Bounding Resource Contention Interference in the Next-Generation Microprocessor (NGMP)

Javier Jalle[†,*], Mikel Fernandez[†], Jaume Abella[†], Jan Andersson[⋆],
Mathieu Patte[∓], Luca Fossati[§], Marco Zulianello[§], Francisco J. Cazorla[†,‡]

[†]Barcelona Supercomputing Center, Spain
[*]Universitat Politècnica de Catalunya, Spain
[§]European Space Agency, The Netherlands
[⋆]Cobham Gaisler, Sweden
[∓]Airbus Defence and Space, France
[‡]Spanish National Research Council (IIIA-CSIC), Spain

***Abstract*—The Space industry, as several other real-time industries, is assessing the use of multicore processors as their main computing platform. While multicore processors bring the potential of integrating several software (mixed-criticality) functions, their use also brings some challenges. In particular, tasks running in multicores may experience high contention delays when accessing multicores' shared resources. This makes that the load that a task puts on shared resources impacts the Execution Time Bounds[1] (ETBs) derived for other corunning tasks. In this paper we focus on the Cobham Gaisler NGMP – acknowledged as one of the multicore processors currently assessed by the European Space Agency for its future missions – for which we propose a measurement-based approach to bound contention interference. Given a task τ, instead of providing ETBs for the highest contention that any set of corunners can generate – already shown to be potentially high – our approach provides bounds that factor in the number of requests contenders generate regardless of how they align with τ's requests. This provides a good balance between ETBs accuracy and independence from the corunners, since our approach only requires controlling the number of requests each task makes to the shared resources.**

***Index Terms*—WCET, multicore, COTS, real-time**

I. Introduction

Real-Time Embedded systems are facing an increase in their performance demands across several domains, such as space, avionics and automotive, as a way to provide more value-added functionality. In the space domain, computing power requirements and the amount of data to be handled by on-board software is rising [30] due to the fact that space missions are becoming more autonomous. In this context, multicore processors can provide the performance required, while enabling the consolidation of applications[2] subject to different criticality levels, resulting in an overall reduction in power, space and weight. In the space domain the Next Generation Microprocessor (NGMP) architecture [5], whose latest implementation is the GR740 [6], is an architecture considered by the European Space Agency (ESA) for its future missions.

Multicores also bring their specific issues to the real-time domain among which contention in the access to hardware shared resources is one of the most prominent [2]. Uncontrolled contention makes that the execution time and Execution Time Bounds (ETB) derived for a task depend on the load its corunner tasks put on hardware shared resources, thus affecting time composability [21], which states that the ETB derived for a task in isolation should not be affected by the rest of the tasks running on the system. Time composability is a premise in many real-time designs since it enables (in the timing domain) incremental development and verification in integrated systems such as Integrated Modular Avionics, IMA [26] in avionics. During system development, time composability enables incrementally integrating applications without the need of regression tests to validate the timing properties of already-integrated applications, which heavily reduces integration costs. During operation, time composability enables updating functions and their associated software without the need for re-analyzing and re-qualifying the system. This is specially beneficial in domains like space where systems may operate during dozens of years and whose functionality is usually updated after deployment.

In a time-anomaly [24] free system, time composability can be achieved by modeling at analysis time a scenario in which each access that the task under analysis (τ) makes to a hardware shared resource suffers the highest contention possible. For instance, in the case of a round-robin bus accessed by Nc cores this is equivalent to assuming that each request suffers maximum contention from each of the remaining $Nc-1$ cores. That is, the single-access maximum (contention) delay, or $samd$, corresponds to:

$$samd = (N_c - 1) \times L_{bus} \quad (1)$$

where L_{bus} is the maximum bus latency for a single request. The resulting ETB estimate in this scenario is *fully time composable* since it accounts for the maximum load that corunner tasks of τ can put (at operation time) on the target resource. This, though, comes at the cost of inflated ETB

[1]We use Execution Time Bound (ETB) instead of Worst-Case Execution Time (WCET) estimate to refer to the upper-limits derived for tasks execution time in multicore. The reason is that WCET estimates, as they are commonly understood, establish a single value that upperbounds program's execution time under any circumstance. While this can be asserted for single-core simple architectures, this is not the case for multicores using more complex pipelines.

[2]In this paper we use the terms application and task interchangeably.

estimates (e.g. up to more than 5x times in a 4-core processor as reported in [12]). Tighter ETB estimates can be obtained by adjusting the bounds to the actual load that corunner tasks put on the target resource, which can be abstracted with an arrival curve [27]. However, time-composability is lost since the ETB for a task becomes dependent on its particular corunners. This confronts industry with the choice of time-composable inflated estimates or tighter non time-composable estimates.

In this paper we use measurement-based timing analysis, which a large fraction of safety-related systems resort on [31] – including the space industry. We propose a contention-prediction model that captures the effect of contention in the NGMP shared resources. For a given task, τ, our model enables deriving both fully time-composable bounds to the contention delay suffered by τ or partially time-composable bounds [11] *which depend on the number of requests generated by τ's corunner tasks, N_{req}, but not on how they align with τ's requests*. Derived bounds are valid for different corunner tasks as long as they generate at most N_{req} requests.

