Computing Safe Contention Bounds for Multicore Resources with Round-Robin and FIFO Arbitration

Gabriel Fernandez*,†, Javier Jalle*,†, Jaume Abella†, Eduardo Quiñones†,
Tullio Vardanega⋆, Francisco J. Cazorla†,‡
*Universitat Politècnica de Catalunya, †Barcelona Supercomputing Center
⋆University of Padua, Italy, ‡Spanish National Research Council (IIIA-CSIC)

Abstract—Numerous researchers have studied the contention that arises among tasks running in parallel on a multicore processor. Most of those studies seek to derive a tight and sound upper-bound for the worst-case delay with which a processor resource may serve an incoming request, when its access is arbitrated using time-predictable policies such as round-robin or FIFO. We call this value *upper-bound delay* (ubd). Deriving trustworthy ubd statically is possible when sufficient public information exists on the timing latency incurred on access to the resource of interest. Unfortunately however, that is rarely granted for commercial-of-the-shelf (COTS) processors. Therefore, the users resort to measurement observations on the target processor and thus compute a "measured" ubd_m. However, using ubd_m to compute worst-case execution time values for programs running on COTS multicore processors requires qualification on the soundness of the result. In this paper, we present a measurement-based methodology to derive a ubd_m under *round-robin* (RoRo) and *first-in-first-out* (FIFO) arbitration, which accurately approximates ubd from above, without needing latency information from the hardware provider. Experimental results, obtained on multiple processor configurations, demonstrate the robustness of the proposed methodology.

Index Terms—Computers and information processing, Real-time systems, Parallel architectures, Multicore processing

✦

1 Introduction

The real-time systems industry has started to consider multicore processors (multicores in the following) as their baseline computing platform, in response to the increasing performance requirement of new applications. This situation extends across a variety of application domains, including automotive [34], avionics [32], and space [33].

In spite of the potential benefit to available performance, embracing multicores for the real-time systems industry is a difficult challenge. Chip providers are driven by the mainstream market and industrial developers of real-time systems must stay in the mainstream and use COTS solutions in order to contain procurement costs. However, mainstream COTS multicores are designed to improve average performance rather than time predictability, which is an essential ingredient to compute tight and sound *worst-case execution time* (WCET) bounds for real-time software programs. Sadly, at the present state of the art, analysis solutions capable of delivering tight and sound WCET bounds for COTS multicores do not yet exist, and *execution-time bounds* (ETB) are derived instead, which may or may not be true upper-bounds.

One of the challenges of timing analysis for COTS multicores stems from the difficulty of determining the worst-case impact of contention on access to hardware shared resources. In this paper, the term ubd, for *upper-bound delay*, denotes that impact factor. Studies exist that investigate the ubd arising on access to the on-chip bus [10] and the memory controller [18], [20]. Those works however yield a tight and sound ubd estimation only when enough information about the timing behaviour of the target processor is available.

Both the *Static Timing Analysis* (STA) and the *Measurement-Based Timing Analysis* (MBTA) methods [34] need trustworthy ubd to compute sound ETBs. STA uses the ubd to cost every request to a shared hardware resource issued by a software program. MBTA, the most used practice in industry at present, needs to know the ubd to gage the contention delay that may be suffered by application programs.

Unfortunately, as the complexity of multicores continues to rise and information on their internal function is increasingly restricted by intellectual property, the static derivation of ubd becomes inordinately harder. As a testimony to that, the contention behaviour of the P4080 processor has been analyzed by an avionics end-user and an STA tool provider [17] using measurements, thereby obtaining a measured approximation of the ubd [16], here denoted ubd_m.

The net consequence of that difficulty is that the confidence that can be placed on ETB rests on the confidence that can be attached to the ubd_m; in particular, on how well it approximates the actual ubd.

To the best of our knowledge, the state-of-the-art techniques used to compute ubd_m most frequently employ specialized programs executing in the application space, often called *resource stressing kernels* (*rsk*) [7], [16], [23], also referred to as *micro-benchmarks*. The *rsk* approach computes the ubd_m by running the *software component under analysis* (*scua*) against a battery of *rsk*. In particular, the ubd_m is derived by dividing the execution-time increment suffered

© © 20xx IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

by the *scua*, (Δ_{ET}), owing to the contention generated by the *rsk*, by the number n_r of access requests made by the *scua*: $ubd_m = \Delta_{ET}/n_r$. Interestingly, whereas *rsk* are expressly designed to produce high contention on a given shared hardware resource (e.g., the bus) so that the designated victim suffers high slowdown, insufficient attention has been devoted to determining whether the ubd is best approximated using the *scua* or an *rsk* as victim.

We show in this paper that the state-of-the-art *rsk* methodology may fail at producing sound ubd_m values. In particular, we analyse the impact that round-robin (*RoRo*) and first-in-first-out (*FIFO*) arbitration policies, widely used in real-time systems due to their time-predictable traits [10] [19], have on the computation of the ubd_m.

In this context, this paper makes the following contributions:
1) We show that a naïve use of *rsk* neither guarantees that the *scua*'s requests suffer the highest contention (ubd) on accessing the target shared hardware resource, nor helps deriving accurate ubd_m approximations to it. The key reason behind this defect is that, under heavy contention scenarios, *RoRo* and *FIFO* produce a "*synchrony effect*" that causes each request issued by the *scua* to suffer a contention delay that can be systematically inferior to the ubd. We show that the contention delay is determined by the time elapsed since the preceding request was served until the current request is issued. We term this duration *injection time*. This phenomenon manifests itself differently for *RoRo* and *FIFO*.
2) We propose a methodology to derive a trustworthy ubd_m that does not need to know the specific latencies of the target shared hardware resource, and thus works for a wide range of COTS processors. Our approach consists in inferring the ubd_m value by varying the injection time between requests to the shared hardware resource. This is realized by inserting a given number of *nop* instructions in between access requests. As the results obtained vary noticeably depending on whether the arbitration policy is *RoRo* or *FIFO*, we propose different methods to obtain the proper ubd_m for each policy.
3) We demonstrate our methodology to derive trustworthy ubd_m for the bus and memory controller on a multicore setup that matches that of the Cobham Gaisler NGMP processor [8], a 4-core multicore considered by the European Space Agency for future missions, which embeds per-core data and instruction caches connected to the L2 with an AMBA AHB bus. For the sake of completeness, we also test our methodology on a variant of this reference multicore design.

Where measurement observations are the most practical and perhaps the only available means to derive the impact of contention bounds for increasingly complex multicores, our approach provides essential aid to computing a trustworthy ubd_m for the bus and the memory controller, and thus increased trustworthiness of the resulting ETB.

The remainder of this paper is organized as follows. Section 2 introduces the impact that RoRo and FIFO incur from contention on access to hardware shared resources. Section 3 describes our reference processor architecture and the constituents of our measurement-based approach to derive the ubd_m, namely, the *resource stressing kernels*. Section 4 illustrates the synchrony effect that occurs with RoRo and FIFO under heavy load conditions. Section 5 and 6 show how our proposed solutions derives ubd_m for the bus and the memory controller, respectively. Section 7 empirically validates our approaches. Section 8 presents related works. Section 9 draws our conclusions from this study.

2 Contention Analysis for RoRo and FIFO

2.1 Studying the Bus and the Memory Controller

The interconnection network and the memory controller are two of the hardware resources whose sharing in multicores causes most bottlenecks for contending tasks that run in parallel. The determination of the ubd for those resources has already received the attention of researchers, under the hypothesis that public documentation on the internal functioning of the processor exists.

- Bus-based interconnection networks are known to require little energy as well as to ease protocol design and verification, while incurring an acceptable slowdown [25], [31]. The Advanced Microcontroller Bus Architecture (AMBA) is a bus exemplar widely used in microcontroller devices as well as in a number of ASIC and SoC parts with real-time capabilities. The AMBA bus is the focus of our work here.
- The memory controller, which polices access to memory and thus is necessarily shared across cores, causes considerable contention and exacts a high toll on the ETB. Several memory controller designs have been proposed to contain this contention overhead [3] [18] [21] [24], which we consider in this work.

We study how to derive ubd for those two hardware shared resources, assuming RoRo and FIFO arbitration. While other arbitration policies exist that aim at better average performance, they usually lead to more pessimistic – or simply not computable – ubd. This is the case for some types of priority arbitration [19] and policies like first-ready first-come first-served (FR-FCFS) [11].

Let us now look at each policy of interest in isolation.

RoRo. Consider a *RoRo*-arbitrated resource, contended by N_c cores, with an access time $\leq l_{res}^{max}$ cycles, where l_{res}^{max} is the maximum delay that it takes for a request to be serviced by the resource. In Sections V and VI we discuss this delay for the bus and the memory, respectively called l_{bus}^{max} and l_{mem}^{max}.

When core c_i, with $i \in \{1, .., N_c\}$, has the highest priority in a given round of *RoRo* arbitration, the priority ordering for the subsequent round becomes: $\{c_{i+1}, c_{i+2}, ..., c_{Nc}, c_1, c_2, ..., c_i\}$, where c_{i+1} becomes the core with the highest priority and c_i gets the lowest. As *RoRo* is work conserving, a lower-priority requester can be granted access to the resource when all higher-priority contenders do not require it.

When all cores continuously issue access requests, the theoretical worst case is that any request r_i issued from the *scua* always has the lowest priority. We therefore have:

$$ubd_{RoRo} = (N_c - 1) \times l_{res}^{max} \quad (1)$$

Under a contention scheme of this type, both STA and MBTA can be applied to the *scua* in isolation (hence with

no parallel contention) and then the worst-case contention overhead can be added compositionally by factoring the above ubd into each access to the shared resource.

Obviously however, the particular time alignment between the *scua*'s access and the circulation of the *RoRo* priority token across cores determines the contention delay actually suffered, so that the ubd_m may be significantly lower than the ubd. This is further discussed in Section 5.

FIFO. Consider now the same resource, this time with *FIFO* arbitration, accessed by N_c cores, where each core can have only up to one pending request in flight. FIFO assigns access priority in order of arrival so that the requests arriving earlier to the arbiter get higher priority.

