Resource Usage Templates and Signatures for COTS Multicore Processors

Gabriel Fernandez[*,†], Javier Jalle[*,†], Jaume Abella[†], Eduardo Quiñones[†], Tullio Vardanega[⋆], Francisco J. Cazorla[†,‡]

[*]Universitat Politècnica de Catalunya, Spain [†]Barcelona Supercomputing Center, Spain
[⋆]University of Padua, Italy [‡]Spanish National Research Council (IIIA-CSIC), Spain

ABSTRACT

Upper bounding the execution time of tasks running on multicore processors is a hard challenge. This is especially so with commercial-off-the-shelf (COTS) hardware that conceals its internal operation. The main difficulty stems from the contention effects on access to hardware shared resources (e.g., buses) which cause task's timing behavior to depend on the load that co-runner tasks place on them. This dependence reduces time composability and constrains incremental verification. In this paper we introduce the concepts of resource-usage signatures and templates, to abstract the potential contention caused and incurred by tasks running on a multicore. We propose an approach that employs resource-usage signatures and templates to enable the analysis of individual tasks largely in isolation, with low integration costs, producing execution time estimates per task that are easily composable throughout the whole system integration process. We evaluate the proposal on a 4-core NGMP-like multicore architecture.

1. INTRODUCTION

The research on timing analysis for multicore processors is still in its infancy. Especially so for COTS multicores, whose timing analysis is a complex challenge that needs to be solved before their adoption in safety-critical real-time systems industry may become viable. Deriving an Execution Time Bound (ETB)[1] for tasks running on multicores is challenged by the contention, also known as inter-task interference, occurring on access to hardware shared resources. Unless otherwise restrained, contention causes the execution time of any one task, hence its ETB, to depend on its co-runners. This has disastrous impact on system design and validation, as it conflicts with the incremental development and verification model that industry pursues to contain qualification costs and development risks. This industrial goal is sought by allowing individual subsystems to be developed in parallel against an agreed master specification, then qualified in isolation and incrementally integrated, with virtually no risk of functional regression at system level. In the time domain, incremental integration and qualification postulate *composability* in the timing behavior of individual parts, whereby the ETB derived for a task determined in isolation, should not change on composition with other tasks.

[1]Due to the lack of definitive Worst-Case Execution Time (WCET) estimation methods for COTS multicores, we use the term "execution time bound" (ETB) instead of WCET.

Several approaches have been proposed to deal with contention for multicore on-chip resources. On the one end of the conceptual spectrum in the state of the art, some authors propose computing ETBs so that they upper bound the effect of any possible inter-task interference a task may suffer on access to hardware shared resources. ETBs computed this way are *fully time composable* [9][10]. They therefore enable incremental integration and qualification, but at the cost of pessimism that may cause untenable over-provisioning, as the timing behavior actually occurring in operation may fall much below the level determined considering the worst-case interference possible in theory [22, 17, 11]. On the opposite end, other authors [5] propose – currently only for research platforms – to determine ETBs simultaneously for multiple tasks in specific configurations. Those ETBs are *non-time composable*, as they only hold valid for the tasks being analyzed and for their specific configuration. If any such parameter changes, all ETBs become invalid and the entire analysis has to be repeated.

In this paper, we tackle resource contention in multicores by proposing the new concepts of resource usage signature (RUs or $\mathcal{S}$) and template (RUl or $\mathcal{L}$). RUs and RUl aim at making the ETB derived for an interfered task τ, time composable with respect to a particular usage u of the hardware shared resources made by the interfering co-runner tasks. The tasks' ETBs are determined for a particular set of utilizations $\mathcal{U}$ such that the ETB derived for any $u \in \mathcal{U}$ upper bounds τ's execution time under any workload so long as the co-runners of τ can be proven to make a resource usage smaller than u. We explain later what "smaller" means and how this can be determined. This abstraction allows deriving time-composable ETBs for individual tasks in isolation for each $u \in \mathcal{U}$, so that the system integrator can safely pull those (interfering) tasks together as long as the resource usage made by their individual set of co-runners is upper-bounded by some u. All that the system integrator has to care in that regard is to characterize the the tasks' access to hardware shared resources (a low-cost abstraction of the task execution time), ignoring any finer-grained detail of that access behavior. In this paper we present an approach to produce ETBs in that manner, using measurement-based timing analysis techniques.

RUs and RUl are, on purpose, made to be agnostic to the particular timing distribution of the resource access requests to be considered. Hence, two tasks generating the same number of accesses to a resource, though with different patterns, have the same signature. The challenge in the proposed method is in determining an effect on the interfered task that upper bounds the interference caused by contending accesses, regardless of the time distribution of those accesses as made by the interfered and the interfering tasks. In this paper we make the following main contributions:
1) We develop the novel concepts of RUs and RUl for the timing analysis of COTS multicores and sketch an algebra of operators over RUs/RUl to enable their practical use.
2) We provide exemplary RUs and RUl for the cases when requests accessing shared resources incur either fixed or variable response latency.

3) We present an implementation of *RUs* and *RUl* for a 4-core NGMP-like [1] architecture, focusing on the bus and the memory controller as exemplars of on-chip shared resources. In our experiments we assume that the L2 cache is partitioned, as it is the case of the NGMP.