Our approach is motivated by the fact that, while the number of requests that a task generates can be bound with existing tools like Rapita System's Verification Suite (RVS) [23], how τ's and its corunners' requests interleave is hard, if at all possible, to measure and control. Hence, instead of predicting request interleaving, our approach derives contention delays for the worst-possible time-alignment of requests. The main contributions of this paper are as follows:

1) We make an in-depth analysis of the hardware shared resources in the NGMP, the way in which requests interact and the delay they may suffer on those resources.

2) We present a prediction model for the contention delay in the bus and the memory controller in the NGMP. Our model, which depends on the time requests take to access shared resources, deals with the case when there are several types of accesses to a resource and each type causes and suffers a different delay depending on the contending accesses. For instance, in the processor AMBA AHB bus, loads missing in the L2 take shorter than loads hitting in L2. We show how our model handles this case.

3) We evaluate our proposal in a solid setup comprising the GR740 implemented in a FPGA. Our proposal provides tighter ETBs than the fully time-composable proposal in [12], since it adapts to the contenders' load on shared resources in a still partially time-composable and friendly way.

The rest of this paper is organized as follows: Section II presents the related work. Section III provides information on the GR740. Section IV details our prediction model. Section V assesses the accuracy of our model. Finally, Section VI presents the main conclusions of this paper.

II. Related Work

Contention on the 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 [8]. Several techniques propose means to upper-bound, during the analysis phase of the system, the $samd$ that a task may suffer on the bus or in memory. In that line, hardware support has been proposed (though not yet implemented in any architecture we are aware of) to artificially delay each request a given τ does by $samd$ cycles [18][13][28]. Other approaches derive $samd$ by using a software-only approach: τ is run against a set of resource stressing kernels that put high load on the resource [12][9] making τ's requests suffer high contention delays.

Other techniques like those in [25] for buses rely on detailed information about resource access latencies and arbitration policies to derive $samd$. Other works, due to lack of information in the processor documentation derive $samd$ from measurements and feed it into static timing analysis. In particular [17] applies this approach to analyze the impact of contention in the P4080. $samd$ can also be derived for memory with [19][1] or without [15] hardware support.

In this paper we follow the theoretical approach in [10] that proposes a methodology to obtain the resource access 'profile' of a given task that defines the use of resources that the task makes on a target shared resource. That profile is used to derive the contention tasks suffer and generate when accessing that resource. In this work, which is a collaboration of end-users in the Space domain (Airbus Defence and Space and ESA), hardware technology providers (Cobham Gaisler) and a research institution (Barcelona Supercomputing Center) we assess the benefits of such an approach on a real platform, the GR740 addressing issues related to NGMP specific arbitration policies and access types to the different resources.

Finally, is it worth mentioning that with few exceptions [29][3], cache partitioning is the common solution in the context of CRTES due to the complexity of estimating ETB accurately on top of shared caches. In the case of the GR740, hardware support exists for way-partitioning the L2 cache. We enable this hardware feature in our experiments.

III. NGMP

The NGMP is being assessed as the processing platform by the ESA in its future missions. The NGMP is a quad-core processor system-on-chip based on the LEON4 SPARC V8 architecture [5] connected by a shared on-chip AHB processor bus to a shared L2 cache and memory, see Figure 1. The NGMP comprises 16 Performance Monitoring Counters (PMC) that can be configured with different events, providing support to measure access counts such in a way that it facilitates the implementation of our prediction model, more details are provided in Section IV-C. This section provides details on some aspects related to the contention in the access to NGMP's shared resources.

A. AHB Processor Bus

The AMBA AHB bus connects cores to the L2 cache and the I/O bridges[3]. The first consideration to make in the case of the bus is that there are different types of requests that can generate different inter-task contention: bus reads (loads) that

[3]In this work, we do not consider I/O related activities, which we assume managed at software level, so that only accesses to L2 interfere each other.

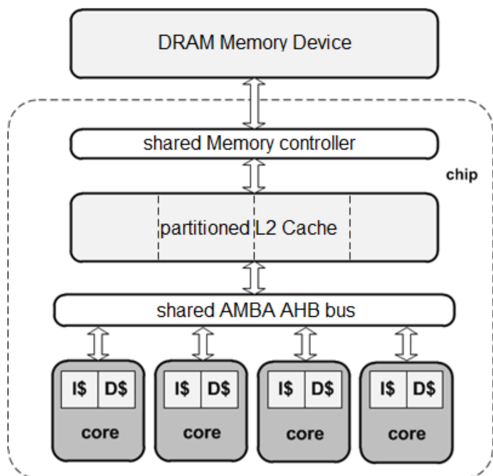


Fig. 1: Block diagram of the main elements of the NGMP

either hit (*l2h*) or miss (*l2m*) on the L2 cache and bus writes (stores) that either hit (*s2h*) or miss (*s2m*) on the L2 cache. These accesses behave differently because hits hold the bus while they are served. Instead, misses wait on a miss queue and are split, i.e. the L2 cache releases the bus while processing the miss, so that other cores can use the bus. In the NGMP, the AMBA AHB bus implements round-robin arbitration.

B. L2 Cache

In our experiments we use the master-index feature of the NGMP that partitions the L2 assigning one L2 cache way to each core. Hence, a given core suffers no contention interference in the L2 due to other cores' evictions.

