned task cannot be co-runners of the tasks in φ_j.

The aTC-allocator receives as input all possible permutations of task tuples, denoted $\mathcal{W}$, where each tuple contains between 1 and m tasks (m stands for the number of cores). Those tuple represent all potential workloads that can occur at any point in time. The workload wasted capacity (*wwc*) for each tuple, i.e. the addition of the *twc* for all tasks in a tuple, is derived by running the tuple on the target platform as described in Section V. wwc represents the CPU capacity that would be wasted if the tasks in the tuple were co-runners.

The algorithm starts by sorting tuples by their *wwc* (line 1) from higher to lower, and it also sorts cores by spare capacity (line 3), from lowest to highest. The algorithm then seeks the tuple with highest *wwc* such that it fits in the core with lowest spare capacity (line 6). A single-core (local) schedulability test (*lscht()*) is used to that end. If a tuple, $tuple_i$, is found it is assigned to that core, $core_j$ (line 14), preventing the tasks in that tuple from being co-runners and suffer that *wwc*. The capacity of $core_j$ is updated by recomputing and then adding the utilization of all the tasks that were already in $core_j$ plus those in $tuple_i$. The re-computation is necessary because now it is certain that the tasks already assigned to $core_j$ will not be co-runners to the tasks in $tuple_i$, so the ETB of all of them will monotonically decrease (line 15). Next, $tuple_i$ and every other tuple containing any of the tasks in $tuple_i$ are removed from $\mathcal{W}$ (line 16).

If no tuple fits in the core with the lowest capacity, the process starts with the second core with lowest spare capacity (lines 5 – 10). If no tuple is found to fit in any of the cores, the task set is not schedulable (lines 7 – 9). Conversely, if all tasks are assigned and the schedule is feasible, the algorithm returns the task set partition.

aTC requires deriving for each task an ETB that covers certain contention conditions. In this case, at each step of the algorithm, the timing analysis stage requires providing a ETB computed considering all the siblings of the task of interest as its co-runners at this time. Only tasks that are group mates of that task are not considered. The next section details how to obtain those ETB.

V. Deriving ETB under different contention conditions

As we mentioned earlier, the research on timing analysis for COTS multicore is not mature yet. The difficulty essentially lies in getting an accurate appreciation of the impact that contention in the use of processor shared resources can have on task execution time and on their ETB. To the best of our knowledge, state-of-the-art timing analysis techniques cannot assert a tight, safe and absolute worst-case value for COTS multicore processors.

In this paper we use pure measurements to determine ETB. Measurement-based timing analysis is a common practice for timing analysis on real processors in industrial domains that include space, automotive and elements of avionics [34], [17], [20]. End-to-end measurements are collected in controlled conditions and the ETB is derived by adding an engineering margin to the Longest Observed Execution Time (LOET) [20]. We acknowledge the uncertainty generated by measurements and the requirements their use places on the user to provide input vectors capable of reducing this uncertainty. Various approaches have been proposed to build confidence arguments in conformance with the requirements and practices of the specific application domain (e.g. automotive, space or avionics). The work in [14] reviews how safety assurance guarantees relate to stipulating bounds on execution time.

Deriving *fTC* ETB. Given a task set, $\mathcal{T}$, in order to provide fully time-composable ETB C_i^{fTC} for each task, we employ a recent approach based on *microbenchmarks*, also known as *resource stressing kernels* (RSK) [29], [21], [13], [11]. RSK are specialized, single-phase user-level programs designed to stress each of the hardware shared resources in the processor. Running task τ_i against a RSK represents a very pessimistic scenario of the inter-task interference that τ_i may experience during operation.

In [29], [13], the authors show that RSK produce greater contention interference on access to hardware shared resources than any real application or other benchmark that they could compare with. We therefore maintain that τ_i's *fTC* ETB can be determined when τ_i runs in parallel with that RSK.

During analysis, we run each task against the particular set of RSK designed for the target platform of interest. While τ_i runs we make sure that in the other cores the chosen RSK (rsk_i) runs at all times. With this method, the *fTC* ETB for τ_i is computed as: $C_i^{fTC} = max_{rsk_i \in RSK}(ET_i^{rsk_i})$, where $ET_i^{rsk_i}$ is the execution time of τ_i when running against rsk_i.