Our results show that when *RUs* and *RUl* are tailored to upper bound the access load caused by a task's co-runners, the ETB of that task is 1.36 times bigger than its execution time in isolation. If templates upper bound the highest number of accesses that any workload could produce, the (fully time composable) ETB would instead be 2.57 times bigger. *RUs* and *RUl* thus provide an effective way of abstracting resource usage in the quest for tight and trustworthy ETBs.

2. FORMALIZATION OF RUs AND RUl

RUs and *RUl* allow analyzing, for the most part in isolation, the timing behavior of tasks, by abstracting the perturbation that they may incur from the contention for hardware shared resources occurring on a multicore caused by co-runner tasks.

2.1 Resource Usage signature (*RUs*)

A *RUs* abstracts the use of resources of a given interfered task, τ_A. Once computed, it will be used for τ_A's multicore timing analysis instead of τ_A itself.

We describe the use of a hardware shared resource through a set of features, which correspond to quantitative values. A *RUs* for task τ_A, is a vector $\mathcal{S}_A = (a_1, a_2, ..., a_n)$ that contains the aggregate of relevant features that characterize all the hardware shared resources, for the evaluation of contention effects. Since *RUs* are quantitative, the *RUs* of distinct tasks are comparable and can also be combined together to form a joint *RUs*.

Consider the reference multicore architecture shown in Figure 1(a), where the bus and the memory are shared. Further consider two types of accesses to those shared resources, for read and write operations respectively. In this case, *RUs* have at most 4 features: bus reads (n_{rd}^{bus}) and writes (n_{wr}^{bus}); memory reads (n_{rd}^{mem}) and writes (n_{wr}^{mem}). *RUs* are thus defined as $\mathcal{S}_A = (n_{rd}^{bus}, n_{wr}^{bus}, n_{rd}^{mem}, n_{wr}^{mem}) = (a_1, a_2, a_3, a_4)$.

If the bus were the only shared resource, the *RUs* of a task τ_A would be abstracted as a *RUs* with two features: n_{rd}^{bus} and n_{wr}^{bus}. If both types of requests hold the bus for the same duration, the *RUs* would consist of a single feature corresponding to the sum of n_{rd}^{bus} and n_{wr}^{bus}, i.e., $\mathcal{S}_A = (n_{rd}^{bus} + n_{wr}^{bus}) = (a_1 + a_2)$. The addition of $\mathcal{S}_B$ to $\mathcal{S}_A$ is given by $\mathcal{S}_A + \mathcal{S}_B = (a_1 + a_2 + b_1 + b_2)$. For comparison, instead, we say that $\mathcal{S}_A$ dominates $\mathcal{S}_B$, $\mathcal{S}_A \succsim \mathcal{S}_B$, if the interference by the former is greater than that by the latter: $a_1 + a_2 \geq b_1 + b_2$.

This reasoning easily extends to the more realistic scenario in which the bus holding times are asymmetric; for example, with reads holding the bus longer than writes. In that case, the *RUs* for τ_A could be either single-feature, considering all accesses as "long" accesses (counting writes as reads in the example), or multi-feature (two, in the example), i.e., $\mathcal{S}_A = (a_1, a_2) = (n_{rd}^{bus}, n_{wr}^{bus})$. In the latter formulation, addition and comparison change as follows: addition is defined as vector addition, i.e., $\mathcal{S}_A + \mathcal{S}_B = (a_1 + b_1, a_2 + b_2)$; for comparison, $\mathcal{S}_A$ dominates $\mathcal{S}_B$, $\mathcal{S}_A \succsim \mathcal{S}_B$ if $(a_1 \geq b_1) \wedge (a_2 \geq b_2)$.

2.2 Resource Usage template (*RUl*)

RUl have the same form as *RUs*, namely, a vector of features $\mathcal{L}_K = (k_1, k_2, ...k_n)$, but with a different use. *RUs* abstract tasks according to their use of the shared resources while *RUl* abstracts the use of the shared resources so that $\mathcal{L}_K$ can be used as an upper bound to the interference effects caused by any task τ_i whose *RUs* $\mathcal{S}_i$ is such that $\mathcal{L}_K \succsim \mathcal{S}_i$ (i.e. $\mathcal{S}_i$ is dominated by $\mathcal{L}_K$).

Tasks are made time composable against some *RUl* $\mathcal{L}_K$ so that the ETB derived for a given task τ_A and for that *RUl*, denoted ETB_A^K, upper bounds τ_A's execution time inclusive of the interference that the contenders of τ_A, whose *RUs* do not exceed $\mathcal{L}_K$, may cause.

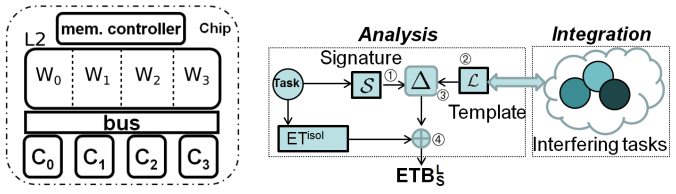

Figure 1: Reference multicore architecture (a), and main steps in the *RUs* and *RUl* methodology (b).

Returning to the example in which the bus is the sole shared resource with all accesses to it incurring the same contention effect: for a $\mathcal{L}_K$ that captures a given number of accesses to the shared bus, we want to determine the highest impact by $\mathcal{L}_K$ on ETB_A, so that ETB_A^K can be regarded as a time-composable bound for τ_A in any workload in which $\mathcal{L}_K \succsim \sum_i \mathcal{S}_i$ for all co-runner tasks τ_i of interest.