Each of the request types identified before (*l2h*, *l2m*, *s2h* and *s2m*) has its own L2 access latency. Interestingly, the latency of requests of the same type can be variable. That is, for each request type access there is a Best-Case (BC) and a Worst-Case (WC) latency. This jitter is caused by the type of previous requests, despite they belong to a different task and hence go to a different cache partition. Our model takes this effect into account by assuming that all latencies suffered on the experiments have the BC and when computing the contention bounds, we add a correcting value that adds for each L2 access the corresponding difference between the WC and the BC. This adds pessimism but its advantage is two-fold: it is a safe upperbound and it removes the need to track the sequence of accesses to determine their exact latency.

The WC and BC latencies are obtained from table 40 in [7] and are 8, 13, 6 and 7 for *l2h*, *l2m*, *s2h* and *s2m* respectively in WC and 5, 6, 0 and 0 for BC.

C. Memory Controller

The memory controller acts as an interface between the processor and the DRAM memory. We differentiate two types of request in the memory: read and write. According to the DRAM protocol, each request has a latency to be responded depending on whether it is a read or write request respectively. The latency it takes the memory to go back into idle state, once a request starts being processed, is fixed regardless of whether the request is read or write and corresponds to the time till a new request can be processed. For this paper, we assume that the memory controller behaves as a FIFO queue. This is a simplification that helps upper bounding the memory controller latency though it introduces some pessimism. Providing a more accurate model of the memory is part of our future work.

IV. Prediction Models

Our prediction models use measurement-based timing analysis techniques to derive a multicore ETB (ETB_{mc}) for a task τ_i, given its ETB in isolation (ETB_{isol}). To that end, the models predict the total effect of contention in the access to the multicore hardware shared resources, called Contention Delay Bound (CDB), and add it to the ETB in isolation:

$$ETB_{mc} = ETB_{isol} + CDB \tag{2}$$

In order to derive CDB, we add the contribution of each hardware shared resource r, CDB_r:

$$CDB = \sum_{r \in R} CDB_r \tag{3}$$

To derive CDB_r, we upper-bound the maximum latency that every access from τ_i to r, n_i^r, may suffer from requests generated by τ_i's corunner tasks, referred to as $c(\tau_i)$.

CDB_r for τ_i assumes that each $\tau_j \in c(\tau_i)$ performs at most a given number of accesses (n_j^r) to resource r. Therefore, ETB_{mc} estimate for τ_i is composable with any other task $\tau_j' \in c'(\tau_i)$ as long as it performs fewer accesses ($n_j'^r$) to the shared resource than $\tau_j \in c(\tau_i)$:

$$n_j'^r \leq n_j^r \tag{4}$$

A. Bus Prediction Model

The NGMP comprises three main shared resources in its data path: the bus, the L2 cache and memory. Since the L2 can be partitioned we do not consider contention of the different tasks in the L2. We start by predicting CDB_{bus} for the bus and later apply the same approach for memory.

We explain three different ways of upper-bounding CDB_{bus}, which present represent different trade-offs between information required, such as the number of accesses of each corunner task, and tightness of the produced bound.

A.1. Theoretical Upper-Bound Delay (UBD)

In this reference model, based on [18], we assume that every single τ_i request is delayed by a request from each of the $N_c - 1$ contenders and that contending requests cause the highest delay, L_{bus}. This is the maximum contention scenario in round-robin arbitration, where the upper-bound delay a request can suffer is given by:

$$samd = (N_c - 1) \times L_{bus} \tag{5}$$

Hence, for τ_i with b_i accesses to the bus, CDB_{bus} is presented in Equation 6, where L_{bus} is the maximum delay any interfered request can suffer from a single interfering request.

$$CDB_{bus} = b_i \times samd = b_i \times (N_c - 1) \times L_{bus} \quad (6)$$

Since we have four different types of requests with different latencies: l_{l2h}, l_{l2m}, l_{s2h} and l_{s2m}:

$$L_{bus} = \max\left(l_{l2h}, l_{l2m}, l_{s2h}, l_{s2m}\right) \quad (7)$$

This model is time-composable by definition because it assumes that all b_i are interfered by i) the highest impact request from ii) all corunners. These two assumptions are sources of pessimism that enable full time-composability.

Interestingly in this model, the worst alignment among the requests of τ_i and the requests of its corunners is assumed. In reality, it can be the case that some τ_i requests become ready to be sent to the bus when its contenders requests have been partially processed so that each τ_i request suffers a delay smaller than L_{bus}. However, predicting how this alignment of requests can happen at operation time is hard (if at all possible). Any small shift in the execution of tasks can change it. Hence, this and the following models, provision time in CDB_{bus} for the worst-case alignment of requests.

A.2. Single-type Model

Analogously to the previous model, the one presented in this section assumes that every corunners' request causes a delay of L_{bus} on τ_i. Unlike the previous model, this one takes into account that not all τ_i requests might be interfered by one request of its corunner tasks. This usually happens when corunner tasks have fewer accesses than τ_i.