Challenges in deriving *tsTC* ETB and *aTC* ETB. We identify three main factors affecting the contention conditions of a given running task: its contention for access to hardware shared resources with its co-runners, its execution phases (with respect to the profile of use of hardware shared resources) and its input vectors. To enable partially Time Composable approaches it is necessary to understand all these factors and cover their timing effect to a sufficient level of confidence. In this regard, the use of COTS multicores and measurement-based analysis techniques introduces some uncertainty that has to be offset when building confidence arguments in accordance with the requirements and practices of the application domain.

In the absence of hardware support to contain or control contention interference, there is no established practice to derive *tsTC* and *aTC* ETB, which we can relate to. Our current solution is to perform experiments in which we tweak the sources of interference as can be done from the input vectors and the interleaving of tasks' execution phases. This technique is obviously sub-optimal in that it does not deliver full certainty, but it does nonetheless help appreciate the system resilience (respectively, vulnerability) to contention effects.

In order to account for the contention conditions generated on task τ_i by a given set of co-runners we make successive runs of the workload in which we shift the time at which we start the execution of each task in the workload with respect to τ_i. Figure 1 provides a schematic view of the case for a dual core arrangement. τ_i runs on core 0, c_0. At each run we shift the release offset of each task by a *shifting factor*, which leads to different *shifting points*. The granularity of the shifting factor and the number of cores in the processor determine the number of experiments to carry out (which ultimately is limited by the time that can be afforded for timing analysis during system development). This process is repeated for different input data vectors if the program's execution is sensitive to them. In our experimental setting, the EEMBC Automotive benchmarks we used present a stable single behavior so we observed negligible difference for different release offsets.

The challenge with global scheduling, which is the assumed scheduling regime for *tsTC*, is that, in theory, all tasks (grouped in workloads of m, where m is the number of cores) can run in parallel at any given point in time. This means that every task in the task set is a potential co-runner of any other task, in a great variety of actual schedules, which obviously leads to high ETB values to upper bound the many different contention conditions that may arise. Meanwhile, with partitioned scheduling, the choice for *aTC*, the schedules of interest are a much smaller set, limited to the interleaving of τ_i's siblings.

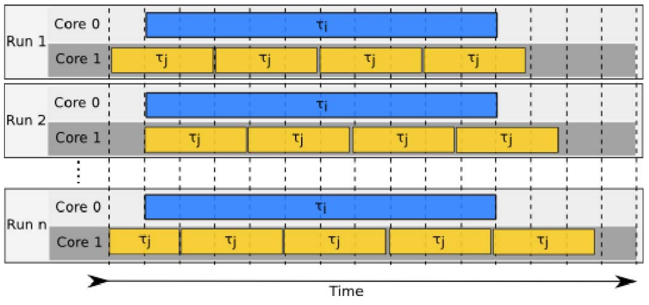


Fig. 1. Schematic view of experimental methodology to compute ETB estimates (dual core case). τ_j in Core 1 is shifted in each run.

VI. Experimental Evaluation

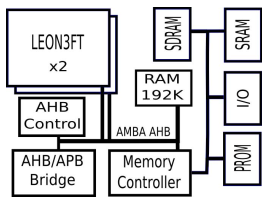
In the following we evaluate one possible realization of the *fTC* ETB and *aTC* ETB approaches for two industrial-quality COTS processors used in the space domain: a GR712RC platform implementing a dual-core LEON3; and a ML510 platform implementing a quad-core LEON4.

A. Experimental Setup

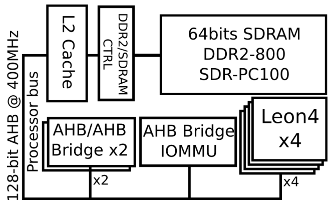
The GR712RC platform has 2 LEON3 processors (see Figure 2(a)), each comprising private first-level 16KB data and 16KB instruction caches. Cores are connected to the on-chip SRAM and the memory controller through an AMBA AHB bus [1]. The memory controller connects both cores to the off-chip SDRAM and SRAM devices. In the GR712RC the effect of the slowdown that a task suffers is mainly due to contention interference on accessing to the on-chip bus.