The theoretical worst case for the *scua* occurs when all cores have a pending request and a request r_i from the *scua* is preceded by $N_c - 1$ older requests from the other cores. This produces the same ubd as for *RoRo*:

$$ubd_{FIFO} = (N_c - 1) \times l_{res}^{max} = ubd_{RoRo} \quad (2)$$

However, by the time request r_i is issued, the oldest request at the top of the FIFO queue may have progressed to near completion, which – again – causes ubd_m to be lower than ubd. We can thus observe that under both *RoRo* and *FIFO*, the worst case occurs contingent on a particular alignment between the *scua*'s request(s) and those of all other contending cores, and is distinct for each arbitration policy.

2.2 Difficulties in Determining the UBD

When the internal workings of the processor cannot be known, the ubd cannot be determined analytically, but only approximated via ubd_m, as was the case in [17].

We noted earlier that designing observation experiments to maximize the impact that the interfered *scua*'s requests suffer from other cores (which is required to "conjure" the ubd) is impaired by the need to control the alignment in time between the *scua*'s requests and those of the contending cores.

Consider N_c arbitrary software components, $SC = \{sc_1, sc_2, ..., sc_{N_c}\}$, one of which is our *scua*, with each sc_i pinned to a distinct core, all contending access to a *RoRo*-arbitrated resource. It is evident that if we simply run all those programs together, with no other precaution, it would be highly unlikely that each and every *scua*'s request incurred worst-case contention. This is so because, when a request r_i from the *scua* is issued in the program run, its *RoRo* priority is not necessarily the lowest and therefore its wait time is less than ubd_{RoRo}. The case of *FIFO* arbitration is analogous, because it is equally unlikely that every single *scua* request is issued when all other cores have pending requests enqueued and none of them is already being served.

In principle, given a specific *scua*, one might possibly design *matching contenders* capable of issuing their access requests with the frequency needed to cause the *scua*'s requests to always be last in the queue and incur ubd contention. However, this effort would be utterly disproportionate, owing to its extreme sensitivity to the particular behaviour of (the particular version of) the *scua* and, even worse, to its critical dependence on detailed knowledge of the inner workings of the resources of interest so that the desired timing of request generation can be well understood and fully controlled.

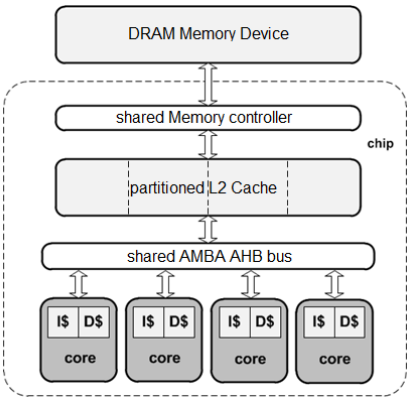


Fig. 1: Block diagram of our reference processor architecture.

We can therefore maintain that soundly approximating the ubd with observation measurements that are affordable for design and implementation costs is an open problem. Interestingly however, solving that problem would be of great value to industrial users, as they would be provided with *scua*-independent test sets capable of causing ubd_m to be a sound approximation of ubd, which could thus be used as an additive factor to the ETB determined for the *scua* in isolation, with state-of-the-art single-core analysis techniques. This is the challenge we tackle here.

3 Elements of the Proposed Solution

We now present the COTS processor that we studied, which we describe first, and the resource-stressing kernels (*rsk*) [7], [16], [23], small application-level programs designed to stress specific hardware resources, which form the state-of-the-art building block to our solution.

3.1 Processor Architecture

The processor we consider in this work is Cobham Gaisler's Next-Generation Multi-Purpose Processor [8], which is one of the multicores currently considered by the European Space Agency for use aboard future satellite missions.

The NGMP is a quad-core processor with private per-core instruction and data caches, referred to as IL1 and DL1 respectively, each with 16 KB capacity, 4-way, 32-byte lines, and 1-cycle hit latency. An AMBA AHB bus serves as the bridge between the IL1 and DL1 on core and the second-level 256 KB 4-way cache (L2), which can be partitioned across cores, one way per core. DL1 is write-through and all caches use LRU replacement. In the NGMP, whose general architecture is depicted in Figure 1, contention only occurs on access to the bus and to the memory controller, since the L2 is partitioned.

3.2 Resource Stressing Kernels

We first discuss the specialization of *rsk* for the processor resources of interest, and then we show that they fail to safely approximate the respective ubd. Subsequently we present a new methodology to do that.

Bus. We call the *rsk* dedicated to the bus, bus stressing kernel (*bsk*). The *bsk* is designed to cause every instruction

```
1: for i = 0 to bus_Accesses/5 do
2:     ld [0x10000000], $0
3:     ld [0x10000400], $0
4:     ld [0x10000800], $0
5:     ld [0x10000C00], $0
6:     ld [0x10001000], $0
7: end for
```

(a) *bsk*

```
1: for i = 0 to bus_Accesses/5 do
2:     ld [0x10000000], $0
3:     ld [0x10010000], $0
4:     ld [0x10020000], $0
5:     ld [0x10030000], $0
6:     ld [0x10040000], $0
7: end for
```

(b) *msk*

Fig. 2: Pseudo-code of *rsk* for the bus made with load operations.

to miss in DL1 and hit in L2. This structure ensures a short turn-around time for memory requests, which keeps the bus as busy as possible.

Given that DL1 uses LRU replacement, the bsk comprises a loop with $W + 1$ load instructions, where W is the number of DL1 cache ways (see Figure 2(a)). Those loads have a predefined stride among them so that they access the same DL1 set, thus exceeding its capacity and systematically missing in DL1. Furthermore, the memory addresses referenced by the *bsk* are designed to exactly fit in L2. In this way, all accesses miss in DL1 and hit in L2.

To hit in L2 we use load operations, which produce the highest bus contention. In the NGMP in fact, L2 hits hold the bus until the L2 serves the request, while L2 load misses are split transactions, which release the bus until memory sends the missed data, and store requests are immediately served, thus keeping the bus busy for a shorter duration.

Figure 2(a) presents the *bsk* for the NGMP: as the DL1 has 4 ways, the loop body of the *bsk* includes five instructions that all map to the same set.

Had the DL1 replacement policy been unknown, we would have designed the loop body to perform $N \gg W+1$ distinct accesses to the same set, for an N that does not exceed the L2 capacity in the corresponding L2 set, to make it highly unlikely for memory operations to hit in DL1.

Memory Controller. Analogously, we call *msk* the *rsk* dedicated to stressing the memory controller. The *msk* design follows the same principles as for the *bsk*, except that the memory accesses in the *msk* have to yield L2 misses. The factors of influence to this end are the size of the way for DL1 and L2 (to cause L2 misses and therefore access the memory controller), and the size of the cache line (that determines the unit of transfer).

For the NGMP, we use a load stride of $64\ KB$, which is an integer multiple of the DL1 way size ($4\ KB$). Hence, all memory accesses map to the same DL1 set. This is also the way size of the L2, hence memory accesses also map onto one and the same set. As L2 uses LRU replacement, every memory access made by the *msk* results in a miss. Figure 2(b) presents the pseudo-code of the *msk*.

4 The Synchrony Effect

Intuitively, one would expect that assigning specialized *rsk* to all cores contending with the *scua* should capture the worst-case contention scenario, and thus allow obtaining a trustworthy approximation of the relevant ubd.

As we show next however, this intuition is wrong in practice, because, when exposed to heavy load conditions, both *FIFO* and *RoRo* experience a particular phenomenon that we term the *synchrony effect*. The essence of this phenomenon is that, when all cores issue requests at a given constant rate to the resource of interest, their requests interleave in a particular way *systematically*, so that their interleaving becomes synchronous. In that situation, the resulting contention delay becomes constant and, more important, unlikely to match the ubd.

We now discuss the *synchrony effect* for the bus, which we obtain by using $N_c - 1$ *bsk* as contenders to the *scua*, under both *FIFO* and *RoRo*. Table 1 lists the key symbols we use in the discussion.

TABLE 1: Main terms used in this paper

Δ_{ET}	Execution time increment suffered by *scua*
n_r	Number of accesses made by *scua*
l_{res}^{max}	Max response time of one resource
N_c	Number of cores
c_i	Core i
R^x	Max no. of requests made by task X in a run
r_i	Request i
δ_i	Injection time between r_i and r_{i-1}
γ_i	Contention delay suffered by request i
et^{isol}	Execution time when running in isolation
et^{rsk}	Execution time when running against rsk
d_{bus}	Execution time increment caused by contention for the bus

4.1 Synchrony Effect under FIFO

The *synchrony effect* causes the shared resource to behave as if it was multiplexed across all cores, with each core being assigned a time slot of duration equal to the service time of an individual request. Interestingly, this applies to both *FIFO* and *RoRo*. Let us now study that effect for *FIFO*.

The contention delay suffered by the *scua* for its request r_{i+1} depends on the time elapsed since its preceding request r_i and how r_{i+1} positions in the request queue.

Let us assume that the *scua* may issue multiple requests to the bus, which we denote $R^{scua} = \{r_0, r_1, ..., r_m\}$. Assume that those requests may be issued at arbitrary times, so that some time elapses between any two subsequent requests from the *scua*. Let us call injection time, denoted δ_i, the time span between the issue of requests r_{i-1} and r_i for any R. Accordingly, for R^{scua}, we have $\{\delta_1^{scua}, \delta_2^{scua}, ..., \delta_m^{scua}\}$.

In our reference architecture, δ_i corresponds to the time elapsed since the data loaded by r_{i-1} is sent back to DL1, until r_i is ready to access the bus. A minimum injection time δ_{min} separates any two subsequent requests from R. δ_{min} is equal to the time it takes for DL1 and the core to process r_{i-1}, once it is served, and execute the instruction corresponding to r_i, until r_i gets ready to access the bus.

When a program runs in parallel with other contenders, each of its request r_i may suffer a contention delay γ_i. Accordingly, for R^{scua}, we have $\{\gamma_1^{scua}, \gamma_2^{scua}, ..., \gamma_m^{scua}\}$.

Since the *bsk* are designed to access the bus with high frequency, their requests have low injection time. In concept, the maximum contention scenario should occur for $\delta_{min} = 0$.

We now illustrate the synchrony effect under *FIFO* with an example where contenders are *bsk* and the *scua* can be either another *bsk* or any other software component.

