Increasing Confidence on Measurement-Based Contention Bounds for Real-Time Round-Robin Buses

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

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

ABSTRACT

Contention among tasks concurrently running in a multicore has been deeply studied in the literature specially for on-chip buses. Most of the works so far focus on deriving exact upper-bounds to the longest delay it takes a bus request to be serviced (*ubd*), when its access is arbitrated using a time-predictable policy such as round-robin. Deriving *ubd* for a bus can be done accurately when enough timing information is available, which is not often the case for commercial-of-the-shelf (COTS) processors. Hence, *ubd* is approximated (ubd_m) by directly experimenting on the target processor, i.e by measurements. However, using ubd_m makes the timing analysis technique to resort on the accuracy of ubd_m to derive trustworthy worst-case execution time estimates. Therefore, accurately estimating *ubd* by means of ubd_m is of paramount importance. In this paper, we propose a systematic measurement-based methodology to accurately approximate *ubd* without knowing the bus latency or any other latency information, being only required that the underlying bus policy is round-robin. Our experimental results prove the robustness of the proposed methodology by testing it on different bus and processor setups.

1. INTRODUCTION

The pressure on real-time industry to adopt multicore as its reference processing platform has increased over the last years. Chip vendors are driven by the mainstream market and its high performance demands rather than by the timing requirements of the comparatively small real-time market. Further, evaluations of – typically non commercial-off-the-shelf (COTS) – multicores in real-time systems performed by academia envisage significant benefits of multicores.

Real-time industry, though, is far from completing the transition to multicores: real-time industry needs to resort to COTS processors to obtain the level of performance needed at an affordable cost. However, COTS processors do not target time predictability as needed in the real-time domain. This calls for timing analysis solutions for COTS multicores, for which to our knowledge a full-fledged WCET estimation solution does not exist. For this reason, in this paper we talk about execution time bounds (ETB) rather than Worst-Case Execution Time (WCET) estimates that is used for single-cores, or customized WCET-aware multicores, for which timing analysis techniques are much more mature.

COTS multicores challenge timing analysis due to the difficulties to consider the impact of contention in shared resources (e.g., a shared bus) on ETB. There have been several works analyzing the worst-case contention that tasks in a multicore suffer due to access to the on-chip bus [6]. For static timing analysis (STA), if enough information about the processor is available, it can be derived the worst impact of contention that applications requests can suffer on the access to the bus, called Upper Bound Delay (*ubd*), which can be then factored in when deriving *ETB*. However, as the complexity of the multicore processors used in real-time domains continues to increase and the information about their internal functioning is not available, the a-priori analytical derivation of *ubd* becomes harder. As a matter of fact, the contention of the P4080 processor has been analyzed by an avionics end-user and a STA tool provider [12] using *measured ubd* values (ubd_m), i.e. values derived from experimentation on the P4080 [11], rather than *ubd*. This fact talks about the difficulties that COTS-multicore end-users and STA tool-providers have in finding processor internal information, about the memory bus in this case, to derive *ubd*. Hence, the confidence of the resulting ETB rests on the confidence on ubd_m and, in particular, on how accurately it approximates the actual *ubd*. In the context of Measurement-Based Timing Analysis (MBTA) deriving ubd_m is also key to determine whether the accesses to a shared resource of a task running in a multicore experience high contention. This ultimately increases confidence on MBTA, which is widely used across automotive, avionics and space industries among others.

To our knowledge, so far ubd_m has been obtained with specific user-level application kernels called, micro-benchmarks or *resource stressing kernels* (*rsk*) [15, 11, 5]. The basic methodology to derive ubd_m consists in running a given *software component under analysis* (*scua*) against several *rsk*. In particular ubd_m is derived dividing the execution time increase of the *scua* w.r.t. its execution time in isolation ($d_{et} = ExecTime_{rsk} - ExecTime_{isol}$) by the number of bus requests made by the *scua*, n_r. That is $ubd_m = d_{et}/n_r$. Despite *rsk* are designed to put high load on a target shared resource (e.g., the bus) so that the *scua* slowdown significantly increases, no evidence has been provided about whether ubd_m closely matches *ubd* for this methodology based on running a given *scua* against several *rsk*, or only running several copies of the *rsk*. Focusing on round-robin(RR) buses, this paper makes the following contributions.
1) We show that running a *scua* against *rsk* putting high load on the bus does not make that all *scua*'s bus accesses suffer a contention equal to *ubd*. We also show that taking as *scua* one *rsk* and running it against several *rsk* copies neither ensures that each request to the bus suffers *ubd* nor helps deriving a good approximation (ubd_m) to it.
2) We identify the reasons behind this inability to derive a ubd_m that closely approximates *ubd*. We show that under heavy load scenarios round-robin presents a *synchrony effect* that makes each request in the *scua* suffer a given (single) contention delay that cannot be ensured to match *ubd*. Contention delay is determined by the time elapsed since the previous request was served until the current one becomes ready to be sent to the bus, which we call the injection time.