The ML510 platform contains a Virtex 5 FPGA that implements a prototype design of the NGMP, a SPARC V8 quad-core processor, developed by Aeroflex Gaisler and the European Space Agency, featuring the latest LEON core design, called LEON4 [4], [2]. Owing to FPGA space limitation, the ML510 platform does not have an on-core floating point unit. The LEON4 is a 32-bit 7-stage pipeline processor, comprising an always-taken branch predictor and private data and instruction caches of 16KB each. Both the instruction and the data caches have 32-byte lines and are 4-way associative. The data cache employs a write-through with no-write-allocate miss policy. Each LEON4-core connects to a shared 256KB L2 cache through an AMBA AHB processor bus with 128-bit data width and round-robin arbitration policy. The L2 cache uses the LRU replacement algorithm implementing a write-back, write-allocate policy. The L2 cache connects to the memory controller through a single memory channel shared by all cores (see Figure 2(b)). In the NGMP, the effect of the slowdown that a task (benchmark) suffers is due to contention interference in accessing the on-chip bus, the on-chip shared L2 cache and the memory bandwidth [13].

Benchmarks. In this work, for application load we used the EEMBC Autobench benchmark suite [27], which is deemed to capture well some elements of real-world real-time application behavior and for this reason is widely used in academic research. We chose benchmark programs that make use of

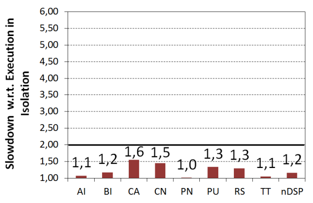


(a) GR712RC (LEON3)

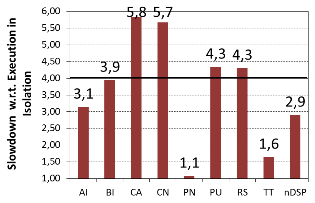


(b) ML510 (LEON4)

Fig. 2. Block diagram of the part of the NGMP and GR712RC architectures analyzed in this paper



(a) GR712RC (LEON3)



(b) ML510 (LEON4)

Fig. 3. Effect of contention interference on EEMBC and nDSP when running on the NGMP and the GR712RC

caches at various levels of intensity, from low to high: `aifirf` (AI), `bitmnp` (BI), `cacheb` (CA), `canddr` (CN), `pntrch` (PN), `puwmod` (PU), `rspeed` (RS) and `ttkprk` (TT). We further used a space-specific synthetic program known as Next Generation DSP Benchmark (*nDSP*)[2].

We developed a random task set generator based on [24]. The generated task sets all implement the equivalent of a sporadic task model, in which the arrival times of jobs of the same task are separated by a minimum inter-arrival time, referred to as the task period, and the tasks are independent. The task sets are generated such that the total utilization of every task set equals $U_{isol} = \sum_{i=1}^{n} \frac{C_i^{isol}}{P_i}$, with $n > m$ where m is the number of cores, n in the number of tasks, and C_i^{isol} is the ETB of τ_i when run in isolation. In the first step of our procedure U_{isol} is 'divided' between the n tasks. For the experiments discussed in this paper, we assigned 40% of U_{isol} to 20% of the n tasks, and 60% of the remaining U_{isol} to the remaining 80% of the tasks (similar results are obtained with other proportions). In the second step, each of the n tasks is then assigned timing load of a randomly chosen EEMBC executed on the real hardware. For each U_{isol} utilization we randomly generated 1,000 workloads.

B. Results with fTC ETB

Figure 3 shows the *fTC* ETB computed for *EEMBC* and *nDSP* for the GR712RC and the NGMP, plotted against the ETB obtained in isolation, i.e. C_i^{fTC}/C_i^{isol}, which reveals a wide range of variations.

For the GR712RC, the *fTC* ETB appear to be only marginally higher than C_i^{isol}, with a maximum increase of 55% and an average of 23%. For this board we may therefore use *fTC* ETB with acceptable pessimism in the determination of the schedulable utilization, for this choice breaks the circular dependence between ETB estimation and task scheduling and consequently simplifies the overall analysis process. For the NGMP instead, the *fTC* ETB, i.e. C_i^{fTC}, are significantly higher than C_i^{isol}, with a maximum increase of 5.8x and an average of 3.65x. This inflation is the consequence of larger contention interference arising from the use of more cores and more hardware shared resources. For the NGMP therefore, using *fTC* ETB would result in a significant loss of CPU capacity, which calls for the use of *aTC*.