Let b_j be the number of accesses to the bus that each contender task $\tau_j \in c(\tau_i)$ performs. Given that tasks have different number of accesses, not all of them can interfere each other. In particular, for a given interfering task τ_j running in core j, *in the worst-case* only the minimum between the number of accesses of τ_i, b_i, and the number of accesses of the interfering task, b_j, suffer a contention delay of L_{bus}. That is, no more than b_i accesses can be interfered and no more than b_j can interfere. In order to compute the contention on the bus for task τ_i, we add the contribution of each interfering task τ_j:

$$CDB_{bus} = \sum_{\tau_j \in c(\tau_i)} \min\left(b_i, b_j\right) \times L_{bus} \quad (8)$$

A.3. Multiple-type Model

The previous model assumes that each interfering request, i.e. those generated by $c(\tau_i)$, belongs to the worst-interfering type, hence generating L_{bus} delay on τ_i. However, corunner tasks generate requests of different types, each of which incurs a different interference on τ_i. This model takes this into account and breaks down the number of requests of the corunners between *l2h*, *l2m*, *s2h* and *s2m*:

$$b_j = b_j^{l2h} + b_j^{l2m} + b_j^{s2h} + b_j^{s2m} \quad (9)$$

The order of these requests, from most interfering to less interfering is, *l2h*, *l2m*, *s2h* and *s2m* (see Section IV-C).

To compute the CDB_{bus}, we pair each interfered request (those coming from τ_i) with the worst eligible interfering request available from each contending core. We start pairing the accesses with the most interfering type (*l2h*) until this interfering type is consumed. The remaining $b_i' = max(0, b_i - b_j^{l2h})$ requests from τ_i are paired with the next interfering type (*l2m*). The remaining $b_i'' = max(0, b_i' - b_j^{l2m})$ with *s2h* and finally the remaining $b_i''' = max(0, b_i'' - b_j^{s2h})$ with *s2m*. With this CDB_{bus} is computed as follows:

$$\begin{aligned} CDB_{bus} = \textstyle\sum_{\tau_j \in c(\tau_i)} [\; & \min\left(b_i, b_j^{l2h}\right) \times l_{l2h} + \\ & \min\left(b_i', b_j^{l2m}\right) \times l_{l2m} + \\ & \min\left(b_i'', b_j^{s2h}\right) \times l_{s2h} + \\ & \min\left(b_i''', b_j^{s2m}\right) \times l_{s2m}] \end{aligned} \quad (10)$$

It is worth noting that the type of the requests generated by τ_i are equally affected by each type of request of its corunner. That is, the interference is determined by the type of the request of the corunner task τ_j only.

B. Memory Prediction Model

To compute CDB_{mem} we apply the same models as for the bus. As explained in Section III, there are two different types of request in the memory, read and write. We assume a task τ_i with m_i requests to the memory and contender tasks $\tau_j \in c(\tau_i)$ with $m_j = m_j^{read} + m_j^{write}$ accesses to the memory each and $m_i' = max(0, m_i - m_j^{read})$. The highest delay in memory is given by $L_{mem} = \max\left(l_{read}, l_{write}\right)$.

Under these constraints the theoretical Upper-Bound Delay model is given by:

$$CDB_{mem} = m_i \times samd = m_i \times (N_c - 1) \times L_{mem} \quad (11)$$

The model based on single request types is as follows:

$$CDB_{mem} = \sum_{\tau_j \in c(\tau_i)} \min\left(m_i, m_j\right) \times L_{mem} \quad (12)$$

The model based on multiple request types is as follows:

$$\begin{aligned} CDB_{mem} = \sum_{\tau_j \in c(\tau_i)} & \min\left(m_i, m_j^{read}\right) \times l_{read} + \\ & \min\left(m_i', m_j^{write}\right) \times l_{write} \end{aligned} \quad (13)$$

From previous discussions it follows that our model builds on two pieces of information: the latency each request uses each shared resource and the number of accesses performed by each task to each shared resource. We describe both in the following subsections.

C. Deriving Access Latencies

Bus. Our model uses as an input the time each request uses each shared resource, which correspond to the bus and memory in our reference architecture. For the bus, in the NGMP, our model requires deriving the bus usage latency of *l2h*, *l2m*, *s2h*

and *s2m*[4]. Since documentation typically does not provide this information, we derived it empirically.

To do so, first we executed a benchmark performing a given type of bus operations as the Task Under Analysis (*tua*), or interfered task; against a range of other benchmarks, or corunner tasks, performing all of them the same type of accesses (which may be a different type as for the *tua*). For instance, in one experiment the *tua* performs *l2h* accesses and the corunner tasks *s2h* accesses.

As a result of performing this process we reached the following three observations:

1) The execution time of the *tua* depends on the type of accesses performed by the corunner tasks. Thus, given a *tua*, its execution time may not be the same if corunners perform *l2h*, *l2m*, *s2h* or *s2m* accesses.
2) The impact of corunner tasks on the *tua* is linear with the number of corunners. Therefore, if the execution time of the *tua* in isolation (normalized) is 1, and it grows to $1+K$ when running against one corunner, then the execution time against C corunners can be upper bounded as $1 + C \times K$ (further considerations on this matter can be found in [9]). In the particular case where corunner tasks are run in all other cores, the execution time of the *tua* is:
$$1 + (N_c - 1) \times K \tag{14}$$
3) The impact of interferences in the execution time of the *tua* is independent of the particular access type performed by the *tua*. Therefore, the execution time in isolation grows by $(N_c - 1) \times K$ when all corunners perform the same type of accesses (i.e. *l2m*), but K depends solely on the type of accesses of the corunners, not on the type of accesses of the *tua*. This can be explained because the interference on the AMBA AHB bus depends only on the arbitration time [14], which in fact depends only on the time the higher priority corunners use the bus and not on the interfered request which is requesting the bus and has to wait the same amount of time regardless its particular access type.