DAC '15 June 07 - 11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3520-1/15/06...$15.00
http://dx.doi.org/10.1145/2744769.2744858.

3) We propose a methodology to derive *ubd* resorting on measurements and without knowing the latency of the bus, hence being applicable to a wide range of processor designs. The basic approach consists in varying the injection time between requests to the bus until each request suffers *ubd.* This is implemented by injecting *nop* operations among the requests accessing the bus.

With our methodology we successfully derive *ubd* on two multicore setups, one of which matches the Cobham Gaisler NGMP multicore processor [2]. Overall, our methodology helps increasing the trustworthiness of the derived ETB of the timing-analysis tool/technique on COTS multicores deploying buses.

2. ON-CHIP BUS CONTENTION ANALYSIS

One of the most critical shared resources in multicore processors is the interconnection network. Bus-based networks can significantly lower energy consumption and simplify network protocol design and verification, with small impact in performance [16, 19]. Examples of widely used buses include the Advanced Microcontroller Bus Architecture (AMBA) that is used not only in microcontroller devices but also on a range of ASIC and SoC parts with real-time capabilities.

We study round-robin (RR) which allows deriving time- composable bounds for the access latency to the bus, such that the load that a requester puts on the bus does not affect the *time bounds* derived for another requester [13]. Let us assume a RR arbitrated bus accessed by N_c requesters (cores). If requester c_i, with $i \in \{1,..,N_c\}$, is granted access in a given round, the priority ordering for the next round is defined as follows: $c_{i+1}, c_{i+2}, ..., c_{Nc}, c_1, c_2, ..., c_i$. The core with the highest priority is c_{i+1} and c_i is the one with the lowest. Since RR is work conserving, a lower priority requester can use the bus when all higher priority requesters do not use it. The *ubd* that a request can suffer due to contention corresponds to the case in which the request has the lowest priority and all the rest of sequesters have pending requests. In that situation, the request has to wait for all the $N_c - 1$ sequesters to use the bus for a maximum of l_{bus} cycles:

$$ubd = (N_c - 1) \times l_{bus} \qquad (1)$$

When not enough details about the hardware are known [12] ,*ubd* cannot be obtained analytically but it has to be derived by experimentation. One of the main complexities when designing an experiment to maximize the impact that the interfered *scua*'s bus requests suffer from other co-running software components is that contention depends on how *scua*'s requests aligns with contending requests. Let us assume several arbitrary software components, $SC = \{sc_1, sc_2, ..., sc_{N_c}\}$, one of which is our *scua* with N_c being the number of cores. If we run all SC, it is unlikely that each *scua*'s request suffers *ubd*, since when a request r_i from the *scua* becomes ready, its RR priority is not necessarily the lowest one and hence, it does not have to wait *ubd* cycles for the bus.

Given a *scua*, in theory, one could design a *worst-contender sc* such that every time the *scua* tries to send a request, it has the lowest priority and all worst-case contenders running in the other cores have a request ready at exactly that moment. This makes the *scua* suffer *ubd* on every request. However, such a worst-case scenario is very complex to reproduce because the worst-contender is *scua*-dependent. And more importantly, the level of knowledge on the timeliness of the requests done by the *scua* and control required on the worst-contender to generate requests in the desired processor cycles is too high to be a viable solution. Hence, from an user perspective it is hard, if at all possible, controlling the particular cycles when the worst-contender issues its requests to enforce a particular interleaving with the request of the *scua.* Overall, we conclude that the approach based on designing worst-case contenders is not possible in general.

```
for i = 0 to $bus_{Accesses}/5$ do
    ld [0x10000000], $0
    ld [0x10001000], $0
    ld [0x10002000], $0
    ld [0x10003000], $0
    ld [0x10004000], $0
end for
```

(a) rsk l2hit

```
for i = 0 to $bus_{Accesses}/5$ do
    ld [0x10000000], $0
    nop
    ld [0x10001000], $0
    nop
    ld [0x10002000], $0
    nop
    ld [0x10003000], $0
    nop
    ld [0x10004000], $0
    nop
end for
```

(b) rsk-nop l2hit (k=1)

Figure 1: Pseudo-code of one *rsk* and *rsk-nop*

From the analysis point of view the goal is accounting *ubd* for every request to the shared resource when deriving ETB for a *scua.* Estimating *ubd* with measurements requires architecting a set of test cases such that enough confidence can be obtained on the execution time measurements to capture the worst contention for the *scua*, so that every request to the shared resource suffers *ubd.*

As building-blocks to design such measurement-based approach we use *rsk* [15, 11, 5], which are small user-level kernels that stress specific hardware resources. *rsk* comprise a single loop with instructions of the same type that are chosen to stress a specific hardware resource. In the particular case of this paper we designed a *rsk* that puts high load on the bus on our reference architecture, in which the bus serves as bridge between private on-core L1 instruction (IL1) and data (DL1) caches and the L2 cache. The *rsk* is architected so that every instruction misses in the DL1 and hits in the L2. This ensures a short turn-around time for the requests hence keeping the bus used as much as possible. For a least recently used (LRU) or FIFO replacement policy, we do so by building a loop with $W + 1$ instructions, where W is the number of DL1 cache ways. For instance, if $W = 4$, five instructions are needed in the loop (see Figure 1(a)). Those instructions are loads having a predefined stride among them which makes them to be mapped into the same DL1 set and to exceed its capacity, hence systematically missing in DL1. Further, accessed addresses fit in L2. In this way all accesses miss in DL1 and hit in L2. Other *rsk* designs focusing on exceeding cache capacity, not a single set, can be easily implemented.