[2]http://www.esa.int/TEC/OBDP/SEMFOU1VW3H_0.html

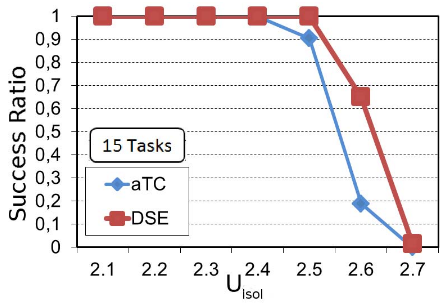
C. Results with aTC ETB

In order to gauge the efficacy of the algorithm presented in this work, we compared it against an alternative approach, called DSE or *Design Space Exploration*, which exhaustively explores all possible feasible allocations for every generated task set. The DSE approach is very time consuming and only serves the purpose of showing a challenging comparison here; e.g. for the case of 20 tasks DSE execution time can be up to 3 orders of magnitude higher than *aTC*. No such heavy approach would really be needed to apply the proposed method. As we mentioned before, for every selected task set utilization in isolation (U_{isol}) we generate 1,000 random workloads, all formed with EEMBC programs. For each such workload, DSE generates all potentially feasible allocations. For a task set with n tasks and a multicore processor with m cores, the total number of possible allocations is $\frac{n!}{m!(n-m)!}$.

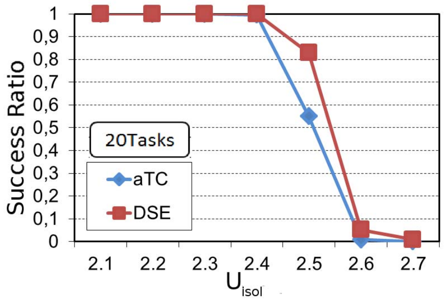
In Figure 4 the X-axis shows the U_{isol} of each task set and the Y-axis shows the success rate, i.e. the ratio of feasible partitions among all possible task-to-core assignments per task set. Diamonds show the success ratio obtained using the *aTC-allocator* while the squares for the *DSE*. Interestingly, the *aTC-allocator* success ratio appears to be quite close to that of DSE, which allows concluding that our algorithm is sufficiently good at finding feasible partitions. Notably, the success ratio decreases when the total utilization in isolation reaches $2.4 - 2.5$ out of 4, which is consistent with state-of-the-art results in the analysis of schedulable utilization with partitioned scheduling [6]. In the next section we analyze in detail why *aTC-allocator* does not reach good success rates for higher utilizations.

D. Schedulable utilization

In singlecore systems, the total utilization of a task set is determined by the simple addition of the utilization of individual tasks considered in isolation. Specific scheduling algorithms for singlecore processors have specific schedulable utilization thresholds, which determine necessary but not sufficient conditions for the feasibility of the task set. Task sets whose utilization falls below the applicable threshold are deemed feasible, with no feasibility guarantees provided otherwise. The efficacy of a given scheduling algorithm can thus be assessed by looking at the ratio of task sets that are feasibly scheduled from those deemed feasible.

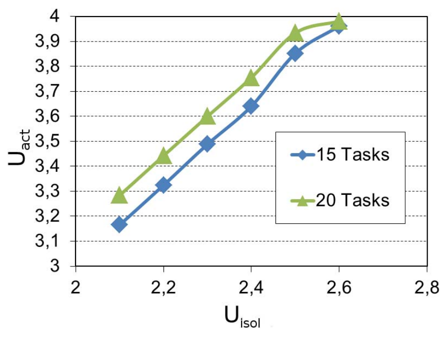


(a) 15 tasks

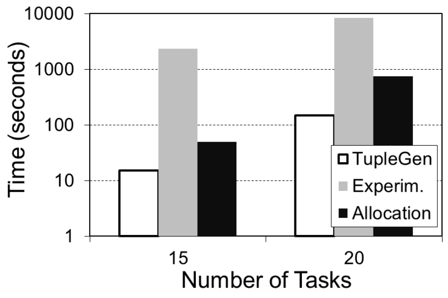


(b) 20 tasks

Fig. 4. Success ratio in finding feasible partitions of task sets with *aTC-allocator* and with the DSE on the NGMP