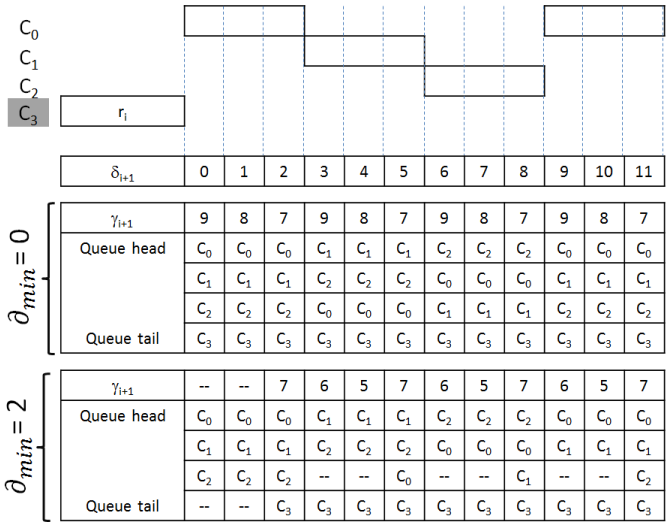


Fig. 3: Contention delay γ as a function of δ (FIFO) for $\delta_{min} = 0$ and $\delta_{min} = 2$, respectively. At each cycle, priorities are shown as at the start of cycle, prior to arbitration.

We explore two scenarios, with $\delta_{min} = 0$ and $\delta_{min} > 0$ respectively. The former, while infeasible in reality, serves for illustration.

Scenario $\delta_{min} = 0$: Let us assume that request r_i of the *scua* is just serviced and all other cores have pending requests enqueued. Figure 3 (rows $\delta_{min} = 0$) illustrates how γ_{i+1} varies as a function of δ_{i+1} (shown in the first row). For instance, if $\delta_{i+1} = 1$, then $\gamma_{i+1} = 8$ since r_{i+1} cannot be granted access to the bus until the ongoing request from c_0 is completed (which takes 2 more cycles) and requests from cores c_1 and c_2 are also serviced (which takes another $3 + 3 = 6$ cycles) since they are both already in the queue.

Assuming that each core can only have one pending request, the worst contention (*ubd*) occurs when r_{i+1} is delayed by the full service of $N_c - 1$ requests coming from the other $N_c - 1$ cores. In this example in Figure 3, this means $\gamma_{i+1} = 9$. When $\delta_{min} = 0$ and l_{bus} denotes the bus service time for an individual request, the synchrony effect manifests in the fact that γ_{i+1} has a periodic behavior that ranges from $(N_c - 2) \times l_{bus} + 1$ (when one contending request is near completion) to $(N_c - 1) \times l_{bus}$ (when all other contending requests are pending and none is being serviced). Thus, the particular value of δ_{i+1} determines the value of γ_{i+1}.

If δ_i^{scua} is arbitrary, it stands to reason that it is very unlikely that all requests $r_i^{scua} \in R^{scua}$ experience $\gamma_i^{scua} = ubd$. If for *scua* we use another *bsk*, which has $\delta_i = 0$ for all $r_i \in R$, as shown in Figure 3, then $\gamma = ubd$ systematically.

Scenario $\delta_{min} > 0$: Owing to cache latency, the common case for the bus is $\delta_{min} > 0$. (For other farther-away off-core resources, such as the memory controller, $\delta_{min} \gg 0$.)

The bottom rows in Figure 3 show the impact on γ_{r+1} when $\delta_{min} = 2$. Right after r_i is serviced, γ_{r+1} would be equal to ubd. However, for 2 cycles r_{i+1} cannot reach the bus and thus $\delta_{r+1} \geq 2$. In particular, if $\delta_{r+1} = 2$, then c_0's request is already being processed at the time r_{i+1} is issued, hence $\gamma_{r+1} < ubd$. If $\delta = 3$, then c_0's request has been processed and its subsequent request will take at least 2 cycles to be issued and reach the queue. Thus, if $\delta = 3$, then

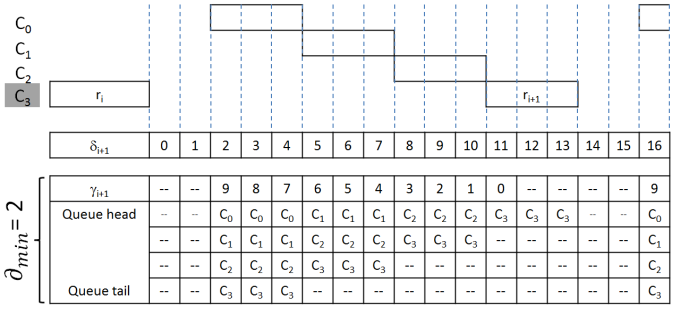


Fig. 4: Example where contention delay γ is maximized for FIFO.

$\gamma_{r+1} = 6$. Analogously, if $\delta_{r+1} = 4$, then $\gamma_{r+1} = 5$. If $\delta_{r+1} = 5$, then r_{i+1} finds the same scenario as for $\delta_{r+1} = 2$, with the only difference that the particular requests in the queue have different core owners, but for the same contention effect on r_{i+1}. Hence, when the *scua* executes against *bsk*, it cannot experience ubd contention regardless of whether the *scua* is a *bsk* itself or not.

In general, if the contending cores execute *bsk*, γ_i^{scua} for request $r_i \in R^{scua}$ can be described with the following equation, where $\delta \geq \delta_{min}$ holds:

$$\gamma_{FIFO}(\delta) = max\left(ubd - ((\delta - \delta_{min}) \mod l_{bus}) - \delta_{min}, 0\right) \quad (3)$$

Note, however, that this does not mean that ubd cannot be experienced systematically. For instance, assume that the *scua* is a *bsk* and the contending cores execute programs that incur $\delta = 11$ in c_0, $\delta = 8$ in $c1$ and $\delta = 5$ in c_2, as shown in Figure 4. In this scenario, after r_i is serviced, the queue is empty for 2 cycles, and when $\delta = \delta_{min} = 2$, then r_{i+1} is issued and contends with requests from all other cores, which arrive simultaneously and are enqueued before it. All requests are processed in order and r_{i+1} experiences $\gamma = ubd$. Then, the queue is empty again for δ_{min} cycles until the same scenario for $\delta = 2$ repeats for $\delta = 16$. However, while this scenario could be hypothetically produced, it is very difficult – if at all possible – for a user to create programs with given δ values, which align in time properly, while ensuring that when requests arrive to the bus simultaneously, they are systematically enqueued in the desired way.

4.2 Synchrony Effect under RoRo

Under *RoRo*, the incoming requests are not necessarily served in order of arrival, but in the order determined by the round-robin assignment of access slots.

Again, we assume that *bsk* are run as contenders. If $\delta_{min} = 0$, all contenders always have a request pending in the queue. Hence, the only parameter that determines who is granted access to the bus is the current priority order. This is better illustrated in Figure 5 (see the $\delta_{min} = 0$ rows). As shown, c_0, c_1 and c_2 always have requests in the queue, either in service or still pending. Notably, r_{i+1} from c_3 becomes the highest priority request when $\delta_{r+1} = ubd = 9$. We also observe that $\gamma_{r+1} = ubd$ only when $\delta_{r+1} = 0$. Otherwise, γ_{r+1} traverses all values from $ubd - 1$ down to 0 consecutively in a round-robin fashion as δ_{r+1} increases.

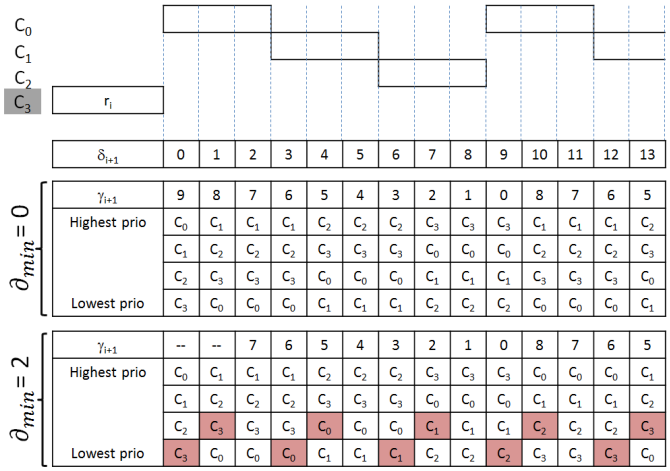


Fig. 5: Contention delay γ as a function of δ (RoRo). In each cycle priorities are those at the start of the cycle, prior to arbitration. Shaded cells in the priority rows correspond to requests not in the queue.

Hence, if $\delta_{min} = 0$, running a *bsk* as *scua* would suffice to observe the highest contention consistently for all of its requests. However, as we noted before, the general case is $\delta_{min} > 0$, owing, for example, to the DL1 cache latency.

Figure 5 also shows the case for $\delta_{min} = 2$. In it, vacant positions in the request queue are marked with shaded cells in the priority rows.

In general, assuming $0 < \delta_{min} \leq ubd$ (as is often the case in reality) so that 100% bus utilization can be reached, then γ stays *exactly the same* as if $\delta_{min} = 0$. This is so because δ_{min} only effects the contents of the request queue. Hence, r_{i+1} can only incur $\gamma_{r+1} < ubd$. Moreover, if δ is constant for all of the *scua*'s requests, then γ is also constant. This observation is of crucial importance in our methodology, as we discuss in the next section.

In the scenario where all contenders are *bsk*, γ can be described with the following equation:

$$\gamma_{RoRo}(\delta) = \begin{cases} ubd & \text{if } \delta = 0 \\ (ubd - (\delta \bmod ubd)) \bmod ubd & \text{otherwise} \end{cases} \quad (4)$$

In general, δ depends on δ_{min} and the particular *scua*. An arbitrary *scua* may observe different values of δ and so little can be concluded about actual contention. Alternatively, running a *bsk* as *scua*, we observe exactly $\gamma = ubd - \delta_{min}$ for all requests. In fact, it is hard to determine the actual value of δ_{min} even when cache latencies are known, since some pipeline stages may delay the access of DL1 misses to the bus. Hence, nothing can be concluded for certain about whether the highest contention has been observed or how far the observation is from the highest extreme.

Taking stock of the *synchrony effect*, we now present a measurement-based method which computes a ubd_m guaranteed to be a safe approximation of the ubd for COTS multicore hardware shared resources, specifically the bus and memory controller, arbitrated with round-robin or FIFO policies.