A maximally time-composable template $\mathcal{L}_{TC}$ exists, which is an upper bound for any workload. $\mathcal{L}_{TC}$ corresponds to the case in which all accesses from the signature suffer the highest contention from the $N_c - 1$ contending cores. In that case, every access from $\mathcal{S}_A$ contends with $N_c - 1$ other accesses, i.e., $\mathcal{L}_{TC} = (N_c - 1) \times \mathcal{S}_A$. Any $\mathcal{L}_K \succsim \mathcal{L}_{TC}$ would produce exactly the same result as $\mathcal{L}_{TC}$, since τ_A cannot be interfered more than the accesses in its signature $\mathcal{S}_A$.

2.3 RUs and RUl through an example

In this section we return to the case in which the bus is the sole shared resource and all accesses to it incur the same contention effect. For now we limit our attention to two cores. The task under analysis, τ_A, runs in one of the two cores. The contending requests from the two cores are arbitrated with the round-robin policy.

Figure 1(b) depicts the process we follow when the proposed approach is applied to this case. First, we obtain the *RUs* of τ_A, denoted $\mathcal{S}_A$. In the example architecture, the *RUs* of tasks using the shared resource is the number of accesses they make, a for τ_A, hence $\mathcal{S}_A = (a)$. Our approach treats contention such that the ETB of τ_A can be derived by upper bounding τ_A's execution time considering the interfering effect that it incurs when its co-runner task, whatever it is, makes up to k contending accesses to the shared resource. To this end we define a *RUl* $\mathcal{L}_K$, which is the system integration parameter that defines the inter-task interference to be considered in the determination of τ_A's ETB. The abstraction captured by $\mathcal{L}_K$ with $\mathcal{L}_K = (k)$ is a *RUl*.

Once the $\mathcal{S}_A$ and $\mathcal{L}_K$ are defined, we determine Δ_A^K, the increment to be applied to the execution time that τ_A may incur, to capture the contention effect from $\mathcal{L}_K$. This corresponds to step 3 in Figure 1(b). More precisely, Δ_A^K upper bounds the increment that the execution time of a task τ_A with at most a accesses to a shared resource may suffer from k contending requests. ETB_A^K (i.e τ_A's ETB determined under the *RUl* $\mathcal{L}_K$) is computed as the summation of ET_A^{isol}, the execution time of τ_A when running in isolation, without contention, and Δ_A^K, the increment that upper bounds the contention effects from any k interfering accesses. This corresponds to step 4 in Figure 1(b). Overall, ETB_A^K is time composable against any co-runner task τ_B with signature $\mathcal{S}_B = (b)$, as long as the *RUs* of the co-runner is lower than $\mathcal{L}_K$, which means that τ_B makes $b \leq k$ contending accesses. We denote this as $tc(ETB_A^K, \tau_B)$, which holds if $b \leq k$.

RUs abstract the distribution of requests over time. Taking into account the exact distribution of requests over time, for instance in the form of requests arrival curves [20], would potentially enable deriving tighter ETB. However, deriving such distributions is complex, as programs normally have multiple paths of execution, each with its own access pattern (distribution). And, paradoxically, considering these particular distributions would decrease timing composability. Instead, our approach only requires the tasks' access count for every individual shared resource, as well as ET_i^{isol} (execution time in isolation) for each individual task τ_i. Notably, both are already had with high accuracy by state-of-the-art technology, e.g., [23]. With our approach, the ability to abstract away from the need to know the exact points in time at which requests would be made to shared resources releases the system integrator from the obligation of adopting rigid and inflexible scheduling decisions (which fares poorly with the development unknowns of novel systems) or from the labour-intensive cost of exact analysis.

Our approach requires the user to set the RUl to capture the potential co-runner tasks precisely. The spectrum of this capture has two ends. On one extreme we find the time-composable templates, $\mathcal{L}_{TC}$, which represent an upper bound for RUl. However, if RUl is close to that template, the ETB of tasks might be unnecessarily increased. On the opposite extreme, if RUl is too small, it constrains the choice of tasks that may be allowed to run in parallel. A simple solution consists in deriving for each task an ETB under different RUl, such that at integration time, the smallest RUl that upper bounds the signature of the actual co-runner tasks is used. With this, the residual part of the timing verification at system integration is small and simple. Selecting the proper number of RUl represents a trade-off between effort and accuracy: the higher the number of RUl the lower the over-estimation of ETB and the greater the analysis time, and vice-versa. Finding appropriate RUl is a standard optimization problem that is part of our future work.

In the example considered in this section we have made several simplifications to facilitate understanding: two cores, one single type of access, synchronous accesses (i.e. the core stalls when the access occurs until served) and a single shared resource. In real processors we have different types of accesses to the shared resource (synchronous and asynchronous), each with a distinct access latency. Hence, simply bounding the effect of contention by adding access counts is not enough.

3. RUs & RUl FOR MEASUREMENT-BASED TIMING ANALYSIS

Next we present one concrete realization of RUs and RUl for use with measurement-based timing analysis (MBTA), specifically for a NGMP-like processor architecture [1].