(a) Relation between U_{isol} and U_{act}



(b) Execution time overheads

Fig. 5. Other metrics for evaluating the performance of *aTC-allocator*

In multicores, to compute the utilization of a task we can either use its ETB in isolation, C_i^{isol} or its *fTC* ETB (C_i^{fTC}). The utilization associated to a task set varies in each case. It is $U_{isol}^{\tau_i \in \mathcal{T}} = \sum_{i=1}^{n} C_i^{isol}$ and $U_{fTC}^{\tau_i \in \mathcal{T}} = \sum_{i=1}^{n} C_i^{fTC}$. Those two ETB are, respectively, a lower bound and an upper bound to the effective task utilization whose actual value will depend on the contention interference incurred in operation for the chosen task-to-core assignment. The source of the complexity lies in the fact that ETB of a task in a multicore conceptually combines two elements which are difficult to determine exactly: the time the task actually uses the CPU to progress, and the time during which the task holds the CPU but is stalled while suffering contention interference. When partitioned scheduling is used, for example, the task-to-core assignment determines which tasks can run in parallel and consequently interfere with each other. This in turn allows determining the ETB for all tasks under that particular allocation (C_i^{act}). It follows that, in that case, the actual CPU utilization of the task set cannot be determined until the task-to-core assignment is fixed.

To overcome this difficulty, we use the *DSE* algorithm to find the feasible (partitioned) task sets and the resulting total utilization. At that point, we can apply our method to determine the actual effective utilization ($U_{act}^{\tau_i \in \mathcal{T}} = \sum_{i=1}^{n} C_i^{act}$) of the task sets that DSE deems feasible and see which of them stay feasible after using the *aTC-allocator*.

Figure 5(a) shows, for every feasible assignment found with the *DSE* algorithm, the total utilization of the task set considering the contention effects arising after the partitioning (U_{act}, Y axis) found with *aTC-allocator*, in relation to the total utilization computed considering tasks running in isolation (U_{isol}, X axis). We can see that, at $U_{isol}^{\tau_i \in \mathcal{T}}$ values around 2.4 and 2.5, where *aTC-allocator* and *DSE* stop finding feasible assignments (cf. figure 4), the $U_{act}^{\tau_i \in \mathcal{T}}$ values reach 3.65-3.75 and 3.8-3.9 respectively. This shows that *aTC-allocator* nearly fills all the cores.

E. Other considerations with aTC ETB

Run time in the offline analysis. *aTC-allocator* takes considerable time to run in the analysis phase for a real COTS multicore processor. We identify three main contributors to this need: the generation of all possible tuples, i.e. $\mathcal{W}$, the execution of those tuples on the target multicore for determining the ETB, and the time required by the *aTC-allocator* itself to compute feasible assignments. The generation of all tuples comprising all potential combinations (tuples) of $\{m, m-1, ..., 1\}$ tasks in $\mathcal{T}$ takes less than 2 minutes for any task set size, (see Figure 5(b)), on a Dell Latitude E6420 embedding an Intel Core i7 processor at 2.40GHz. The execution of all tuples in our ML510 board takes around 3 hours. Of course the duration of this step depends on the frequency of the board – 70 Mhz in our case – and the size of the applications under analysis. The execution of *aTC-allocator* for the problem at hand takes little less than 13 minutes.

Computational complexity. It is clear that our approach moves complexity into timing analysis to derive those *aTC* ETB. Most of the experimentation time is expected to be consumed in determining the *aTC* ETB. In the absence of hardware support for controlling contention interference, the evidence about the impact of such interference is obtained by exposing the task under different configurations of its potential co-runners. The experimentation time required may be larger if the tasks have release offsets and operation modes which need to be co-executed to capture their effect. Moreover, the cost may grow exponentially with the number of tasks and cores. Whereas reducing that overhead is part of our future work, to the best of our knowledge, there currently is no other viable proposal for the timing analysis of real-time tasks running on a COTS multicore processor that does not provide full support for contention control or removal.

VII. Conclusions

The advent of multicore processors challenges the viability of the two-step timing analysis approach followed for singlecore systems: contention interference effects in a multicore are much more complex in nature and fine in grain than what can be captured in compositional response time analysis by simply widening the tasks' ETB with the time intervals during which tasks cannot progress due to inter-task interference. This creates a dependence between the ETB derived for a task and its scheduling at run time. If *fTC* ETB estimates can be derived with low impact on pessimism this dependence is broken. If they cannot be derived we sketch a solution based on the new concept of *pTC* ETB for partitioned scheduling. We regard the problem for global scheduling to be harder to solve in a tractable manner.