To infer the latencies we take as a reference the *l2h* benchmark that constantly accesses the L2 cache and hence the bus. Further since the benchmark always hits in L2, each request on the bus has a short turn-around time. This benchmark is executed as *tua* in a workload comprising 3 corunner benchmarks, which correspond to the 3 remaining cores. The corunners perform accesses of the same type to the bus continuously. Hence, there are 4 different workloads depending on the type of access performed by the other three corunners: *l2h*, *l2m*, *s2h*, *s2m*.

Figure 2 shows the measured execution time for all workloads. To infer the bus latencies, we divide the execution time overhead of the *tua* with respect to the execution time in isolation by the amount of contenders (3 in each case) and then

[4]Please note that these latencies are not the same as those obtained in Section III-B for the L2 cache.

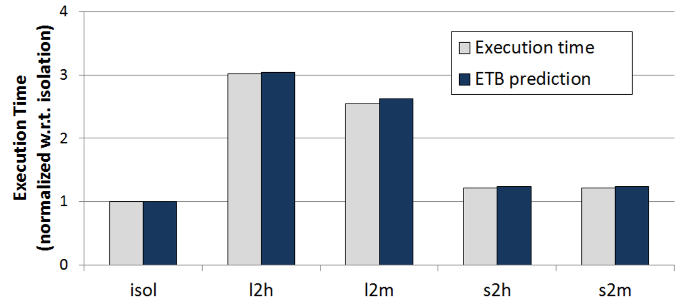


Fig. 2: Execution time and ETB of *l2h* benchmark in different workloads

divide these cycles by the amount of bus accesses performed by each contender. For instance, given an execution time of T_{isol} for the *tua* in isolation and T_{l2h} for the *tua* against 3 *l2h* corunners, the interference of an *l2h* access is obtained as follows where N_{req} is the number of *l2h* requests performed by each corunner:

$$l_{l2h} = \left\lceil \frac{T_{l2h} - T_{isol}}{(N_c - 1) \times N_{req}} \right\rceil \tag{15}$$

This way we obtain the number of interference cycles per bus access type: 9, 7, 1 and 1 for l_{l2h}, l_{l2m}, l_{s2h} and l_{s2m} respectively[4]. With these latencies we compute CDB_{bus} with Equation 6 and build the ETB prediction shown in Figure 2, which is computed using Equation 2.

Techniques to improve the confidence on derived bus latencies are proposed in [9]. Part of our future work consists of integrating those methods on top of our model and compare them against our method to derive latencies. Nevertheless, our prediction models are compatible with any method to obtain the access latencies.

Memory. The approach followed to obtain memory latencies is analogous to that for bus latencies with some small differences. First, instead of using benchmarks accessing the bus, we use *l2m* as *tua*, which performs memory reads. As corunner tasks we use first 3 copies of a *l2m* benchmark, that generates memory reads. The latency of memory reads obtained in this case is 18 cycles. In the second experiment we use 3 copies of *s2m* as corunners. The latency of memory writes obtained is again 18 cycles because there is no difference between read and write operations in terms of memory interference since, in both cases, the timing is defined by the time to open and close the memory page or row, which is identical for both.

D. Deriving Access Counts

The NGMP provides 16 PMCs that can be configured with different events and can be measured using the commercially available tool GRMON2 [4]. Among other events, we are interested in the per-core bus reads and writes (0x40 - 0x50 in [5]) and per-core L2 hits and misses (0x60 - 0x61).

Bus. The total number of L2 accesses (i.e. hits and misses) corresponds to the number of bus accesses. However, there is no way to break down L2 hits/misses into reads and writes,

i.e. it is not possible to determine exactly the number of l2h, l2m, s2h and s2m accesses.

In this scenario our approach is to estimate those values in the most pessimistic way: Given task τ_j, we can obtain the number of L2 hits and misses, b_j^h and b_j^m, and the number of bus read and writes, which is equivalent to the number of L2 loads and stores, b_j^l and b_j^s. Our goal is to distribute b_j^h, b_j^m, b_j^l and b_j^s into b_j^{l2h}, b_j^{l2m}, b_j^{s2h} and b_j^{s2m} such that their total impact is maximized. For the l_{l2h}, l_{l2m}, l_{s2h} and l_{s2m} latencies in our reference architecture, the following equations maximize the impact. First, we assume the maximum amount of requests from the worst possible interfering request, i.e. l2h:

$$b_j^{l2h} = \min(b_j^l, b_j^h) \tag{16}$$

Then we subtract this value from b_j^l and b_j^h, obtaining $b_j'^l = max(0, b_j^l - b_j^{l2h})$ and $b_j'^h = \max(0, b_j^h - b_j^{l2h})$, and repeat the algorithm with the next worst interferers, i.e. l2m, s2h and then s2m, with $b_j'^s = max(0, b_j^s - b_j^{s2h})$ and $b_j'^m = \max(0, b_j^m - b_j^{l2m})$, to obtain b_j^{l2m}, b_j^{s2h} and b_j^{s2m}.

$$b_j^{l2m} = \min(b_j'^l, b_j^m) \tag{17}$$

$$b_j^{s2h} = \min(b_j^s, b_j'^h) \tag{18}$$

$$b_j^{s2m} = \min(b_j'^s, b_j'^m) \tag{19}$$

Once we have all accesses properly classified as l2h, l2m, s2h and s2h for each contending task τ_j, we can proceed with the model described before.