3. SYNCHRONY EFFECT

Intuitively one would expect that running a *scua* against several *rsk* represents the worst-case contention scenario, which could be used to obtain ubd_m. Next we show that this intuition is wrong in practice. This is so because under heavy load conditions RR buses trigger a *synchrony-effect.*

The *synchrony-effect* makes the bus behave as if it were time-multiplexed among cores, with each core having a time slot equivalent to the delay it takes to process one request. In such a time-shared scenario the RR time window is equivalent to the addition of the time slot given to each core. The latency suffered by each *scua* request(r_i) depends on the time interval between r_i and its preceding request r_{i-1} and how it aligns to the RR time window.

In this section we analyze the *synchrony effect* under a scenario with high load on the bus, which we achieve by using $N_c - 1$ *rsk* as the contenders of the *scua.* In a first experiment, Section 3.1, we consider as *scua* an arbitrary *sc*. In a second experiment, Section 3.2, we consider as *scua* a *rsk.* All experiments are carried out in our reference multicore that is detailed in Section 5(a 4-core multicore with each core comprising a private DL1 and IL1, each core connects to the L2 cache through a RR arbitrated bus). Designing a *rsk* that stresses the bus requires that most of the *rsk*'s instructions access and hit in the L2 cache. In particular we use load operations that hit in

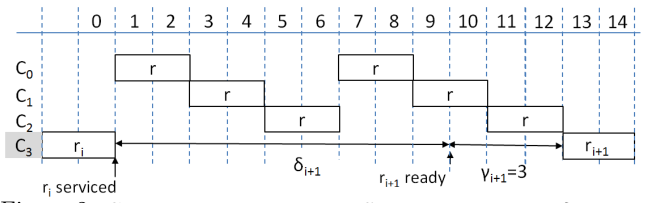


Figure 2: Contention delay, γ_{i+1}, suffered by r_{i+1} when $\delta_{i+1} = 9$

L2. L2 load hits produce the highest bus contention since they keep the bus busy until the L2 answers the request, rather than splitting the request as it occurs with L2 load misses or being immediately answered as happens with all store requests.

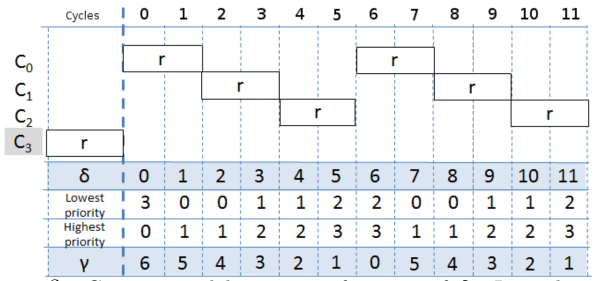
3.1 Synchrony effect: scua against rsk

Let us assume that the *scua* has several requests to the bus: $R = \{r_0, r_1, ..., r_m\}$. Those requests occur at arbitrary times (e.g., due to a DL1 miss), so that some time elapses since a request r_i is serviced until the next one, r_{i+1}, is ready to be issued. Such inter-request latency also determines the (injection) time between the requests they generate. Let δ_i be the injection time between requests r_{i-1} and r_i. Hence, for the *scua* we have $\Delta_{scua} = \{\delta_1, \delta_2, ..., \delta_m\}$. In our reference architecture δ_i is equivalent 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. When the *scua* runs simultaneously with other *sc*, each of its request r_i may suffer a contention γ_i. Hence, we have $\gamma_{scua} = \{\gamma_0, \gamma_1, ..., \gamma_m\}$ for the requests in the *scua*.

As *rsk* put high load on the bus, intuitively using $N_c - 1$ *rsk* as the contenders of the *scua* should make that each of its requests suffer *ubd* in the bus. Since *rsk* are designed to access the bus with high frequency, they have low injection time among requests δ^{rsk}. Of course, *rsk* must not complete execution before the *scua*.