We show two real processors representative of both cases: while in the dual-core LEON3-based GR712RC it is possible to use for each task its *fTC* ETB without incurring high overheads, this is not so for the quad-core NGMP as the *fTC* ETB may be 5.8x higher than in isolation (i.e. assuming no inter-task interference).

Acknowledgments

The research leading to this work has received funding from: the European Union's Horizon 2020 research and innovation programme under grant agreement No 644080(SAFURE); the European Space Agency under Contract 789.2013; and COST Action IC1202, Timing Analysis On Code-Level (TACLe). This work has also been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557. Jaume Abella has been partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717.

References

[1] *AMBA Bus Specification. http://www.arm.com/products-/system-ip/amba/amba-open-specifications.php*.

[2] *NGMP Preliminary Datasheet. http://microelectronics-.esa.int/ngmp/LEON4-NGMP-DRAFT-1-6.pdf*.

[3] B. Akesson et al. Predator: a predictable SDRAM memory controller. In *CODES+ISSS*, 2007.

[4] J. Andersson et al. Next generation multipurpose microprocessor. In *DASIA*, 2010.

[5] ARM Ltd. The ARM Cortex-A9 processors (white paper). http://www.arm.com/files/pdf/armcortexa-9processors.pdf, 2009.

[6] A. Bastoni et al. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *RTSS 2010*.

[7] A. Burns. Preemptive priority-based scheduling: An appropriate engineering approach. In *Advances in Real-Time Systems, chapter 10*. Prentice Hall, 1994.

[8] D. Compagnin et al. Putting RUN into practice: Implementation and evaluation. In *ECRTS 2014*.

[9] C. Cullmann et al. Predictability considerations in the design of multi-core embedded systems. In *ERTS*, 2010.

[10] G. Fernandez et al. Contention in multicore hardware shared. resources: Understanding of the state of the art. In *WCET Workshop*, 2014.

[11] G. Fernandez et al. Increasing confidence on measurement-based contention bounds for real-time round-robin buses. In *DAC*, 2015.

[12] G. Fernandez et al. Introduction to partial time composability for COTS multicores. In *ACMS SAC*, 2015.

[13] M. Fernández et al. Assessing the suitability of the NGMP multi-core processor in the space domain. EMSOFT, 2012.

[14] P. Graydon and I. Bate. Safety assurance driven problem formulation for mixed-criticality scheduling. In *MCS Workshop*, 2013.

[15] S. Hahn et al. Towards compositionality in execution time analysis – definition and challenges. In *CRTS*, December 2013.

[16] T. Kelter et al. Static analysis of multi-core TDMA resource arbitration delays. *Real-Time Systems*, 2013.

[17] R. Kirner and P. Puschner. Obstacles in worst-case execution time analysis. In *ISORC*, 2008.

[18] Y. Li et al. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS*, 2009.

[19] J. Mathai et al. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986.

[20] E. Mezzetti and T. Vardanega. On the industrial fitness of WCET analysis. *WCET Workshop*, 2011.

[21] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *EDCC*, 2012.

[22] M. Panić et al. Parallel many-core avionics systems. EMSOFT '14, 2014.

[23] M. Paolieri et al. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA'09*.

[24] M. Paolieri et al. IA3: An interference aware allocation algorithm for multicore hard real-time systems. In *RTAS'11*.

[25] M. Paolieri et al. Timing effects of DDR memory systems in hard real-time multicore. *ACM TECS*, 2013.

[26] R. Pellizzoni et al. Worst case delay analysis for memory interference in multicore systems. In *DATE*, 2010.

[27] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.

[28] P. Puschner et al. Towards composable timing for real-time software. In *STFSSD*, 2009.

[29] P. Radojković et al. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. In ACM TACO 2012.

[30] P. Regnier et al. RUN: optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *RTSS 2011*.

[31] S. Schliecker et al. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *DATE*, 2010.

[32] A. Schranzhofer et al. Timing analysis for TDMA arbitration in resource sharing systems. In *RTAS*, 2010.

[33] A. Schranzhofer et al. Timing analysis for resource access interference on adaptive resource arbiters. In *RTAS*, 2011.

[34] I. Wenzel et al. Measurement-based timing analysis. In *ISoLA*, 2008.