Memory. Our current implementation does not provide access counters for the memory controller. Hence, the exact number of memory accesses cannot be obtained, even though L2 cache misses are known, since there is no way of accounting indirect memory accesses such as writes generated by evictions of dirty L2 lines. The actual number of memory accesses can either be estimated using the number of L2 misses, which is a lower bound of the memory accesses or estimated with the number of L2 misses and the number of bus writes, which is an upper bound.

E. Assumptions

Our model is based on the assumption that the number of accesses of a task is not affected by the contenders. This happens only if the L2 is partitioned, i.e. not shared. Otherwise, the number of accesses to the bus or memory for a task executed in isolation does not match those obtained when running along with other tasks.

Also our model assumes that no timing anomalies [24] are present. Timing anomalies is an open research field, and is difficult to prove that a real processor is time anomaly free [16]. Nevertheless, if timing anomalies can occur, they cannot trigger a domino effect by construction, i.e. it is a *compositional architecture with constant-bounded effects* according to the classification in [32]. In those architectures, which may experience timing anomalies but no domino effects, the impact of timing anomalies can be accounted for easily by counting how many times they can be triggered and padding ETBs by the product of this count and the maximum impact of one timing anomaly. This approach is fully in line with our prediction model that accounts for contention in shared resources. Further, note that our model has no impact on timing anomalies, which occur (or not) regardless of our model.

V. Experimental Results

We evaluate our proposals on a real GR740 [6] FPGA prototype on a Xilinx ML510 board. We used the commercially available Cobham Gaisler GRMON2 [4] debug monitor software to directly extract the PMC from the statistic unit of the GR740, without affecting execution. The model is directly constructed from the readings obtained in one execution of each task, i.e. no further post processing is required.

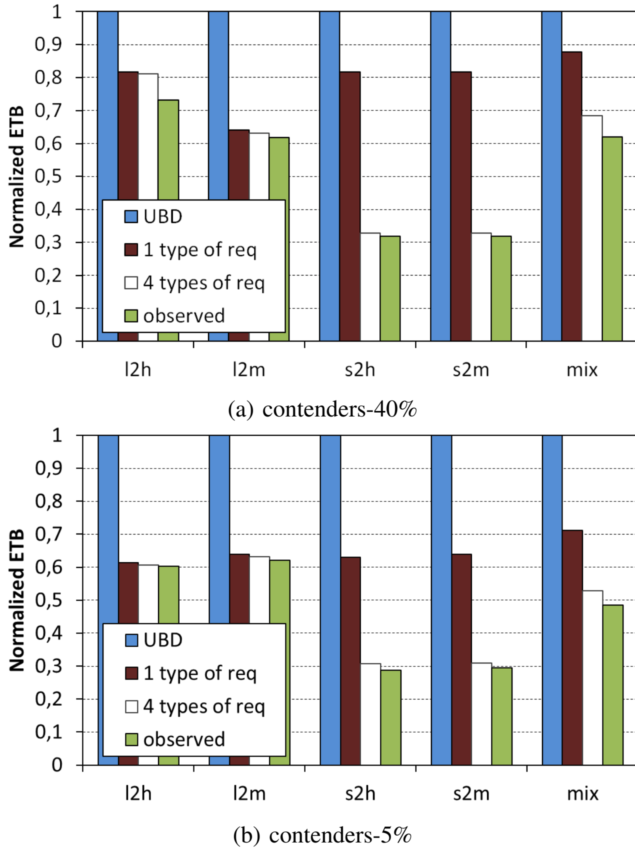
A. Bus and memory prediction models

Our first experiments put the shared resources under high pressure to test the tightness of the bounds obtained with the prediction model. To that end we use as reference applications a set of synthetic kernels [12] that inject constant high pressure either on the shared bus or on the shared memory. The Bus-Stressing Kernel, or *bsk*, comprises memory read and write requests that always miss the L1 and hit the L2, thus maximizing the traffic on the bus. This is done by having 5 memory accesses that access the same set of the L1 cache, thus exceeding its 4 ways. The same approach is used for the Memory-Stressing Kernel (*msk*) that comprises memory read requests that always miss on the L1 and also miss on the L2.

In all experiments we use one reference task (*tua*) and three tasks as corunners. In particular, as reference task on which ETB is to be derived we use bsk-ld-40% in which 40% of its instructions are loads that access the bus. As corunner tasks we use *bsk* whose frequency of access is 40% and 5%, i.e. 40% and 5% of the instructions are accesses to the bus. Those *bsk* used as corunners access the bus with requests of a different type across experiments: *l2h*, *l2m*, *s2h*, *s2m* and a *mix*, which consist of a *l2h*, a *l2m* and a *s2h bsk* together.