When a given *scua* runs (in a given core c_j) against $N_c - 1$ *rsk* we observe that in the same cycle when a given request r_i of the *scua* is completed, each of the $N_c - 1$ *rsk* have a pending request. This is generally the case since $N_c - 1$ can fully load the bus – otherwise the resource would be overdimensioned. That is, the time it takes r_i to be processed by the bus, is longer or equal than δ^{rsk}, which is the time any *rsk* needs to have a request ready. As a result, the rounds of arbitration after r_i is processed are fixed with RR priority given to cores c_{j+1}, c_{j+2}, ..., c_{N_c}, c_1, ... c_j. In Figure 2, once request r_i sent from core c_3 is serviced in cycle 0, the requests from the other contenders (c_0, c_1, c_2) are ready. In this scenario, the sequence of events after r_i is processed is fixed. In fact the same sequence happens after every request of c_3. This sequence starts with the grant being given to c_0 then to c_1 and finally to c_2. If at the end of this sequence c_3 has another request r_{i+1} ready, it is given the grant and the process starts again. The arbitration sequence repeats until c_3 has a request. *Note that it does not matter how RR priorities are assigned at the beginning of the execution*: after the first request of c_3 the rounds of arbitration, and hence the contention delay each request suffers (γ_i) by the following requests, are the same and depend on Δ_{scua}.

In the scenario drawn in Figure 2, r_{i+1} becomes ready in cycle 9 when the grant is given to c_1, so it has to wait $\gamma_i = 3$ cycles that is smaller than *ubd*, 6 cycles in this example. In a different run r_{i+1} may become ready in a different cycle, hence suffering different contention. Hence, the fixed injection time among requests makes that each request r_i suffers a given γ_i in each run that can be smaller than *ubd*. Overall, the synchrony behavior that RR presents under high load conditions makes that *the sequence of events after each arbitration is the same* so that running the *scua* against *rsk* fails achieving that each request of the *scua* suffers *ubd*.



δ	0	1	2	3	4	5	6	7	8	9	10	11
Lowest priority	3	0	0	1	1	2	2	0	0	1	1	2
Highest priority	0	1	1	2	2	3	3	1	1	2	2	3
γ	6	5	4	3	2	1	0	5	4	3	2	1

Figure 3: Contention delay γ as a function of δ. In each cycle priorities are those at the start of the cycle, prior to arbitration.

3.2 Synchrony effect: rsk against rsk

Next we show that when we use a *rsk* as *scua* and run it in a workload with other *rsk* as contenders, requests of the *scua* do not suffer *ubd* delay. In such experiment all requests have the same injection time, i.e $\delta_i = \delta, \forall i$, with $\delta = \delta^{rsk}$.

Let us assume that request r_i from core c_j becomes ready exactly the same cycle when the previous request r_{i-1} completes its execution in the bus, i.e. $\delta = 0$. In that scenario, we know that 1) in that very same cycle the priority of c_j is the lowest, and 2) due to the *synchrony effect* in that very same cycle the rest of contenders will have a request ready and hence r_i would suffer a contention $\gamma = ubd = 6$. This scenario is presented in the upper part of Figure 3. The first column in the matrix in the lower part of the figure shows δ – that equals 0 in this case –, the core with the highest and lowest priority, and the contention delay γ suffered by r_i.

Let us now assume that the injection time is higher than 0, $\delta > 0$. This may be due to the fact that after r_{i-1} completes in the bus, it takes a given processing time the core to execute the next instruction that generates r_i. In the matrix at the bottom of Figure 3 we see that, as δ increases γ_i decreases down to 0. This happens when all the other contenders have already processed their requests and the priority of c_3 is the highest, making r_i suffer a contention $\gamma = 0$. In Figure 3 we observe that this latter case happens for $\delta = 6$. When the delay of the current request r_i and the previous request r_{i-1} is $\delta = ubd + 1 = 7$, by the time r_i is sent c_3 has the lowest priority and the next core in RR order, $c_{(3+1)\%4} = c_0$, already spent one cycle on the bus, so $\gamma = ubd - 1$. The same behavior repeats periodically with a period of *ubd*. Overall, the contention delay, γ, that each request of the *rsk* suffers is:

$$\gamma(\delta) = \begin{cases} ubd & \text{if } \delta = 0 \\ (ubd - (\delta \bmod ubd)) \bmod ubd & \text{otherwise} \end{cases} \quad (2)$$

If $\delta = 0$ then each *rsk* request suffers a contention delay $\gamma = ubd$. However, if there is a minimum injection time between the accesses generated by two consecutive instructions, the *ubd* is never reached, despite the *rsk* having consecutive instructions making requests to the bus. Hence, our methodology has to deal with this consideration when deriving *ubd*.

In the general case δ depends on the architecture under consideration. For instance, in our reference architecture, with $\delta = \delta^{rsk} = 1$, we are only able to reach $\gamma(\delta) = ubd - 1$. As seen in Figure 3, for $\delta = 1$ we have $\gamma = 5$ (so $ubd - 1$). Without knowing the particular values of *ubd* and δ it is hard – if at all possible – to determine which value has γ_i for each request r_i in *rsk* and even harder ensuring if γ_i matches *ubd*. Overall, using *rsk* as *scua* and running it against other *rsk* is not sufficient to make $ubd_m = ubd$.

4. PROPOSED METHODOLOGY

Our goal is deriving a methodology based on several test cases executed on the target multicore platform to derive *ubd*. We build on the *synchrony effect* presented in Section 3.