5 Deriving the UBD for the Bus

In this section, we first describe the strategy we follow. Then we show how it can be implemented and applied in practice for the bus in our reference architecture, considering both *FIFO* and *RoRo* arbitration. Finally, we summarise some architectural issues of relevance.

5.1 Nop-Based Methodology

As captured with Equations 3 and 4, when using *bsk* as contenders, the synchrony effect causes the amount of contention suffered by any request to be a function of δ.

We use that notion to construct a new *bsk*, illustrated in Figure 6(a), which we call *bsk-nop*. In the *bsk-nop* we intersperse low-latency (*nop*) operations between the (load) instructions that access the bus. The effect of those *nop*s is to delay the injection time of each request to the bus, which modifies the δ value accordingly. Hence, whereas in the *bsk*, constituted of consecutive contending requests, we have $\delta = \delta_{min}$, if we add just one (for the sake of example) *nop* in between loads, we obtain $\delta = \delta_{min} + \delta^{nop}$, where δ^{nop} is the delay added by one *nop*.

```
1:  for i = 0 to bus_Accesses/5 do
2:      ld [0x10000000], $0
3:      nop
4:      ld [0x10000400], $0
5:      nop
6:      ld [0x10000800], $0
7:      nop
8:      ld [0x10000C00], $0
9:      nop
10:     ld [0x10001000], $0
11:     nop
12: end for
```

(a) *bsk-nop*

```
1:  for i = 0 to bus_Accesses/5 do
2:      ld [0x10000000], $0
3:      nop
4:      ld [0x10010000], $0
5:      nop
6:      ld [0x10020000], $0
7:      nop
8:      ld [0x10030000], $0
9:      nop
10:     ld [0x10040000], $0
11:     nop
12: end for
```

(b) *msk-nop*

Fig. 6: Code of rsk_{nop} *implementations:* bsk-nop *and* msk-nop

By varying the number k of *nop* instructions inserted between load operations, each resulting bus request experiences a different δ_k. Figure 7 shows this effect for *FIFO* with $\delta_{min} = 1$, which manifests as a saw-tooth profile. An analogous phenomenon occurs for *RoRo*, see Figure 9.

5.2 *Bsk-nop* for *FIFO*

Figure 7 uses Equation 3 to plot γ as a function of $\delta_{min} = 1$. We see there that the values taken by $\gamma = ubd - \delta_{min}$ periodically repeat every l_{bus} cycles. This repetitive behavior reflects the fact that the requests issued by *bsk-nop* over l_{bus} cycles find decreasing contention load in the queue, until a contending request issued by one *bsk* running in parallel on another core is queued again. The maximum contention delay experienced is $\gamma = ubd - \delta_{min}$, hence systematically inferior to ubd, since once a contending request is serviced, it takes δ_{min} cycles for a new request to be enqueued. At that time, contention is highest when the contenders are *bsk*, and amounts to the theoretical worst case (ubd) minus the progress performed during δ_{min} cycles. Observing the saw-tooth shape in Figure 7, we see that its period is equal to l_{bus}. The maximum of the corresponding function is: $(N_c - 1) \times l_{bus} - \delta_{min}$. In this case, ubd corresponds to $N_c - 1$ periods of the function. For instance, if we consider the

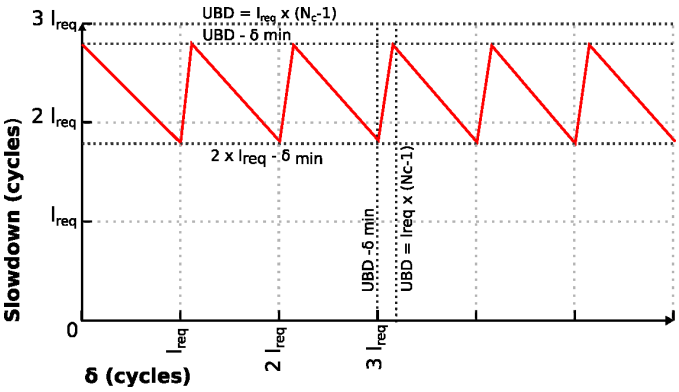


Fig. 7: Saw-tooth behavior for *FIFO* with $\delta_{min} = 1$.

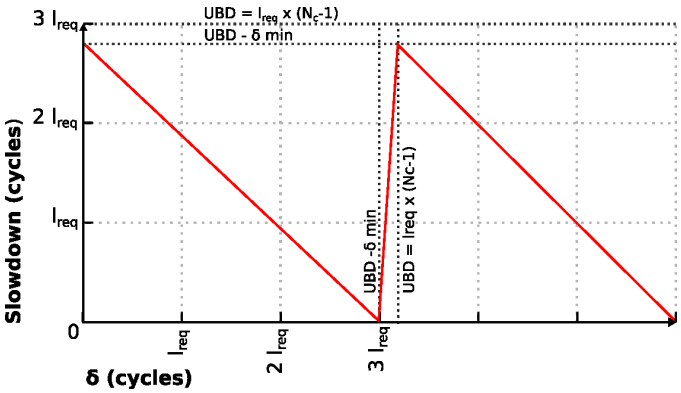


Fig. 9: Saw-tooth behavior for *RoRo* with $\delta_{min} = 1$.

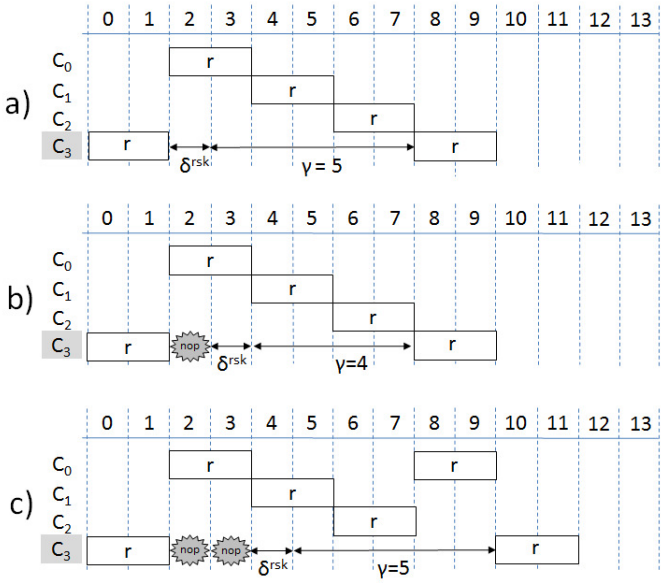


Fig. 8: Timeline of the *FIFO* scenario for different k *nop* instructions: a) $k = 0$, b) $k = 1$, c) $k = 2$.

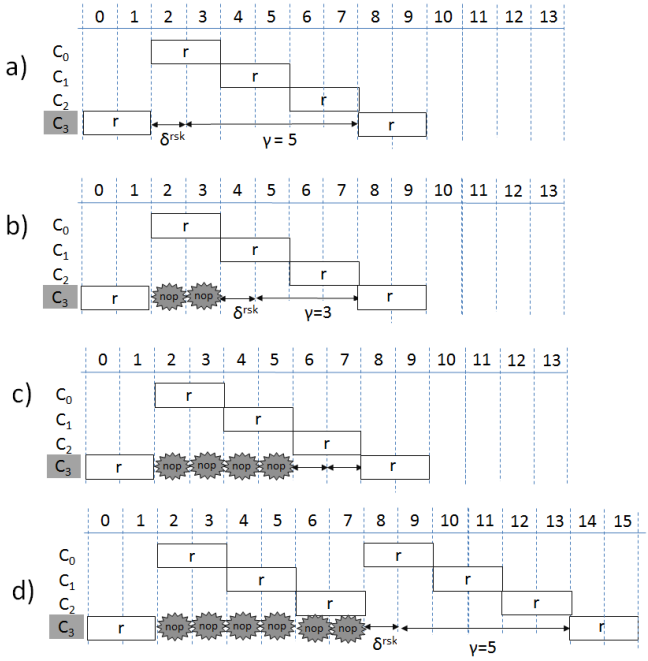


Fig. 10: Timeline of the *RORO* scenario for different k *nop* instructions: a) $k = 0$, b) $k = 2$, c) $k = 4$, d) $k = 6$.

example in Section 4.1 where $N_c = 4$, $\delta_{min} = 2$ and $l_{bus} = 3$, the saw-tooth will range between 7 and 5 cycles, and it will repeat every $l_{bus} = 3$ cycles. Thus, $ubd = l_{bus} \times (N_c - 1) = 9$. As shown, although we cannot observe the actual ubd, we can accurately infer it based on measurements with our methodology.

Figure 8 illustrates this phenomenon, for $l_{bus} = 2$, $\delta^{rsk} = \delta_{min} = 1$, and an increasing number of inserted *nop*s, with $\delta^{nop} = 1$. We start from scenario a), where we assume $\delta^{rsk} = \delta_{min} = 1$ and we see that the request issued from core c_3, where the *scua* runs, suffers a contention of $\gamma(\delta^{rsk}) = 5$ cycles. In scenarios b) and c), we show the effect of increasing the number of *nop* instructions inserted between load operations in all contenders. In scenario b), we see that $\gamma(\delta^{rsk} + \delta^{nop}) = 4$, whereas in scenario c), core c_3 loses its turn for access to the bus, which increases its γ to 5 cycles again and shows the periodicity of γ as a function of l_{bus}, where $l_{bus} = 2 \times \delta^{nop}$ in this case. For higher *nop* counts, scenarios a), b) and c) repeat.

5.3 *Bsk-nop* for *RoRo*

Figure 9 shows the variation in the contention delay incurred with *RoRo*, as captured with Equation 4. The contention value reaches $ubd - 1$ at most, which, for $\delta_{min} > 0$, occurs periodically at every ubd cycles.

This phenomenon is better illustrated in Figure 10, again for $l_{bus} = 2$, $\delta^{rsk} = \delta_{min} = 1$, and an increasing number of inserted *nop*s, with $\delta^{nop} = 1$. We start from scenario a), where the request issued from core c_3, where the *scua* runs, suffers a contention delay of $\gamma(\delta^{rsk}) = 5$ cycles. In scenarios b)-f), we show the effect of increasing the number of *nop* instructions inserted between load operations in all contenders. In scenario b), $\gamma(\delta^{rsk} + \delta^{nop})$ decreases down to 4. Through the scenarios c)-f), $\gamma(\delta)$ keeps decreasing as the number k of inserted *nop* instructions increases from 1 to 5. In scenario g), when $k = 6$, the situation becomes the same as in scenario a).