3.1 Methodology

Our approach uses *micro-kernels* [22, 17, 11], a set of single-phase user-level programs with a single execution behavior designed so that all their operations access a given shared resource, e.g. the bus. Micro-kernels consist of a main loop whose body includes a substantial number (e.g. 256) of instructions designed to generate a steady stress load on target resources. The fact that the loop body executes repeatedly the same instruction causes the target resource to be continuously accessed. Moreover, placing a high number of identical instructions in the loop body drastically reduces the impact of control instructions (down to 2-4%) [11]. For the architecture in Figure 1(a), a loop body including load instructions that hit in the L2 cache stresses the bus. We consider two types of micro-kernels:

Resource stressing kernels, $RStK$, place a configurable load on a given shared resource, so that running a task against a $RStK$ may represent contention scenarios of interest.

In theory, one could design a *worst-contender* kernel that generates the maximum contention that a task τ_i can suffer. However, such kernel would be specific for the task to be interfered and for the target processor [22]. Consider for example, a single shared resource arbitrated by a least-recently-used policy, where the task that accessed the resource last gets the least priority. In that case, the worst-contender kernel should generate a request in exactly the same cycle as the task of interest, so that every request from that task gets delayed by the contender, and for the next round of arbitration the task has the lowest priority again. The level of control required on the application behavior and the granularity of intervention are too fine-grained and laborious to be used in practice [22].

Resource sensitive kernels, $RSeK$, are designed to upper bound the execution time increase suffered by any other task, with a smaller or equal signature, owing to the interference from a given template $\mathcal{L}_K$. Consider a scenario in which bus accesses hold the bus for a constant duration. Further assume that we want to determine Δ_A^K for τ_A, i.e its ETB increment due to a template $\mathcal{L}_K$ with k accesses. Intuitively, one could get an estimate of it by running τ_A several times against a $RStK$ that makes k accesses. However, in order to gain confidence in the ETB obtained, the experiment should be repeated with different *alignments* of the $RStK$, so that the interleaving of accesses varies enough and the worst case can be observed in a measurement. In practice, this may require excessive experimentation effort. The need for repeating the experiments with different alignments stems from the uncertainty on the time distribution of accesses, which is hard, if at all possible, to measure and control by timing analysis technology. We can therefore conclude that studying the task under analysis against micro-kernels is not viable. Instead, we use micro-kernels *to model both the interfered and the (set of) interfering tasks*: $RStK$ and $RSeK$ are designed to account for bad alignments of requests: $RSeK$ is made of instructions that cause accesses to the shared resource and that continuously contend with $RStK$ requests.

We define $\Delta_{RSeK}^{RStK} = ET_{RSeK}^{RStK} - ET_{RSeK}^{isol}$, where ET_{RSeK}^{RStK} is the execution time when a given $RSeK$ with the same signature as task τ_A runs against a $RStK$ implementing a template $\mathcal{L}_K$ with k accesses; and ET_{RSeK}^{isol} the execution time when the $RSeK$ runs in isolation. For task τ_A, let $\Delta_A^K = ET_A^K - ET_A^{isol}$ be the execution time increase τ_A suffers when it runs against $\mathcal{L}_K$. $RSeK$ and $RStK$ are designed so that $\Delta_{RSeK}^{RStK} \geq \Delta_A^K$ holds for any request alignment of τ_A under $\mathcal{L}_K$ contention. To that end, we run the $RSeK$ in isolation and then against $N_c - 1$ copies of $RStK$ so that all $RSeK$'s accesses to the shared resource suffer high contention, causing a measurable Δ_{RSeK}^{RStK} to emerge. In the next section we show how to derive the number of accesses of the $RSeK$ and the $RStK$, based on the number of accesses of the template and signature under consideration.

Δ_{RSeK}^{RStK} is used to compute the ETB estimate for τ_A as follows: $ETB_A^K = ET_A^{isol} + \Delta_{RSeK}^{RStK}$. ETB_A^K is composable with any set of interfering tasks against which τ_A runs in parallel, if their total number of accesses is lower or equal to k. That is, the addition of the signatures of the interfering tasks is dominated by $\mathcal{L}_K$: $(S_i + S_j + ... + S_l) \precsim \mathcal{L}_K$. Interestingly, given a task τ_B whose signature is dominated by τ_A, i.e. $S_B \precsim S_A$, the obtained Δ_{RSeK}^{RStK} for τ_A can be used to upper bound τ_B's execution time: $ETB_B^K = ET_B^{isol} + \Delta_{RSeK}^{RStK}$.

Overall, RUs and RUl provide powerful abstractions for the interfered and the interfering tasks, which simplify the integration of multiple tasks by combining their signatures.

3.2 The case of a NGMP-like architecture

Our reference multicore architecture [1] comprises $N_c = 4$ symmetric cores, see Figure 1(a), each equipped with private instruction cache (IC) and data cache (DC). The cores have an in-order time-anomaly-free design [16]. Load operations are blocking, whereby the pipeline is stalled until the load is re-

interfered	st			l2h			l2m		
interfering	l2h	l2m	st	l2h	l2m	st	l2h	l2m	st
Impact	7	2	2	7	2	2	2x7=14	2x2=4	2x2=4

Figure 2: Impact from/to the different access types to the bus.

solved. Each core has one 2-entry write-buffer that holds store requests until they are resolved, without stalling the processor. The processor is stalled solely to preserve memory consistency, when a store finds the write-buffer full or a load operation finds the write-buffer non-empty.