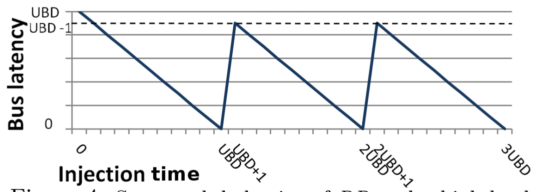


Figure 4: Saw-tooth behavior of RR under high load

4.1 The rsk-nop kernel

The synchrony effect allows enforcing each request to suffer contention as determined by the injection time (see Equation 2). To that end we generate a new *rsk* called *rsk-nop*(Figure 1(b)) in which we inject low-latency operations, e.g. *nop* operations, between the instructions accessing the bus (e.g., loads). Those *nop*s delay the cycle in which each request to the bus becomes ready with respect to the previous request, which artificially modifies the injection time of the *rsk*. That is, while in the original case with consecutive bus-accessing operations the time between them is $\delta = \delta^{rsk}$, if we add one *nop* the latency becomes $\delta = \delta^{rsk} + \delta^{nop}$, where the latter is the delay added by one *nop*.

By varying the number of *nop* operations, k, inserted between bus-accessing operations, each request experiences different contention delays (δ). As a result, the contention delay experienced by the different *rsk-nop* has a saw-tooth behavior as shown in Figure 4. The maximum contention (shown in the Y axis) obtained with Equation 2 is *ubd* and only occurs when $\delta = 0$ (shown in the X axis). With $\delta > 0$, the maximum contention obtained is $ubd - 1$ at every point in which δ is one cycle more than a multiple of *ubd*.

This phenomenon is better illustrated in Figure 5. We start from the scenario in Figure 5 a), in which we focus on an architecture with $\delta^{rsk} = 1$ and a request that suffers $\gamma(\delta^{rsk}) = 5$ cycles. In Figure 5 b)-d) we show the effect of increasing the number of *nop* operations between instructions generating consecutive requests, with $\delta^{nop} = 1$. In scenario Figure 5b), $\gamma(\delta^{rsk} + \delta^{nop})$ decreases down to 4 with respect to the original scenario depicted in Figure 5a). $\gamma(\delta)$ keeps decreasing as the number of *nop* operations injected, k, increases from 1 up to 5, see Figure 5c). Note that cases for $k = 2$, $k = 3$ and $k = 4$ have been omitted for space constraints. However, in the scenario d), when $k = 6$, we observe that the next request has to wait for all 3 other cores to proceed with their requests, thus increasing $\gamma(\delta)$ up to 5. It can be observed that, by varying the number of *nop* operations between requests, we can explore different alignment scenarios that appear due to the synchrony effect.

Overall, when $\delta^{rsk} > 0$, the maximum contention that requests can suffer as k varies is $ubd - 1$, as shown in Equation 2. The contention reaches *ubd* only when the injection time among any of the bus-accessing instructions is zero. However, *the period of the saw-tooth is exactly the* ubd *value regardless of* δ^{rsk}. Hence, the exact value of *ubd* can be derived from the *saw-tooth period* of $\gamma(\delta)$ when varying k, and this holds true for any injection time.

4.2 rsk-nop application methodology

Our methodology to derive *ubd* requires carrying out several experiments using *rsk-nop* as *scua* and several *rsk* – the original ones without any *nop* operations between bus-accessing instructions – as contender *sc*.

rsk-nop, used as *scua*, can be parameterized by varying in an incremental way the number of *nop* operations, k, between bus-accessing requests, as well as the type of instructions used to access the bus: *rsk-nop*(t, k). In our target architecture the type of instructions that can be used are *store* or *load* instructions. By default we use load instructions for both the *rsk-nop*

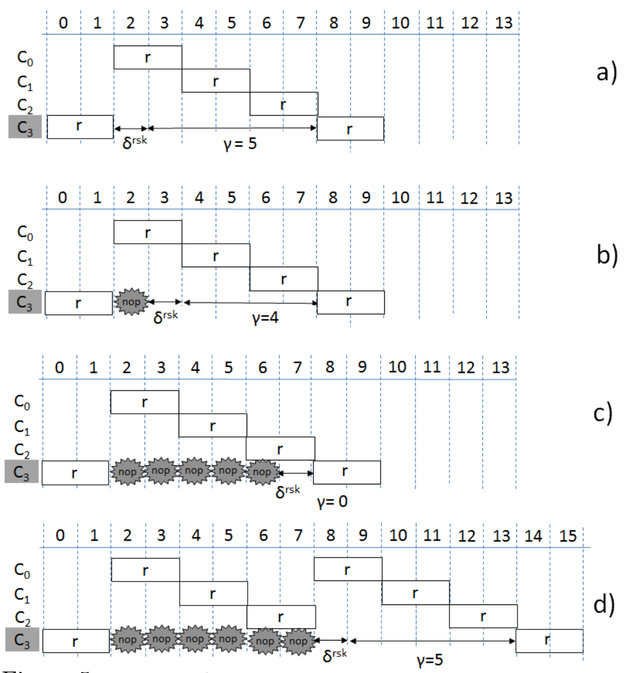


Figure 5: Timing of the scenario draw in Figure 3 as we add *nop* operations: a) 1 nop, b) 2 nop, c) 5 nop, d) 6 nop.

and the *rsk*. In Section 5 we show the issues of using stores due to the presence of store buffers in the pipeline.

rsk, used as contender *sc*, can be parameterized by varying the type of instruction, t, used to stress the resource, $sc(t)$.