The following observations are made: (i) for $ubd \geq \delta_{min} > 0$, we have $\gamma \leq ubd - 1$, as per Equation 4; (ii)

the variation of γ is periodic, with period ubd, independent of δ_{min}; and, more importantly, (iii) the exact value of ubd can be inferred from the period of $\gamma(\delta)$, which varies with k: this holds true for any δ_{min} as long as $\delta_{min} \leq ubd$.

5.4 Applying the Rsk-nop Method

Our method to determine the ubd requires carrying out several experiments using *rsk-nop* as *scua* and normal *rsk*s as contenders. *rsk-nop*(k) is parametrized by varying, incrementally, the number k of *nop* instructions inserted between the operations that access the bus.

We run *rsk-nop*(k) against $N_c - 1$ instances of *rsk*, recording its observed execution time, $et^{sc}_{scua}(k)$, and computing the increment from its execution in isolation, $et^{isol}_{rsk-nop}$.

$$d_{bus}(k) = et^{rsk}_{rsk-nop}(k) - et^{isol}_{rsk-nop} \tag{5}$$

Plotting the values of $d_{bus}(k)$ for a range of k, we see a saw-tooth behavior, with period ubd. Assume now that the closest extremes of that period correspond to k_i and k_j respectively. With that in mind, for *FIFO* we have:

$$\begin{aligned} ubd_{FIFO} =& (N_c - 1) \times l_{reqFIFO} \text{ where} \\ l_{reqFIFO} =& |k_i - k_j| : (k_i \neq k_j) \text{ and } (d_{bus}(k_i) = d_{bus}(k_j)) \end{aligned} \tag{6}$$

For RoRo, the ubd can be computed as the period of the resulting saw-tooth shape of d_{bus}.

$$ubd_{RoRo} = |k_i - k_j| : (k_i \neq k_j) \text{ and } (d_{bus}(k_i) = d_{bus}(k_j)) \tag{7}$$

5.5 Deriving L^{max}_{bus}

The magnitude of the ubd depends on two factors: the number of rounds that the request has to wait to gain access to the shared resource of interest (denoted $N_c - 1$); and the longest possible service time from it (denoted l^{max}_{bus}). In the measurement-based approach presented in this work, specialized rsks have to be designed to incur a l^{max}_{bus} response time. In our reference architecture, that duration is determined by whether the accesses to the bus are reads or writes, and hits or misses in the L2. In [12], we empirically determine, for the processor of interest, that read hits to the L2 use the bus for 9 cycles, read misses for 7 cycles, and writes for 1 cycle, regardless of whether or not they miss in L2. In our methodology we secure a l^{max}_{bus} response time by causing all memory operations in the rsk to be read hits to the L2.

5.6 Multicycle Nop Operation

So far we have assumed that $\delta_{nop} = 1$. This is indeed the case in most architectures, since *nop* instructions do not have input/output dependencies and use the fast integer pipeline, if present. In the unlikely case that $\delta_{nop} > 1$, varying the number of *nop* instructions in the *scua* will be equivalent to sampling the saw-tooth behavior shown in Figures 7 and 9. If the value of δ_{nop} can be determined, then we can obtain the saw-tooth period easily. Otherwise, we infer δ_{nop} as follows: we use a *rsk* whose loop body solely includes k *nop* instructions, as many as possible without causing misses in the instruction cache; at that point, by dividing the observed execution time of that *rsk* by k, we obtain a very accurate measure of δ_{nop}.

5.7 Summary

The method we have illustrated in this section empirically derives ubd_m, requiring little in the way of knowledge about the underlying architecture, which is in fact very rarely available as public documentation.

Let us summarize the essence of our contribution at this point. First, we tested our approach for the bus, under *FIFO* and *RoRo*, and we have shown it to work. Second, our approach requires knowing the type of instructions that may generate requests to the bus, which is typically documented in the processor's manuals. Third, we can claim confidence in the ubd_m obtained with our method for two reasons. On the one hand, $N_c - 1$ cores running a *rsk* should suffice to raise the bus utilization up to 100%, also considering the handshaking overhead. This can be ascertained using the performance monitoring counter support provided by most COTS processor architectures (the Cobham Gaisler NGMP, for instance, provides registers $0x17$ and $0x18$ to measure per-core and cumulative bus utilization [9]). On the other hand, we have shown how the user can gauge δ_{nop}, which is needed to determine the saw-tooth period.

The derived bound, ubd_m, can be used by STA as ubd by adding it compositionally to the access time to the bus considered without contention [17]. With MBTA, instead the user must determine an upper bound n_r to the number of requests that the *scua* issues to the bus. The ETB of the *scua* is then padded with the quantity $n_r \times ubd_m$.

6 UBD FOR THE MEMORY CONTROLLER

In this section we show how to empirically derive the ubd for the memory controller. In our reference architecture, the L2 forwards its misses to a request queue located in front of the memory controller. Each core has one entry in that request queue, which therefore has 4 positions. On an L2 miss, a split command is sent to the bus to stall the core that caused the miss, until the corresponding memory request has been served. In the meanwhile, the other cores can continue working. To determine which pending request accesses memory, the memory controller implements two arbitration policies, *FIFO* and *RoRo*, which we discuss below in isolation.

Before we do that, though, we must clarify an inner detail of consequence. Assume that, at a given point in time, the request queue is full, so that it contains N_c requests. Once one of those requests, r_i, has been served, two actions occur. First, a new request r_j from another core is granted access to memory. Second, the core that issued r_i (and has now resumed working) may miss again in L2 and therefore cause a request r'_i to be stored in the request queue of the memory controller. As an L2 access is faster than a memory access, it is fair to assume that, in general, r'_i gets stored in the request queue *before* r_j is served.

As a building block to our measurement-based analysis, we use the *msk* concept outlined in Figure 2(b). This kernel causes a continuous stream of misses in DL1 and in L2, with

each such request going to memory. Following the same methodology as for the bus, we generate a variant of this kernel, called *msk-nop*, which inserts a variable number of *nop* instructions in between cache accesses (cf. Figure 6(b)).

6.1 *Msk-nop* for *FIFO*

Using *msk* as *scua*, under *FIFO* arbitration, we must consider that the time to serve a memory request is longer than the time it takes for the *scua* to reach memory with another request r_{i+1} after its previous request has been served. When r_{i+1} reaches memory, it is preceded by exactly $N_c - 1$ pending requests, one for every other core, which all run *msk*. $N_c - 2$ of those requests are still awaiting service, whereas one of them has begun to be serviced for a duration that corresponds to the δ_{min} factor for memory. We can therefore see that this scenario is analogous to the one we have seen for the bus under *FIFO*, shown in Figure 3 for $\delta_{min} > 0$. The extent of contention captured in that case is high, but not enough to observe a *ubd* contention effect.

Using *msk-nop* as *scua* allows us to explore a range of γ whose period extends to l_{mem}. During that duration, the number of pending requests that precede r_{i+1} is exactly $N_c - 1$ for $l_{mem} - \delta_{min}$ cycles, and $N_c - 2$ for δ_{min} cycles. Plotting the observed γ as a function of the *nop* instructions inserted in the *msk-nop* used as *scua*, we would see the exact same shape as shown in Figure 7, except with a different scale.

6.2 *Msk-nop* for *RoRo*

Analogously to the case of *FIFO* arbitration, if we use an *msk* as *scua*, whenever a request r_{i+1} reaches memory after its previous request has been served, it is preceded by exactly $N_c - 1$ requests. One of those pending requests has begun to be serviced for δ_{min} cycles: this means $\gamma = ubd - \delta_{min}$. We can therefore see that this scenario is analogous to what we saw for the bus with *RoRo*, as shown in Figure 5($\delta_{min} > 0$). Once again, ubd contention is not observed.

Using *msk-nop* as *scua*, we obtain the "sawtooth plot" depicted in Figure 9, in which γ ranges between $ubd - 1$ and 0, which allows us to the derive ubd for the memory controller analogously to what we do for the bus under *RoRo*.

6.3 Deriving L_{mem}^{max}

As for the bus, the response time of requests to main memory, which is required to determine l_{mem}^{max}, may vary. As noted in [15], [21], the duration of a DRAM request in general depends on: i) the memory mapping scheme, which defines the mapping of physical addresses from the processors to the actual memory blocks in the memory devices; ii) the row-buffer policy; iii) the type of the request; and iv) the type of the predecessor request.

The response time to a memory request depends on the type of request, the target page (bank and rank), and the same set of parameters for the immediately preceding request. For instance, serving a request of a given type is typically faster when the preceding request is of the same type (i.e., Read-After-Read or Write-After-Write) than otherwise, and obviously influenced by whether the accesses go to the same bank and rank or not. In the same line, access to open pages (i.e. hitting in the row-buffer) is faster than to close pages that have to be loaded back in the row-buffer.

Those effects have been thoroughly studied in the literature [15], [21] and are typically well documented in DRAM specifications [13], [29], [30].

Based on this information, *msk* can be designed to cause the service time to equal l_{mem}^{max}. All it takes is to alternate the types of operations, and to set the address of the accesses to target the desired bank and rank in accord with the memory row-buffer managing policy and the memory mapping scheme in place.

6.4 Memory Refresh

An intuitive solution to deal with memory refreshes consists in factoring the refresh delay t_{RFC} in the *ubd*. However, this solution is exceedingly pessimistic, as it considers that every individual request is affected by a refresh operation.

With measurement-based approaches instead, the execution time observations taken on the real platform already naturally account for the impact of refreshes. Depending on how measurements align with refresh periods, the number of refreshes that can affect the execution time may be just one more than those actually observed. Hence, it is enough to pad the observed execution time with t_{RFC}.