Figure 3a and Figure 3b show the result for bsk-ld-40% when the frequency of access of the contenders is 40% and 5% respectively. In both figures we show the ETB when using the UBD approach [12] or our approach with a single-type and four types of requests; and the observed execution time. In all cases, the predicted ETB estimates are above the observed execution time. The UBD model, since it assumes that every access of the task under analysis suffers $samd$, leads to the highest ETBs. Our model, that accounts only for the contention the task under analysis suffers, tightens the ETB. As presented in the previous sections, if the method is made aware of the request types and their associated latency (4 types of request) ETBs are further reduced.

In Figure 3b, the corunners make fewer accesses than in Figure 3a. For the UBD approach this has no impact since it only focuses on the number of requests of the task under analysis that remains the same. Instead, our models reduce ETBs since they effectively capture the fact that the corunners make fewer accesses.



(a) contenders-40%

(b) contenders-5%

Fig. 3: ETB for each bus prediction model: UBD (fully time composable requests); And our approach with 1 and 4 request types.

We performed the same experiment for the memory model, using msk-ld-40% as *tua* and corunners with 40% and 5% of memory accesses. In this case, the single type of request or multiple types of request models are equivalent, since read and writes to memory have exactly the same impact. The results are analogous to those obtained for the bus model. Therefore, we omitted the figures since they provide no further insights.

B. EEMBCs

As final evaluation we apply the whole prediction model with the EEMBC Autobench suite [20] as reference applications. We run each EEMBC benchmark under a relatively high pressure scenario composed of two tasks, one continuously accessing the bus (*bsk*) and the second accessing the memory (*msk*). In this scenario, neither the bus nor the memory controller suffer the highest pressure, since that requires all remaining cores accessing simultaneously each resource [22].

No memory accesses. As presented in Section IV-D, there is not a specific PMC to measure the number of accesses to the memory. In order to cancel out the impact of this, in a first experiment we focus on the case in which the *tua* is run twice in a row and measurements are taken during the second run. Due to the small footprint of EEMBC Autobench, this results in almost zero misses in the second run.

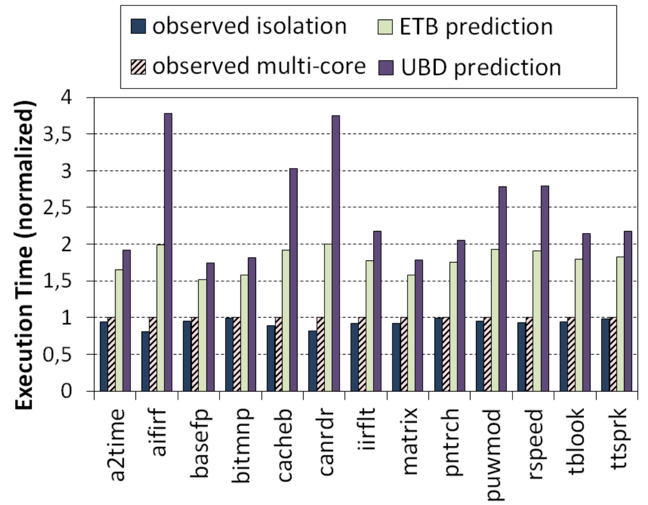


Fig. 4: ETBs for EEMBC when assuming a no cache misses.

Under that assumption, we present the results of our model and UBD in Figure 4. Recall that in each workload we run one EEMBC benchmark executed and the two contenders presented above. Results are normalized w.r.t. the execution time of the EEMBC in the workload. We show the execution time in isolation, the execution time in the workload and the predicted ETB using the multiple type of requests prediction model for the bus and memory, as well as the UBD. We can clearly see that our prediction model reduces the pessimism of the UBD model by 67% being only 79% higher than the actual execution time. In Figure 3, the observed execution time is much closer to the prediction when compared with Figure 4. This is because the scenario in Figure 3 is designed to experience severe contention, whereas the scenario described here experiences much lower contention (far below the upper-bounded contention).

General case. Our next step is to evaluate the natural case in which programs perform memory accesses. According to Section IV-D, we can estimate the access to the memory using the number of L2 misses. The number of L2 misses does not consider the dirty evictions that generate memory accesses. To take into account the dirty evictions into the memory accesses we can use either an optimistic lower bound based on the number of L2 misses or a pessimistic upper bound based on the number of L2 misses plus the bus writes. To that end, we build three scenarios:

- Pessimistic scenario. We assume that every write operation results in a dirty eviction, i.e. an access to memory.
- Accurate scenario. In this case, from a simulation tool we derive the exact number of memory accesses.
- Optimistic scenario. We disregard the dirty evictions and take the number of L2 misses as memory accesses.

Figure 5 shows the results obtained with our model under each of the previous scenarios and the observed execution time in the workload and isolation. These results provide a good estimate of the benefits of improving our reference design with a PMC that explicitly measures memory accesses.