Bus. Our example processor implements round-robin bus arbitration so that if, in a given round, core $c_i, i \in \{1,..,N_c\}$ is granted access to the bus, the priority ordering in the next round is: $c_{i+1}, c_{i+2}, ..., c_{N_c}, c_1, c_2, ..., c_i$. A lower priority core can use the bus when all higher priority cores do not use it. The *bus access jitter* that a task incurs on access to the bus, depends not only on the number of co-runners but also on the way their requests interleave. The worst contention situation happens when a task τ_B assigned to core c_i requests the bus in a given round of arbitration, simultaneously with tasks in all other cores and the previous round was assigned to c_i.

L2 cache. The L2 cache processes up to one miss per core at a time and allows hit-under-miss and miss-under-miss so that when a miss from a core is processed, hit/miss requests from other cores can be served. The 4-way L2 is partitioned so that every core is allowed to use 1 way[2].

Memory controller. The L2 sends a request to the memory controller on every L2 miss. Requests are stored in a FIFO request queue, with one entry per core. The memory controller assumes a single DRAM device with close-page policy.

3.3 Bus

The bus handles three distinct request types, which differ in the contention they induce and suffer. Stores (*st*) either hit or miss on the L2, which are served immediately by the L2 and hold the bus for 2 cycles. L2 load hits (*l2h*) hold the bus for 7 cycles because they are not split by the bus and insert wait states on the bus for the hit latency of the L2 (5 cycles). L2 load misses (*l2m*) that are split by the L2 and perform a new arbitration whenever the L2 responds to the miss, holding the bus 2 cycles in each arbitration. Figure 2 shows the contention suffered by a source (interfered) request by another (interfering) request for all request types. *l2h* generate the highest contention and *l2m* are the most affected since they suffer two rounds of arbitration: *l2m* can therefore be interfered twice by two concurrent contending requests, one round of arbitration per each such request.

Our approach based on *RUs* and *RUl* does not require knowing the exact time of request issue, but whether they have asymmetric timing behavior in the impact they suffer and they cause to other request types so that RStK and RSeK can be designed with the appropriate request types. The *RStK* and *RSeK* for the bus are called *BStK* and *BSeK*:

BSeK (abstracting interfered task bus usage). The signature of a task τ_A running in this architecture may take different forms, with different levels of tightness and experimentation effort. The canonical signature for the bus contains the number of accesses of each type made by the task. That is: $\mathcal{S}_A^{bus} = (a_{st}, a_{l2h}, a_{l2m})$. This can be simplified by realizing that *l2h* and *st* access the bus once whereas *l2m* do it twice with exactly the same timing as *l2h* and *st*. Moreover, the delay suffered by an access does not vary whether the access was generated by a *l2h*, *st* or *l2m*. Hence, signatures have the form: $\mathcal{S}_A^{bus} = (a_{st} + a_{l2h} + 2 \times a_{l2m})$.

BSeK can be implemented with either *l2h* or *st*. *l2m* are not appropriate as it is not possible to place high pressure on the bus with *l2m* since they miss in cache and take long to be served from memory, leaving the bus idle in the meantime. *l2h* and *st* instead can place very high pressure on the bus. Our approach considers *BSeK* to only have *st* operations.

BStK (abstracting interfering task(s) bus usage). Templates can be mono- ($\mathcal{L}_{1D}$) or bi-dimensional ($\mathcal{L}_{2D}$).

$\mathcal{L}_{2D}$. *st* and *l2h* generate different impact on the bus (recall that *l2m* are equated to 2 *st*). In particular, *l2h* produces the highest impact and *st* the lowest. This allows generating bi-dimensional templates: $\mathcal{L}_{2D} = (k_{l2h}, k_{2\times l2m+st})$, whereby *BStK*s comprises load L2 hit accesses and store accesses to generate each respective type of interference.

$\mathcal{L}_{1D}$ templates comprise only *l2h*, which generate the highest interference. A given $\mathcal{L}_{1D} = (k_{l2h})$ with k *l2h* accesses upper bounds the impact that one or several tasks, whose bus access count is lesser or equal to k, can generate on any other interfered task. $\mathcal{L}_{1D}$ are easier to generate and simplify experimentation, but they increase the pessimism of ETBs, since *st* are considered to generate the same impact as *l2h*.

Putting it all together. Deriving the access count for *BSeK* and *BStK* varies for $\mathcal{L}_{1D}$ or $\mathcal{L}_{2D}$ as we show next.

$\mathcal{S}_A - \mathcal{L}_{1D}$. Let a and k be the number of accesses in the signature $\mathcal{S}_A$ and the template $\mathcal{L}_K$ respectively. Running *BSeK* and *BStK* concurrently, we derive an upper bound to the increase in execution time (the delta) that k accesses of the template can have on the a accesses of the signature. If $k \geq (N_c - 1) \times a$ then each request of $\mathcal{S}_A$ suffers the impact of $N_c - 1$ contenting requests. If this is not the case, only $\lceil k/(N_c - 1) \rceil$ requests from $\mathcal{S}_A$ suffer impact.

The number of request accesses generated by the *BSeK* is given by $N = min(a, \lceil k/(N_c - 1) \rceil)$. By running this *BSeK* against $N_c - 1$ *BStK* copies, each having a number of accesses largely above N, we derive an upper bound to the impact that $\mathcal{L}_K$ has on $\mathcal{S}_A$. The impact that a task can suffer due to a template $\mathcal{L}_K$ with k *l2h* is upper bounded as: $\Delta_{BSeK}^{BStK} = ET_{BSeK}^{BStK} - ET_{BSeK}^{isol}$. The ETB derived for a given task τ_A and template $\mathcal{L}_K$ is: $ETB_A^K = ET_A^{isol} + \Delta_{BSeK}^{BStK}$.