Another solution is possible when a Δ_{cont} factor is used to compositionally increase the task's WCET, determined in isolation, with the contention overhead on access the bus and the memory computed without considering refreshes. In that case, the number N_{REF} of refresh operations occurring during Δ_{cont} can be easily computed with the following recurrence relation: $N_{REF}^{(k+1)} = \left\lceil \frac{\Delta_{cont} + N_{REF}^{k} \times t_{RFC}}{t_{REFI}} \right\rceil$, where t_{REFI} is the rate at which refresh commands [13] are sent to all banks, and t_{RFC} is the number of cycles that a refresh command takes to complete. In fact, the impact of refresh operations can just be added to the computed WCET, without having to be captured in the computation of the *ubd*. The value to add is given by $(1 + N_{REF}) \times t_{RFC}$, where N_{REF} is the fixed-point solution of the above equation.

6.5 Side Effects of Bus Contention

When deriving the ubd for memory, we must consider that the access requests to it may also compete for the bus, thus incurring some further delay effects. In general however, bus contention is much lower than memory contention, hence the former cannot mask the latter during observation runs. Moreover, owing to the synchrony effect discussed earlier (which originates from the fact that the *msk* issue requests at constant rate), the bus access requests corresponding to memory accesses are served in the bus with the same frequency as the service rate of memory, but with lower occupancy. Assume for example that $l_{mem} = 10$ cycles and $l_{bus} = 2$ cycles. In that case, we could have memory requests served at cycles $[2..11]$ for core c_0, at cycles $[12..21]$ for core c_1, at cycles $[22..31]$ for core c_2, and so forth, and bus requests at cycles $[0..1]$ for core c_0, at cycles $[10..11]$ for core c_1, at cycles $[20..21]$ for core c_2, and so forth.

When using *msk-nop* as *scua*, the issue of requests from c_0 is increasingly delayed until they collide in the bus with

requests from c_1. Under *RoRo* arbitration, this collision is not an issue since which request is granted access to the bus first has no impact on memory contention as long as all contending requests reach memory before the corresponding core becomes the highest priority contender in memory. Under *FIFO* arbitration instead, if both requests are issued to the bus at exactly the same cycle, whether one or the other gets granted first may invert the order of access to memory for one round. However, as long as the hardware behaviour is deterministic, the shape of the plots will remain as in Figure 7, and our approach to derive ubd will continue to work correctly.

7 Evaluation

We first present our experimental set-up. Subsequently, in sections 7.2 and 7.3, we show how *rsk-nop* allows deriving the *ubd* in the face of the synchrony effect. In that narration, we first assume knowing the bus and memory controller latency as well as the actual value of the corresponding *ubd*. This information is instead assumed unknown in Section 7.4, which demonstrates the applicability of our methodology to a real COTS multicore.

7.1 Experimental Setup

We model a 4-core NGMP simulator [8] running at 200 MHz, comprised of a bus that connects cores to the L2 cache and an on-chip memory controller, see Figure 1. Each core has its own private instruction (IL1) and data (DL1) caches. IL1 and DL1 are 16 KB, 4-way with 32-byte lines. The shared second level (L2) cache is split among cores, with each core receiving one way of the 256 KB 4-way L2. Hence, contention only happens on the bus and the memory controller. DL1 is write-through and all caches use LRU replacement policy. Our simulator model includes a closed-page 2-GB one-rank DDR2-667 [29] memory, with 4 banks, burst of 4 transfers, and a 64-bit bus that provides 32 bytes per access, which fits a cache line. In our configuration, the longest service latency for requests of any type is 23 cycles.

In a study that we carried out for the European Space Agency, we assessed the performance fidelity of our simulator against a real NGMP implementation, the N2X [9] evaluation board. To that end, we used a low-overhead real-time kernel that allowed cycle-accurate observations and run benchmark applications on it. The results we obtained for the EEMBC benchmarks [22], a suite of real-world automotive software functions, showed an average deviation of less than 3%. For the HAWAII benchmark [5], an algorithm used to process raw frames coming from the state-of-the-art near-infrared (NIR) HAWAII-2RG detector, the deviation reduced to less than 1%.

7.2 Synchrony Effect on the Bus

In order to show the robustness of the proposed methodology, we evaluate it in the *reference architecture* as presented above, as well as a in *variant architecture* (labeled as *ref* and *var* respectively in following figures). In the latter, we change DL1 and IL1 access latency to 4 cycles (instead of 1 cycle). This variation increases the minimum injection time (δ_{min}) of all bus-access instructions by 3 cycles.

TABLE 2: Randomly-generated workloads used for evaluation

number	the 8 4-task EEMBC-benchmark workloads
1	`cacheb, puwmod, canrdr, rspeed`
2	`iirflt, cacheb, puwmod, canrdr`
3	`ttsprk, iirflt, cacheb, puwmod`
4	`aifirf, ttsprk, iirflt, cacheb`
5	`tblook, aifirf, ttsprk, iirflt`
6	`a2time, tblook, aifirf, ttsprk`
7	`basefp, a2time, tblook, aifirf`
8	`pntrch, basefp, a2time, tblook`

For the purpose of showing how *rsk-nop* enables sound valuations of the ubd to be inferred from ubd_m, we use the following timing information for both architectures. A given request suffers maximum contention latency of $l_{bus} = 9$ cycles per contender: 6 cycles corresponding to the L2 hit latency, and 3 cycles for bus transfer and arbitration handover. Following Equation 1, this yields $ubd = 27$ cycles for the bus.

In a first experiment, we run eight 4-task workloads randomly generated with the EEMBC benchmarks, on the *ref* architecture. The workloads are itemized in Table 2.

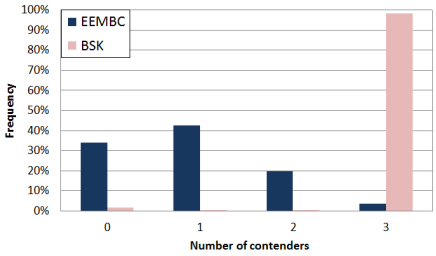
Figure 11(a) presents the histogram of the number of contenders ready to send a request when the EEMBC benchmark in core c_0 requests the bus to start a transaction under *FIFO* (results for *RoRo* are analogous). The results obtained for different workloads are quite similar so we omit them here.

Most of the times, the requests issued by the EEMBC benchmark in c_0 find the bus empty or with just one contender. Only occasionally, the EEMBC in c_0 crosses ways with 2 or 3 contenders. This provides empirical evidence that real application workloads are not easily amenable to generate scenarios in which the number of contending requests is as high as the theoretical worst case. As workloads or time-alignments results may vary, in fact, no a-priori guarantees can be provided that requests align in the worst possible way.

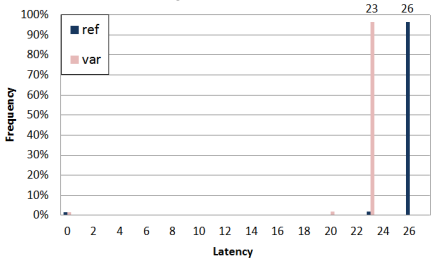
Incidentally, while for *FIFO* all contenders will be served first at some point, in the case of *RoRo* the particular state of the priority assignment determines whether those contenders will be served before or after c_0.

In a second experiment we run 4 *bsk* that constantly access the bus. In this case, (see the pink and light grey bars in Figure 11(a)), we observe that, for almost every arbitration round, the number of contenders is $N_c - 1 = 3$. Hence, the *bsk*s reach their goal of causing maximum contention load on the bus. Yet, owing to the synchrony effect, this ability does *not* suffice to ensure that every individual request from the *scua* incurs a ubd. As we have seen earlier, in fact, when $\delta_{min} > 0$ for both *FIFO* and *RoRo*, the actual contention is always inferior to ubd.

This experiment, in which we run 4 *bsk*s, allows observing this phenomenon in more detail, measuring the actual contention delay γ_i that each individual request issued by c_0 suffers. Figure 11(b) shows the histogram of γ under the reference and the variant architecture. Results for *FIFO* (shown in the figure) and *RoRo* (not shown in the figure) are practically identical. We observe that the synchrony effect causes almost all requests in each case to incur the



(a) Histogram of contenders



(b) Histogram of access latency for *bsk*.

Fig. 11: Results for the bus for FIFO

same latency, since the injection time among requests is the same. Moreover, we observe that the distance among the (observed) ubd_m and the actual ubd (27 cycles in this case) varies across the two architectures: ubd_m is 23 for the *var* architecture and 26 for the *ref* one. This shows that the approximation quality of ubd_m varies as a function of the δ_{min} of the underlying architecture, which in turn disqualifies the use of *bsk*s as sound means to approximate the ubd. As we saw earlier in fact, $ubd_m = ubd - \delta_{min}$ when $0 < \delta_{min} < l_{bus}$.

The 2% requests with different ubd_m correspond to the requests executed until all *bsk* get synchronized and those requests at the beginning of the loop, due to the effect of loop control instructions.

We may therefore conclude that, in the general case, when the details about the latency of the bus are unknown, the use of *bsk*s does not allow estimating the ubd with sufficient accuracy and confidence.

7.3 Synchrony Effect on the Memory

The same conclusions presented in the previous section for the bus, also hold for the memory controller, where a request may suffer a maximum contention of 23 cycles, whereby $ubd = (N_c - 1) \times 23 = 69$ cycles.

Our results, omitted for space constraints, confirm that: i) using three *msk*s, one per core contending with the *scua*, suffices to ensure that more than 98% of the times any requests issued by the *scua* find 3 pending contending requests enqueued at the memory controller; ii) in both the reference architecture and the variant one, the ubd_m is 69 cycles.

7.4 Evaluation of the *Bsk-nop* Methodology for the Bus

For the evaluation of the *bsk-nop* methodology, for FIFO and RoRo, we assume that no latency information is known.

FIFO: As shown in Section 5.1, to infer ubd, the injection time can be varied by inserting *nop* instructions between consecutive accesses of the *rsk* used as *scua*.

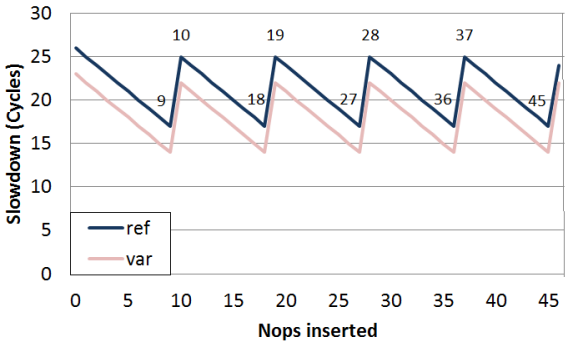


Fig. 12: Slowdown when executing *bsk-nop* as *scua* against 3 *bsk* co-runners with *FIFO*.