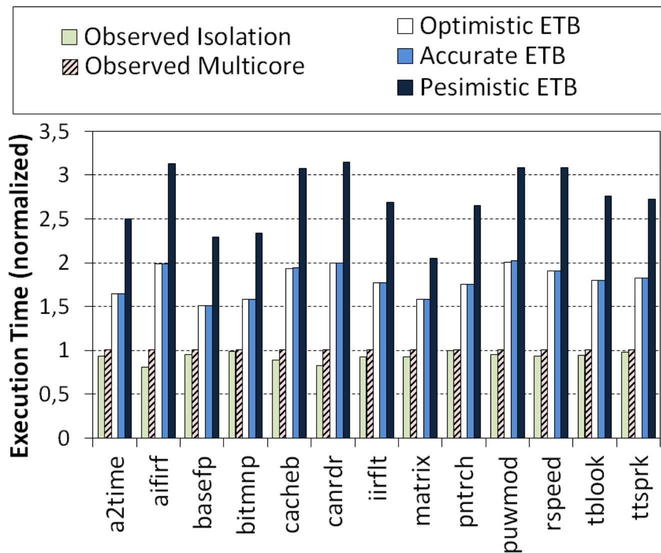


Fig. 5: ETBs for EEMBC under the optimistic, pessimistic and accurate scenarios.

As expected the pessimistic scenario that considers all writes as dirty evictions is overly pessimistic. In particular it is 138% more pessimistic than the actual observed execution time. The accurate scenario in which we assume that the PMC for access count exists leads to very tight estimates, 64% more pessimistic than the actual observed execution time and less than 0.1% more pessimistic than the optimistic scenario. This is due to the small memory footprint of the EEMBC benchmark, that fit on the L2 cache. As a result, the number of dirty evictions is close to zero in most scenarios.

VI. Conclusions

In this paper we present a prediction model of the shared resource contention for the GR740 that takes into account the number of accesses and their type for a given task and its corunner tasks, which can be easily obtained with PMCs. The model abstracts (i.e. makes worst-case provisions) for the way in which requests interleave in time, which would challenge time composability since such time interleaving could easily change during operation.

Derived Execution Time Bounds (ETBs) are shown to be accurate and tighter than fully-time composable ETBs. Those derived estimates are valid for any workload in which the task runs as long as the number of accesses (per type) is smaller than those assumed at analysis. This provides a good balance between tightness and time composability.

VII. Acknowledgements

The research leading to this work has received funding from: the European Space Agency under contracts 4000109680, 4000110157 and NPI 4000102880. 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] B. Akesson et al. Predator: a predictable SDRAM memory controller. In *CODES+ISSS*, 2007.

[2] F. J. Cazorla et al. Multicore OS benchmarks. Technical report, European Space Agency, 2012.

[3] S. Chattopadhyay et al. A unified WCET analysis framework for multicore platforms. *ACM Trans. Embed. Comput. Syst.*, 13(4), Apr. 2014.

[4] Cobham Gaisler. *GRMON2-User Manual Version 2.0.62, March 2015*.

[5] Cobham Gaisler. *NGMP Preliminary Datasheet Version 2.1, May 2013*.

[6] Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor - GR740-UM-DS-D1 - Data Sheet and Users Manual, 2015*.

[7] Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor - LEON4-N2X Data Sheet and Users Manual Version 2.1*, 2015.

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

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

[10] G. Fernandez et al. Resource usage templates and signatures for COTS multicore processors. In *DAC*, 2015.

[11] G. Fernandez et al. Seeking timecomposable partitions of tasks for cots multicore processors. In *ISORC*, 2015.

[12] M. Fernandez et al. Assessing the suitability of the NGMP multi-core processor in the space domain. In *EMSOFT*, 2012.

[13] J. Jalle et al. Deconstructing bus access control policies for real-time multicores. In *SIES*, 2013.

[14] J. Jalle et al. AHRB: A high-performance time-composable amba ahb bus. In *RTAS*, 2014.

[15] H. Kim et al. Bounding memory interference delay in cots-based multi-core systems. In *RTAS*, 2014.

[16] T. Lundqvist et al. Timing anomalies in dynamically scheduled micro-processors. In *Real-Time Systems Symposium*, 1999.

[17] J. Nowotsch et al. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS*, 2014.

[18] M. Paolieri et al. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009.

[19] M. Paolieri et al. Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions. *ACM Trans. Embed. Comput. Syst.*, 12(1s), 2013.

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

[21] P. Puschner et al. Towards composable timing for real-time software. In *1st International Workshop on Software Technologies for Future Dependable Distributed Systems*, 2009.

[22] P. Radojković et al. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 2012.

[23] Rapita systems Ltd. Rapita Verification Suite. http://www.rapitasystems.com/products/rvs.

[24] J. Reineke et al. A definition and classification of timing anomalies. In *WCET*, 2006.

[25] J. Rosen et al. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*, 2007.

[26] RTCA & EUROCAE. DO-297 Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations.

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

[28] H. Shah et al. Measurement based WCET analysis for multi-core architectures. In *RTNS*, 2014.

[29] M. Slijepcevic et al. Time-analysable non-partitioned shared caches for real-time multicore systems. In *DAC*, 2014.

[30] A. West. NASA Study on Flight Software Complexity. Final Report. Technical report, NASA, 2009.

[31] R. Wilhelm et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7:1–53, May 2008.

[32] R. Wilhelm et al. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(7):966–978, July 2009.