We run *rsk-nop*(t, k) against $N_c - 1$ copies of *rsk*(t), recording its execution time, $et^{sc}_{scua}(t, k)$, and computing the execution time increase with respect to the execution time of *scua* in isolation, $d_{bus}(t, k) = et^{sc}_{scua}(t, k) - et^{isol}_{scua}(t, k)$. The observed $d_{bus}(t, k)$ has a saw-tooth behavior as we vary k and its period gives $ubd(t)$ for each type of access t. As expressed in the Formula 3, the execution time increase suffered for two different injection times, k_i and k_j, will be equal if $i - j = ubd$:

$$ubd(t) = |k_i - k_j| : (k_i \neq k_j) \text{ and } (d_{bus}(t, k_i) = d_{bus}(t, k_j)) \quad (3)$$

In the previous discussion we have assumed that $\delta_{nop} = 1$. This is typically the case in most architectures since nop operations do not have input/output dependencies and use the fast integer pipeline – if any. In the unlikely case $\delta_{nop} > 1$, varying the number of *nop*s will be equivalent to sampling the saw-tooth behavior presented in Figure 4. If the value of δ_{nop} can be derived, we can obtain the *saw-tooth period* easily. To this end, we have designed a *rsk* in which all the operations in the loop-body are nops. The loop body is made as big as possible without causing instruction cache misses. By dividing the execution time of such *rsk* by the number of nop operations executed we can derive δ_{nop} very accurately.

4.3 Summary

The proposed methodology empirically derives ubd_m requiring very limited knowledge about the underlying architecture (often available in the corresponding manuals).

Inputs: Our approach requires knowing that the bus arbitration policy is RR and the type of instructions that may generate requests to the bus. Both of which can be found in processors' manuals.

Confidence: Two elements are central to confidence on the obtained ubd_m. First, $N_c - 1$ cores running a *rsk* should suffice to increase the utilization of the bus to 100%, other than handshaking time. In many architectures, performance monitoring counter support exists to measure the bus utilization. For instance, counters $0x17$ and $0x18$ in the Cobham Gaisler NGMP provide per-core and overall bus utilization [3]. And second, it is required deriving δ^{nop} since it is needed to determine the *saw-tooth period*. As stated before, our simple *rsk* including only *nop* operations can be used to derive δ^{nop}.

Using ubd_m: The derived bound, ubd_m, can be used by STA by simply adding it to the access time to the bus [12]. With MBTA it is required to determine an upper-bound to the number bus requests, n_r, that the *scua* performs to the bus. The ETB of the *scua* is padded with $pad = n_r \times ubd_m$.

5. EVALUATION

First, we detail our experimental setup in Section 5.1. Then, we show how *rsk-nop* helps deriving *ubd* in the presence of the synchrony effect (Section 5.2). For this validation of the methodology we assume that the bus latency and the actual value of *ubd* are known. This information is not provided in Section 5.3, which represents the actual case of the applicability of the methodology to a COTS multicore.

5.1 Experimental Setup

We model a 4-core NGMP [2] running at 200MHz comprising a bus that connects cores to the L2 cache and an on-chip memory controller. Each core has its own private instruction (IL1) and data (DL1) caches. IL1 and DL1 are 16KB, 4-way with 32-byte lines. The shared second level (L2) cache is split among cores with each core receiving one way of the 256KB 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. With DRAMsim2 [20] we model a 2-GB one-rank DDR2-667 [10] with 4 banks, burst of 4 transfers and a 64-bit bus, which provides 32 bytes per access, i.e., a cache line. In a study with the European Space Agency we evaluated the performance estimates provided by our simulator against a real NGMP implementation, the N2X [3] evaluation board, using a low-overhead kernel that allowed cycle-level validation. Our results for EEMBC benchmarks showed a deviation in terms of accuracy of less than 3% on average and for the NIR HAWAII benchmark [8] the inaccuracy reduces to less than 1%.

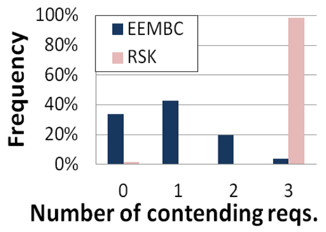
In order to show the robustness of the proposed methodology we evaluate it in this *reference architecture* as well as a *variant architecture* (labeled as *ref* and *var* respectively in following figures) in which DL1 and IL1 latency is 4 instead of 1 cycle, which increases the injection time of all bus-access instructions by 3 cycles, from 1 to 4. We show how our methodology based on *rsk-nop* manages to derive *ubd* on both setups.