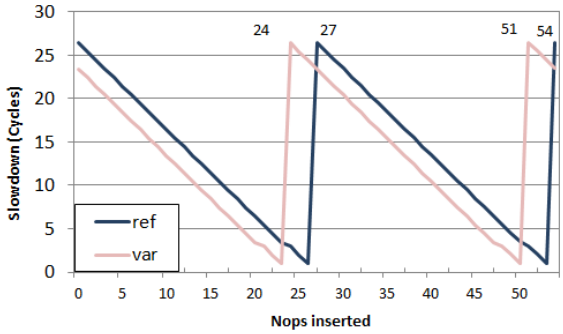


Fig. 13: Slowdown when executing *bsk-nop* as *scua* against 3 *bsk* co-runners with *RoRo*.

The Y-axis in Figure 12 shows the slowdown suffered by *bsk-nop* with respect to its execution in isolation and the horizontal axis represents the variation of γ as a function of the number of nop instructions inserted.

The experimental results match those in Figure 7: the period of each sawtooth is 9 cycles, which corresponds to l_{bus}. As discussed in Section 5.1, however, we have to take into account $N_c - 1$ periods. For instance, from the first peak (cycle 10) until the fourth one (cycle 37), the difference is exactly $ubd = 37 - 10 = 27$ cycles. Notably, the results for the *ref* and *var* architectures are exactly the same, but the absolute contention value decreases as δ_{min} increases.

RoRo: Figure 13 shows the result of the same experiment when the bus uses *RoRo*. As predicted in Figure 9, the slowdown is sawtooth-shaped, with period $ubd = 51 - 24 = 27$ cycles for *var*, and $ubd = 54 - 27 = 27$ cycles for *ref*. Hence, the period of the sawtooth is the same for both architectures, which proves the robustness of our method in inferring the ubd under different processor arrangements.

7.5 Evaluation of the *Msk-nop* Methodology for the Memory

We now repeat the same experiment as for the bus, by injecting nop instructions in the *msk-nop* used as *scua*. Since *ref* and *var* yield analogous results again, we only report those we obtained for the *ref* architecture.

FIFO: The vertical axis in Figure 14 shows the slowdown in cycles, compared with execution in isolation. The horizontal axis shows the number of *nop* operations inserted between memory accesses as shown in Figure 6(b) for the *msk*. We can observe the same sawtooth shape as in Figure 12, but

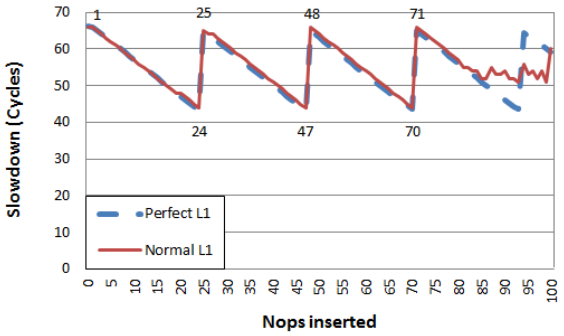


Fig. 14: *Msk-nop* methodology for FIFO.

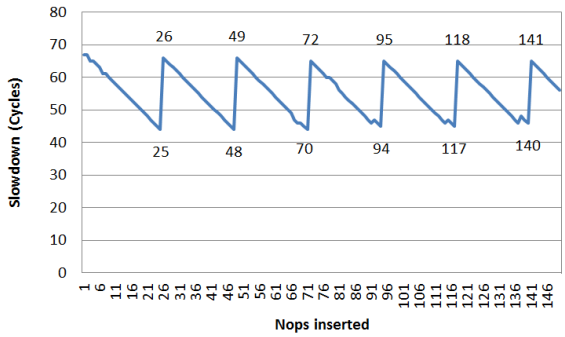


Fig. 15: *Msk-nop* IL1-aware methodology for FIFO.

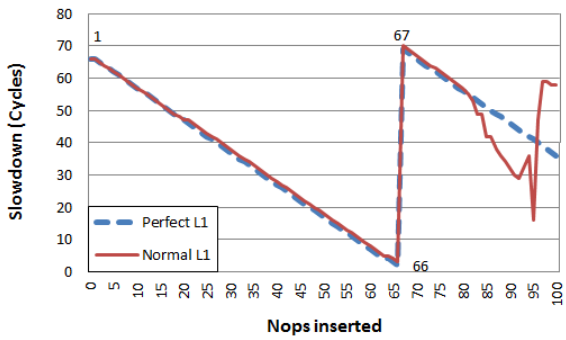


Fig. 16: *Msk-nop* methodology for RoRo

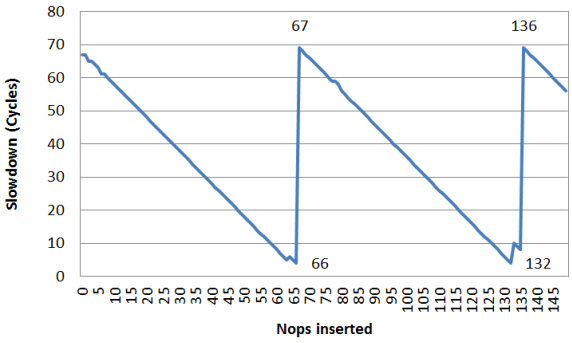


Fig. 17: *Msk-nop* IL1-aware methodology for RoRo

with larger scale. The shape reaches its peak with a period of 23 cycles when 2, 25, 48 and 71, ... *nop* instructions inserted, which means $l_{mem} = 25 - 2 = 23$, as expected.

Beyond 71 *nops*, the results stop following the sawtooth shape. We studied why that happens and concluded that at that point, the number of *nop* instructions in the loop is large enough to exceed the IL1 capacity, so that IL1 misses occur at each iteration. In order to confirm this observation, we repeated the experiment with a processor set-up in our simulator that comprises a perfect IL1, i.e. an IL1 in which all accesses are hits. This is shown as "L1 perfect" in Figure 14: we observe that execution times follow the sawtooth shape, confirming our hypothesis about the increase in the number of conflicts in IL1. In order to solve this problem we propose the following approach.

Instruction Cache-Aware *Msk-nop* Methodology. The *msk-nop* methodology first adds a given number of memory accessing operations (loads) in the main loop. This number is usually high to reduce the overhead (in relative terms) of the loop control applications, see Figure 2. In the *msk* used for the experiments in the previous section, 50 load operations were included in the loop body, whose memory size therefore is around 200 bytes. When we add one *nop* instruction between successive loads, the loop body doubles in size. When the number of *nop* instructions between loads reaches 80, the size grows to $(50 \times 80) \times 4 = 16,000$ bytes, which equals the IL1 size. As shown in Figure 12, the results start degrading just past that number of *nop* instructions.

To test the impact of having more than 80 *nop* instructions between load operations, we simply reduce the number of load operations in the loop body such that its size, taking into account the size of load operations and the *nop* instructions between them, does not exceed the instruction cache size (16KB in this case). For instance, we place 50 loads in the loop for all experiments below 80 *nop* instructions. Then, we reduce the load count down to 40 for all experiments until 100 *nop* instructions, and so forth, always ensuring not to exceed the IL1 capacity.

With the new experiment in Figure 15 we can corroborate that $ubd_m = (49 - 26) \times 3 = 69$ cycles, so $ubd_m = ubd$.

RoRo: Figure 16 shows the results for *RoRo* with the original *msk-nop* that can exceed the IL1 cache size. As for *FIFO*, the shape degrades beyond 80 *nop* instructions, with the difference that, in this case, deriving the ubd may not be possible if we do not fix our *msk-nop* methodology. Again, when making the IL1 perfect, the sawtooth shape obtained is as expected, so we apply exactly the same solution as for *FIFO*: we keep the loop size below the IL1 cache size at all times. We do so with the experiment reported in Figure 17: there we observe that the distance between two teeth of the plot is exactly $udb_m = 136 - 67 = 69$ cycles, so $ubd_m = ubd$.

7.6 Summary

As shown, our methodology based on injecting *nop* operations in the corresponding *rsk* allows producing the sawtooth shapes needed to derive the ubd for both *FIFO* and *RoRo* arbitration policies. Differences in the shape across resources (bus and memory) only affect the scale of the plots, but not their interpretation. Also, we have observed that it is critically important to keep the size of the *rsk* small enough to fit in IL1 to prevent IL1 misses from corrupting the observation results and consequently breaking our methodology.

8 Related Work

Timing analysis techniques can be broadly categorized into Static and Measurement-Based Timing Analysis (STA and MBTA respectively) [34].

STA relies on an accurate timing model of the hardware under test. STA further creates a mathematical representation of the application, which is combined with the timing model to derive bounds to the application's timing behavior on that hardware. STA places strong emphasis on soundness and safeness guarantees, which allows it in principle to conform with the requirements of safety and qualification standards. However, the validity of the bounds depends on the correctness of the hardware timing models, which are difficult to develop and test. This is compounded by the lack of timing information of processor implementations [2]. Even when hardware manufactures provide timing information, experience shows that it can be inaccurate or outdated with respect to the actual chip implementation. For example, the FreeScale e500mc core documentation alone comprises three revisions already, with considerable changes among them [26]. In the case of multicores, this lack of information affects the impact of contention that tasks suffer in the access to shared hardware resources. All these difficulties have caused real-time industry and even STA tool providers to use measurement-based techniques [16] to derive contention bounds, as done for the P4080 [17].

MBTA executes the program on the real platform under stressful conditions and collects measurement observations from it. Those measurements are later reconditioned to approximate an upper bound to the program's WCET. For instance, the longest observed execution time, or high watermark, is recorded and inflated with a safety margin (e.g. 20%) pre-determined based on expert knowledge. For multicores, the reliability of results obtained with MBTA depends, among other factors, on ensuring that the measurement runs cause the application to incur maximum contention (*ubd*) on all of its accesses to all hardware shared resources. Resource-stressing kernels (*rsk*) [23] are used to gauge the contention occurring on access to certain shared resources in parallel processor architectures. They are also used in [7] to characterize the NGMP [8] and in [16] to study the Freescale P4080.