$\mathcal{S}_A - \mathcal{L}_{2D}$. In this case we account for the fact that requests sent by the interfered task, τ_A, suffer different interference by the *l2h* and *l2m*/*st* sent by the interfering tasks, abstracted in $\mathcal{L}_{2D}$. In this approach we pair up every request in τ_A with $N_c - 1$ requests in $\mathcal{L}_{2D}$ causing the highest interference (*l2h*) on the former. If the number of those requests in $\mathcal{L}_{2D}$ is exhausted, we pair up τ_A requests with those in $\mathcal{L}_{2D}$ causing the second worst interference (*st*).

We generate two *BSeK* and *BStK* pairs to capture the impact that accesses in $\mathcal{S}_A$ suffer from *l2h* and *l2m*/*st* in $\mathcal{L}_{2D}$ so that:

$$\Delta_{BSeK}^{BStK} = \left(\Delta_{BSeK_1}^{BStK_l} + \Delta_{BSeK_2}^{BStK_s} \right) \quad (1)$$

$BSeK_1/BStK_l$ and $BSeK_2/BStK_s$ capture the interference on τ_A's accesses caused by the *l2h* and *l2m*/*st* in $\mathcal{L}_{2D}$ respectively. $BSeK_1$ and $BSeK_2$ have different number of *st* operations, N_1 and N_2. $BStK_l$ comprises *l2h* operations whereas $BStK_s$ comprises *st* operations.

Let assume for example $a = 30$, $k_{l2h} = 60$, and $k_{st} = 80$. In this case, $BSeK_1$ has $N_1 = min(30, \lceil 60/3 \rceil) = 20$ *st*, which we pair up with 20 accesses in $\mathcal{S}_A$; and $BSeK_2$ has the rest of accesses in $\mathcal{S}_A$, $N_2 = 30 - 20 = 10$ *st*, which we pair up with 3×10 requests out of the 80 accesses in k_{st}. The remaining 50 *st* in k_{st} are not paired since they will not cause further impact on $\mathcal{S}_A$. Overall, an upper bound to the impact that an application can suffer due to $\mathcal{L}_{2D}$ is given by:

$$ETB_A^K = ET_A^{isol} + \left(\Delta_{BSeK_1}^{BStK_l} + \Delta_{BSeK_2}^{BStK_s} \right) \quad (2)$$

For the memory controller we follow the same principles as for the bus, with the particularity that the impact from/to the read/write request types is homogeneous. Hence we only need

[2]The ARM A9 and the NGMP do implement this feature.

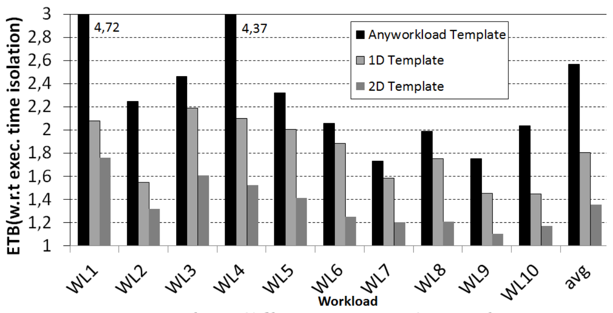

Figure 3: ETBs for different templates for 10 4-task workloads. Results are normalized to the execution time in isolation.

$\mathcal{L}_{1D}$ templates. The *RStK* and *RSeK* for the memory are called *MStK* and *MSeK*.

3.4 Multi-resource signatures

In the presence of multiple shared resources, the signatures and templates must cover the hardware features so as to soundly upper bound contention in each of them. For the reference architecture considered in this work, signatures and templates are as follows: $S_A^{bus+mc} = (a_{st} + a_{l2h} + 2a_{l2m}, a_{mem})$ and $\mathcal{L}_K^{bus+mc} = (k_{st} + 2k_{l2m}, k_{l2h}, k_{mem})$.

It is possible that a task suffers contention in several shared resources simultaneously, so that the impact of the contention does not accumulate but rather overlaps. However, determining trustworthy bounds to the degree of overlap in the contention suffered on requests to different resources is complex. Signatures and templates are intentionally made agnostic to the distribution of requests over time. As we focus on the number of requests to each resource rather than on their timing, it is difficult to determine how contending requests overlap. Our current approach assumes no overlap in contention, which in our time-anomaly free processor design is a safe assumption on the maximum impact of contention. Overall, in the presence of a template for the bus $\mathcal{L}^{bus}$ and the memory $\mathcal{L}^{mc}$ (a.k.a. $\mathcal{L}^{bus+mc}$), a task is assumed to suffer the sum of the contention generated by both templates:

$$ETB_A^{\mathcal{L}_K^{bus}+\mathcal{L}_K^{mc}} = ET_A^{isol} + \Delta_{BSeK}^{BStK} + \Delta_{MSeK}^{MStK}$$

4. EVALUATION

For our evaluation, we model a 4-core NGMP-like symmetric multicore [1] at 150 MHz comprising a bus connecting cores to the L2 cache and an on-chip memory controller. This processor model is relevant as it constitutes a potential baseline for the space domain. To model the DRAM memory system we use DRAMsim [6], a well-known memory simulator with which we model a close-page DDR2-667 [15] memory. As part of a study carried out for the European Space Agency we evaluated the performance estimates provided by our simulator against a real NGMP implementation, the N2X [3] evaluation board, using a low-overhead kernel that allowed cycle-level validation. The results obtained for the EEMBC Automotive [21] benchmarks, the reference benchmarks used in this paper, showed an execution time deviation of less than 3% on average. For the NIR HAWAII benchmark [13], the inaccuracy was less than 1%.