For the evaluation we use the EEMBC Autobench suite [14], which models some real-world automotive critical functionalities. We also use *rsk* and *rsk-nop* as presented in previous Sections, which use load operations to access the bus.

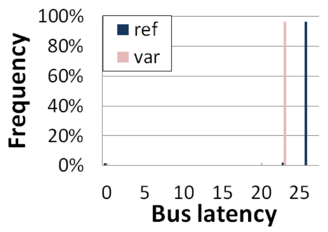
5.2 Observing the Synchrony Effect

For the purpose of showing how *rsk-nop* allows approximating *ubd* we use the following timing information: the bus has a maximum latency of 9 cycles per contender (6 cycles corresponding to the L2 hit latency and 3 cycles for bus transfer and arbitration handover). As a result *ubd* is 27 cycles in this case, see Equation 1.

In a first experiment, we run 8 randomly generated 4-tasks workloads with EEMBC benchmarks under the *ref* architecture. Figure 6(a) shows the histogram of the number of contenders ready to send a request when the EEMBC in core c_0 tries to access the bus. Results across different workloads are quite similar. As it can be seen (dark bars), the EEMBC in core c_0 finds the bus empty or with one contender most of the times when other EEMBC are used as contenders. This shows that with real workloads it is very difficult to obtain a worst-case scenario in which the contention suffered is *ubd*.

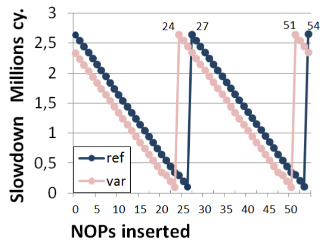


(a) Contending requests

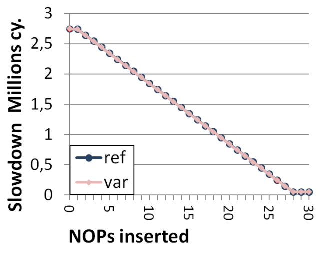


(b) Contention delay

Figure 6: Histogram of bus contenders and latency



(a) Load *rsk*.



(b) Store *rsk*.

Figure 7: Slowdown when executed *rsk-nop* as *scua* against 3 *rsk* co-runners. Results shown as as a function of nop instructions

In a second experiment we run 4 *rsk* that constantly access the bus. In this case, see light-gray bars in Figure 6(a), we observe that on almost every arbitration round the number of contenders is $N_c = 4$. *However this does not imply that the* ubd *is suffered by each request due to the synchrony effect.* This is so because a given c_i does not always have the lowest RR priority. For instance, it can happen that when c_i tries to access the bus three other contenders are also ready but the core holding the bus is c_{i-1}, so that c_i suffers low contention delay to get the bus.

We analyze this phenomenon in more detail by measuring the actual contention delay γ_i each request suffers. Figure 6(b) shows the histogram of the contention delay suffered by all requests of the *rsk* under the reference and the variant architecture. We observe that the synchrony effect makes that almost all requests in each case have the same latency since the injection time among requests is the same. Further, we observe that the distance among the observed upper-bound delay, i.e. ubd_m, and the actual one –27 in this case – varies across the two architecture: ubd_m is 23 for the *var* architecture and 26 for the *ref* one. Hence, depending on the the injection time in the underlying architecture, the accuracy of ubd_m varies, which prevents using *rsk* to accurately derive *ubd*.

We observe that most of the requests, 98% of them in Figure 6(b), have the same contention delay. This value depends on the number of load operations in the body of the *rsk*: the load operations in the boundary of loop iterations have a higher injection time than consecutive load operations inside the body due to the effect of loop-iteration control operations. In our case we unroll the loop body as much as possible not to cause instruction cache misses. This allows reducing the overhead to less than 2%.

5.3 Evaluation of rsk-nop methodology

As shown in Section 4, the injection time can be varied by inserting *nop* instructions between consecutive accesses of the *rsk* used as *scua* to derive *ubd*. In Figure 7(a) the vertical axis shows the slowdown (in millions of cycles) suffered by *rsk-nop* with respect to its execution in isolation and the horizontal axis represents the variation of γ in nops injected. As predicted in Figure 4, the slowdown is saw-tooth shaped, whose period is *ubd* $(27 = 51 - 24)$ for *var* and $(27 = 54 - 27)$ for the *ref*. Hence, the period of the saw-tooth shape is the same for both variant architectures, which evidences the robustness of the method detecting *ubd* under different setups. Note that slowdown results have been obtained reading the execution time that can be easily obtained in any COTS multicore.