The authors of [1] analyze the impact of resource sharing in multicores and critique the confidence that one can obtain with *rsk*. We acknowledge the need to increase the confidence on the results provided with *rsk*, which in fact is the focus of this paper by proposing the *rsk-nop*-based methodology.

The authors of [28] highlight a counter-intuitive behavior with a RoRo-based multicore: the execution time of a task running against a given number of cores can be smaller than when running against less cores. Our work nails down the prime reason behind this particular behavior, namely the synchrony effect, and takes advantage of it to derive the *ubd*.

WCET estimates for various arbitration policies have been derived in the past for RoRo [19], TDMA [14], a RoRo-inspired group-based policy called MBBA [4], and even comparatively [10]. The authors of [27] propose a method based on Performance Monitoring Counters (PMC) to compute WCET estimates with measurement-based timing analysis, when the *ubd* for a RoRo bus is known.

All these works assume knowledge about the bus timing, whether the slot sizes or the maximum transfer times. Our work requires no knowledge about that.

In the conference version of this paper [6], we concentrated on RoRo arbitrated buses. In this work, we extend our methodology in two directions: we cover another common arbitration policy (FIFO) and provide solutions for another shared resource (memory). Moreover, we also analyze the timing interactions of different hardware shared resources such as the bus and memory access. While the methodology proposed in this paper is assessed against the NGMP processor, we expect it to apply for other processors that embed fully non-blocking caches and out-of-order execution like the ARM Cortex A9 and A15.

Whereas in our reference architecture each core can have a single outstanding request to the L2, thereby exploiting memory-level parallelism among tasks, other architectures allow multiple outstanding requests per core to the L2 to be stored until service. In the latter case, the *rsk* should be designed to ensure that the L2 request buffer saturates so that each request actually takes l_{res}^{max} to be served. Out of order execution, which is a challenge per se for timing analysis, can be accounted for in the design of the *rsk* so that it does not affect the intended behaviour. The fact that *rsk*s use only *nop* instructions and memory operations should ease that fix.

It is worth noting that, at present, the real-time systems industry predominantly uses multicores to consolidate multiple independent applications on the same chip. Those applications either share no data at all or, if they do, the sharing happens off-chip (e.g. in memory). This trend reflects the fact that current timing analysis techniques expect the last-level cache to be partitioned among cores, precisely to prevent data sharing. Hence, while it is clear that, in the long run, parallel (as opposed to partitioned) programming will become mainstream for real-time systems, this is still a recent (and very active) area of research, not yet ready for industrial use. For this reason, in this work we can safely assume that the application programs are partitioned across cores and do not share data, so that the coherence mechanism does not perturb the execution-time measurements.

9 Conclusions

The lack of information about the internal working of modern COTS processors makes the use of measurement observations the sole viable means to infer the timing parameters required to dimension the worst-case execution time of application programs.

For the bus and the memory, which are highly shared resources in multicore hardware, the parameter of interest is the maximum contention delay that a request can suffer on access, which we call upper-bound delay, *ubd*.

The level of trust that can be placed in the execution time bounds derived for application programs running on COTS multicore processors depends on the soundness of the analysis technique in use and the accuracy of the timing parameters that it employs, including the *ubd* for buses and memory.

In this paper we have presented a measurement-based methodology that requires no knowledge on the timing parameters for access to the bus and memory resources, and yet is able to derive their *ubd* soundly and tightly. This ability increases the confidence in the execution-time bounds computed for application programs running on COTS multicore processors that use FIFO or RoRo arbitration.

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 TIN2015-65316-P. Jaume Abella has been partially supported by the MINECO under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717. The authors would like to thanks Paul Caheny for his help with the proofreading of this document.

REFERENCES

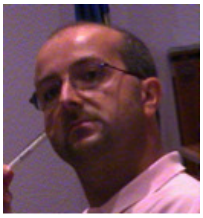
[1] Andreas Abel et al. Impact of resource sharing on performance and performance prediction: A survey. In *CONCUR*, 2013.
[2] Jaume Abella. et al. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *SIES*, pages 1–10, June 2015.
[3] Benny Akesson et al. Predator: A Predictable SDRAM Memory Controller. In *CODES+ISSS*, 2007.
[4] Roman Bourgade et al. MBBA: A multi-bandwidth bus arbiter for hard real-time. In *Embedded and Multimedia Computing (EMC)*, 2010.
[5] European Space Agency. "GENERAL BENCHMARKING AND SPECIFIC ALGORITHMS. NIR HAWAII-2RG BM algorithm". Internet: http://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Data_Processing/General_Benchmarking_and_Specific_Algorithms.
[6] Gabriel Fernandez. et al. Increasing confidence on measurement-based contention bounds for real-time round-robin buses. In *DAC*, 2015.
[7] Mikel Fernández et al. Assessing the suitability of the NGMP multi-core processor in the space domain. In *EMSOFT*, 2012.
[8] Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor - Data Sheet and Users Manual*, 2011.
[9] Cobham Gaisler. *LEON4-N2X Data Sheet and User's Manual*, 2013.
[10] Javier Jalle et al. Deconstructing bus access control policies for real-time multicores. In *SIES*, 2013.
[11] Javier Jalle et al. A dual-criticality memory controller (DCmc): Proposal and evaluation of a space case study. In *RTSS*, 2014.
[12] Javier Jalle et al. Bounding Resource Contention Interference in the Next-Generation Microprocessor (NGMP). In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, 2016.
[13] JEDEC. *DDR2 SDRAM Spec. JEDEC Standard No. JESD79-2E*, 2008.
[14] Timon Kelter et al. Bus-aware multicore WCET analysis through TDMA offset bounds. *ECRTS*, 2011.
[15] Hyoseung Kim. et al. Bounding memory interference delay in cots-based multi-core systems. In *RTAS*, 2014.
[16] Jan Nowotsch et al. Leveraging multi-core computing architectures in avionics. In *EDCC*, 2012.
[17] Jan Nowotsch et al. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS*, 2014.
[18] Marco Paolieri et al. An analyzable memory controller for hard real-time CMPs. In *Embedded System Letters (ESL)*, 2009.
[19] Marco Paolieri et al. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009.
[20] Marco 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.
[21] Zheng PeiWu et al. Worst case analysis of DRAM latency in multi-requestor systems. In *RTSS*, 2013.
[22] Jason Poovey et al. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.
[23] Petar Radojković et al. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM TACO*, 2012.
[24] Jan Reineke et al. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *CODES+ISSS*, 2011.
[25] Erno Salminen et al. Benchmarking mesh and hierarchical bus networks in system-on-chip context. *Journal of Systems Architecture*, 2007.
[26] FreeScale Semiconductor, Inc. *e500mc Core Reference Manual. Rev 3*, 2013.
[27] Hardik Shah et al. Measurement based WCET analysis for multi-core architectures. In *RTNS*, 2014.
[28] Hardik Shah et al. Timing anomalies in multi-core architectures due to the interference on the shared resources. In *ASP-DAC*, 2014.
[29] Kingston Technology Corporation. *KVR667D2S5/2G Datasheet*, 2011.
[30] Micron Technology, Inc. *DDR2 256 Mbit datasheet*, 2006.
[31] Aniruddha N. Udipi et al. Towards scalable, energy-efficient, bus-based on-chip networks. In *HPCA*, 2010.
[32] Christopher B. Watkins et al. Transitioning from federated avionics architectures to integrated modular avionics. In *DASC*, 2007.
[33] Adam West. NASA study on flight software complexity. Final report. Technical report, NASA, 2009.
[34] Reinhard Wilhelm. et al. The worst-case execution time problem: overview of methods and survey of tools. *ACM TECS*, 7(3):1–53, 2008.

Gabriel Fernandez is a PhD student at the Computer Architecture Department of the Technical University of Catalonia (UPC) and a junior researcher at the Barcelona Supercomputing Center(BSC). He received his M.Sc. degree in Computer Science Research and Investigation from UPC in 2015. His research focuses on enhancing the time predictability on COTS multicore.

Javier Jalle received his PhD degree in Computer Architecture from the UPC in 2016. He worked as a PhD student at the Computer Architecture Department of the UPC and a junior researcher at the BSC, Spain. He was also part of the NPI program of the European Space Agency during his PhD. His research focused on improving the time predictability of multicore processors. Currently, he is a Software engineer at Cobham Gaisler AB.

Jaume Abella is a senior PhD. Researcher in the CAOS group at BSC. He worked at the Intel Barcelona Research Center (2005-2009) in the design and modeling of circuits and microarchitectures for fault-tolerance and low power, and memory hierarchies. Since 2009 in BSC he is in charge of hardware designs for FP7 PROARTIS and PROXIMA, and BSC tasks in ARTEMIS VeTeSS. He has authored +15 patents and +80 papers in top conferences and journals.

Eduardo Quiñones is a senior PhD. Researcher at BSC. He received his MS degree in 2003 and his PhD. in 2008 at the UPC. His area of expertise is in safety-critical systems and high performance compiler techniques. He is involved in several FP7 European projects (parMERASA, PROARTIS, PROXIMA and P-SOCRATES), as well as some bilateral ESA-BSC projects. He spent one year as an intern at Intel Research Labs (2002 - 2003).



Tullio Vardanega is at the University of Padua, Italy. He holds a MSc in Computer Science from the University of Pisa, Italy, and a PhD in Computer Science from the Technical University of Delft, Netherlands. He worked at European Space Agency (ESA) from July 1991 to December 2001. At the University of Padua, he teaches and leads research in the areas of high-integrity distributed real-time systems and advanced software engineering methods. He has a vast network of national and international research collaborations. He has co-authored 150+ refereed papers and held organizational roles in several international events and bodies, for ESA, the European Commission, ISO, IEEE and Ada-Europe.

Francisco J. Cazorla is the leader of the CAOS group at BSC. He has led projects funded by industry (IBM and Sun Microsystems), by the European Space Agency (ESA) and public-funded projects (FP7 PROARTIS project and FP7 PROXIMA project). He has participated in FP6 (SARC) and FP7 Projects (MERASA, VeTeSS, parMERASA). His research area covers both high-performance and real-time systems on which he is co-advising several PhD theses. He has co-authored 3 patents and over 100 papers in international conferences and journals.