Our *RSeK*/*RStK* approach works on the premise that the contention suffered by each request of the *RSeK* upper bounds the contention suffered in any other scenario. The authors of [25] show that round-robin arbitration can have anomalous cases when a higher number of contenders introduces less contention on the bus. In fact, we show in [8] that the *RStK* cannot necessarily generate the worst (maximum) contention on *RSeK*, due to the alignment of requests. To solve this, we applied a solution based on adding *nop* operations between *RSeK* requests to modify their alignment. For instance, in the case of the bus, since we use *store* requests for the *RSeK*(see Section 3.3), we prove in [8] that each *RSeK*'s request suffers the maximum contention. In our reference architecture, if *load* operations were used in the *RSeK*, each request would suffer exactly one cycle less than the maximum contention on each request as shown in [8], which can be compensated with a correction factor.

4.1 Experimental results

Our evaluation was carried out along 2 axes. First, we compared the tightness of 1D and 2D templates against fully-time composable ETB, that can be obtained by software [11][22] or hardware [19] methods. Secondly, we compared 2D templates, for which tighter results are obtained, to the case in which the task under analysis runs agains *RStK*.

1D vs 2D signatures. Figure 3 compares the scenario with a fully-time composable template, $\mathcal{L}_{TC}$, valid for any workload (*any workload template* in the figure), with 1D ($\mathcal{L}_{1D}$) and 2D ($\mathcal{L}_{2D}$) templates fitting the potential interference in the corresponding workload. We analyze 10 randomly generated workloads and show results for the benchmark running on core 0. Similar results are obtained for the other cores.

For instance, for workload W8 <*pntrch*(PN), *basefp*(BA), *a2time*(A2), *tblook*(TB)>, we consider PN as the task under analysis and a template that corresponds to the aggregate of signatures of the three other benchmarks. This causes $\mathcal{L}_{1D}$ to have $564,227$ bus accesses (as many as the addition of bus accesses of *BA*, *A2* and *TB*). This is abstracted by *RUs*/*RUl* so that only $564,227/3 = 188,076$ bus accesses from PN suffer high contention and the rest suffers no contention. To measure this effect, we run a *BSeK* with 188,076 accesses against 3 BStK with a large number of accesses. The same process is followed for the memory. $\mathcal{L}_{2D}$ is generated analogously, but considering separately *l2h* and *st* bus accesses.

Figure 3 shows the ETB for the first benchmark in the workload (under *anyworkload*, $\mathcal{L}_{1D}$ and $\mathcal{L}_{2D}$), normalized to its execution time in isolation. We observe that fitting templates to actual contention ($\mathcal{L}_{1D}$ and $\mathcal{L}_{2D}$) in the workload tightens ETBs significantly. This effect is particularly noticeable for WL1 and WL4. Also, in all cases $\mathcal{L}_{2D}$ provides tighter ETBs than $\mathcal{L}_{1D}$. This is so because with $\mathcal{L}_{1D}$ all accesses to the bus are assumed to be *l2h*, which generate the highest contention, while $\mathcal{L}_{2D}$ better captures the fact that there are two type of requests generating different contention (*l2h* and *l2m-st*). For instance, WL4 has a normalized ETB of 4.37 (more than 4x the execution time in isolation) when using a template valid for any workload. If we use $\mathcal{L}_{2D}$ for this workload, the ETB is only 1.53. Overall, our approach allows reducing the ETB from 2.57 to 1.8 with $\mathcal{L}_{1D}$ and 1.36 with $\mathcal{L}_{2D}$ templates on average for the 10 workloads.

Owing to strict page limits we are unable to report the contention impact generated by the memory. Notably however, in our processor set-up the bus has higher impact than the memory, as the L2 cache filters out most memory accesses. Of the contention impact in $\mathcal{L}_{2D}$, 78% stems from the bus and only 22% from the memory.

RUs/RUl vs. EEMBC/RUl. In order to assess the pessimism incurred in ETB obtained with $\mathcal{L}_{2D}$ we compared them with the execution time for the task (i.e EEMBC), denoted *ET*, taken when the task run as part of a workload comprising *RStK* [11][22]. This workload represent a pessimistic yet possible contention scenario that the task can suffer. Figure 4 shows ETB obtained with $\mathcal{L}_{2D}$ relative to *ET*. Notably, the incurred pessimism was always below 45%, 20% on average. We contend that the benefits provided by RUs/RUl in the simplification of timing analysis upon system integration, pays off for the increase in WCET estimates.

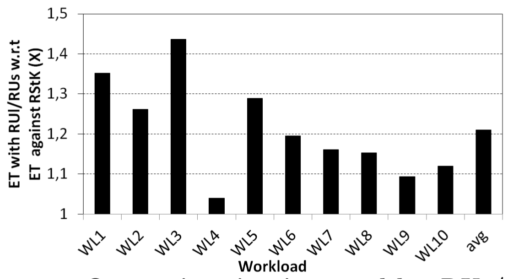

Figure 4: Overestimation incurred by RUs/RUl

5. RELATED WORK

Contention on access to hardware shared resources has been thoroughly studied in the state of the art. A taxonomic summary of the relevant works can be found in [7]. Authors in [4] propose a methodology to obtain the signature of tasks and replace them with kernels that mimic their shared resource usage pattern as a way to reduce the variability in measurement-based analysis. Instead, we use signature and templates to abstract the contention tasks cause and suffer, bounding contention effect [8]. Works addressing off-chip contention assume no contention for on-chip resources, which are assumed replicated. Off-chip contention for the bus is handled with TDMA buses [2] whose analysis case is the worst possible alignment of the task requests to their TDMA slots. Works assuming dynamic arbiters [24] consider the particular pattern of accesses of each contender to the bus. For on-chip resources, two main approaches have been followed, both requiring some hardware support: isolation or bounded interference. The former uses TDMA arbitration and partitioned caches to prevent interaction among tasks [14]. The latter bounds the maximum impact that one task may generate on co-runners [19]. However, as far as we can tell, such specialized hardware support is not fully or readily available to industry: while cache partitioning has been implemented in hardware, e.g. in the Cobham Gaisler NGMP and the ARM A9, for the bus and the memory controller instead such support is not provided. When the shared cache is not partitioned, alternative solutions – around the concept of *partial time composablity* – have been proposed to approximate the time composability properties provided by templates and signatures [10].

In the absence of hardware support in COTS processors, contention effects can be analyzed, bounding the memory latency (for instance for Intel Core-i7 [12]), or even deriving WCET estimates (for Freescale P4080 [18]). In the latter research, authors use a static timing analysis approach with run-time monitoring of the resource usage that benefits from the knowledge of the workload to be able to derive tight WCET estimates. As a consequence of the limitations in the state of the art for COTS, the execution time of a task becomes dependent on its co-runners, which is a major impediment to incremental development and qualification. This is the challenge we have tackled with our approach based on resource signatures and templates.

6. CONCLUSIONS

We presented a novel approach to studying the contention on the bus and memory controller, building on the concept of RUs and RUl that abstract the resource usage made by the task under analysis and by its contenders. These notions help abstract the interference impact suffered by the task under analysis and the interference effects generated by its contenders. The notions embodied in our proposal provide a simple yet powerful mechanism to aid time-composable integration of multiple tasks in a multicore. A wise selection of RUl allows obtaining tight upper bounds to execution time, for modest cost and effort, thereby facilitating incremental development and qualification for systems targeting COTS multicore processors.

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 NPI Contract 40001102880; 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.

7. REFERENCES

[1] *NGMP Preliminary Datasheet Version 2.1, May 2013.*
[2] A. Schranzhofer et al. Timing analysis for TDMA arbitration in resource sharing systems. In *RTAS*, 2010.
[3] Aeroflex Gaisler. *LEON4-N2X Data Sheet and User's Manual*, 2013.
[4] J. Bin et al. Using monitors to predict co-running safety-critical har real-time benchmark behavior. In *ICITES*, 2014.
[5] S. Chattopadhyay et al. A unified WCET analysis framework for multicore platforms. *ACM Trans. Embed. Comput. Syst.*, 13(4), Apr. 2014.
[6] W. David et al. Dramsim: a memory system simulator. *SIGARCH Comput. Archit. News*, 2005.
[7] G. Fernandez et al. Contention in multicore hardware shared. resources: Understanding of the state of the art. In *WCET Workshop*, 2014.
[8] G. Fernandez et al. Increasing confidence on measurement-based contention bounds for real-time round-robin buses. In *DAC*, 2015.
[9] G. Fernandez et al. Introduction to partial time composability for cots multicores. In *SAC*, 2015.
[10] G. Fernandez et al. Seeking time-composable partitions of tasks for cots multicore processors. In *ISORC*, 2015.
[11] M. Fernández et al. Assessing the suitability of the ngmp multi-core processor in the space domain. EMSOFT, 2012.
[12] K. Hyoseung et al. Bounding memory interference delay in COTS-based multi-core systems. In *RTAS*, 2014.
[13] A. Jung and P.-E. Crouzet. The h2rg infrared detector: introduction and results of data processing on different platforms. 2012.
[14] T. Kelter et al. Bus-aware multicore WCET analysis through TDMA offset bounds. *Real-Time Systems, Euromicro Conference on*, 2011.
[15] Kingston. KVR667D2S5/2G Datasheet, 2011.
[16] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium*, 1999.
[17] J. Nowotsch et al. Leveraging multi-core computing architectures in avionics. In *EDCC*, pages 132–143. IEEE, 2012.
[18] J. Nowotsch et al. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS*, 2014.
[19] M. Paolieri et al. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009.
[20] R. Pellizzoni et al. Worst case delay analysis for memory interference in multicore systems. In *DATE*, 2010.
[21] J. Poovey. *Characterization of the EEMBC Benchmark Suite.* North Carolina State University, 2007.
[22] P. Radojković et al. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM TACO*, 2012.
[23] RapiTime. *www.rapitasystems.com.*
[24] S. Schliecker et al. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *DATE*, 2010.
[25] H. Shah et al. Measurement based wcet analysis for multi-core architectures. RTNS, 2014.