So far we have used load operations in the *rsk* and *rsk-nop*. We can also use stores, having in mind that our reference architecture has a store buffer that keeps store requests and allows instructions to proceed in the pipeline unless the buffer is full, i.e. a store request is considered completed as soon as it is put in the buffer. The requests in the buffer access the bus with an injection time $\delta = 0$ since once the buffer is filled, requests can be issued in consecutive cycles. In a high occupation scenario of the buffer, store request suffer *ubd* in our scenario, i.e., one entry of the buffer is freed every *ubd* cycles. As δ increases (by inserting *nop* operations), the slowdown in the rsk-nop corresponds to the difference between the latency of a new empty slot in the buffer, i.e., *ubd*, and δ. When δ is higher than *ubd* the buffer is able to allocate an empty slot before a new request comes, thus the slowdown suffered is always zero because the buffer is effectively hiding the store latency. As it can be seen in Figure 7(b), this causes that for one entire period the slowdown has a saw-tooth shape, while for following periods, the slowdown is zero. We observe that the first period spans from $k \in [1,..,28]$, whose length matches the ubd. The one cycle shift in k is caused by the number of entries in the store buffer and its processing time.

6. RELATED WORK

Buses in real-time systems are used for off-chip and on-chip communication. Our work focuses on on-chip buses, such as the AMBA bus [7]. Deriving WCET estimates for various arbitration policies has been analyzed in the past including Round-Robin [13], TDMA [9] a similar policy to round-robin with groups [13] called MBBA [4], or even a comparison between arbitration policies [6]. In [17] authors propose a method based on Performance Monitoring Counters (PMC) to enable deriving WCET estimates with Measurement-based timing analysis, when the *ubd* for a round-robin bus is known. All these works assume knowledge about the bus timing: slot sizes or maximum transfer times. Our work assumes no knowledge about the timing of the bus.

In [18] authors report a counter intuitive behavior with a round-robin based multicore: the execution time of a task running against a given number of cores can be smaller than its execution time when running against fewer number of cores. Our work identifies the reasons behind this counter intuitive behavior, namely the synchrony effect behavior, and takes advantage of it to derive the *ubd*.

Resource-stressing kernels (*rsk*) [15], are used to characterize the contention on certain resources of a multithreaded architecture. They are also used in [5] to characterize the NGMP [2] or in [11] to characterize the Freescale P4080.

In [1], which analyzes the impact of resource sharing in multicore, authors criticize the confidence that one can obtain with *rsk*. We acknowledge the need to increase the confidence on the results provided with *rsk*, and in fact the focus of this paper is increasing confidence on those measurements for which we propose *rsk-nop*. The need to increase confidence with measurements is also confirmed by [12] in which the contention results obtained with micro-benchmarks in [11] for the P4080 are used as input to a commercial timing analysis tool.

7. CONCLUSIONS

The lack of information about internal processor timing behavior advocates for the use of measurements to derive those unknown timing parameters. For the bus, this parameter is the maximum contention delay a request can suffer when accessing the bus: *ubd*. We have proposed a measurement-based methodology that needs no information about the bus timing parameters to successfully derive *ubd*. Overall our methodology increases the trustworthiness on the derived ETB for COTS multicore processors deploying round-robin buses. Trustworthyness depends on both the soundness of the timing-analysis tool/technique and the input parameters given to the timing analysis tools, ubd_m in this case.

Acknowledgments

The research leading to this work has received funding from: the European Union's Horizon 2020 research and innovation programme under grant agreement No 644080(SAFURE); the European Space Agency under Contract 789.2013 and NPI Contract 40001102880; and COST Action IC1202, Timing Analysis On Code-Level (TACLe). This work has also been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557. Jaume Abella has been partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717.

8. REFERENCES

[1] A. Abel et al. Impact of resource sharing on performance and performance prediction: A survey. In *CONCUR*, 2013.
[2] Aeroflex Gaisler. *Quad Core LEON4 SPARC V8 Processor - Data Sheet and Users Manual*, 2011.
[3] Aeroflex Gaisler. *LEON4-N2X Data Sheet and User's Manual*, 2013.
[4] R. Bourgade et al. MBBA: A multi-bandwidth bus arbiter for hard real-time. In *Embedded and Multimedia Computing (EMC)*, 2010.
[5] M. Fernández et al. Assessing the suitability of the NGMP multi-core processor in the space domain. EMSOFT, 2012.
[6] J. Jalle et al. Deconstructing bus access control policies for real-time multicores. In *SIES*, 2013.
[7] J. Jalle et al. AHRB: A high-performance time-composable amba ahb bus. In *RTAS*, 2014.
[8] A. Jung et al. The h2rg infrared detector: introduction and results of data processing on different platforms. In *European Space Agency*, 2012.
[9] T. Kelter et al. Bus-aware multicore WCET analysis through TDMA offset bounds. *ECRTS*, 2011.
[10] Kingston. KVR667D2S5/2G Datasheet, 2011.
[11] J. Nowotsch et al. Leveraging multi-core computing architectures in avionics. In *EDCC*, 2012.
[12] J. Nowotsch et al. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS*, 2014.
[13] M. Paolieri et al. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009.
[14] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.
[15] P. Radojković et al. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM TACO*, 2012.
[16] E. Salminen et al. Benchmarking mesh and hierarchical bus networks in system-on-chip context. *Journal of Systems Architecture*, 2007.
[17] H. Shah et al. Measurement based WCET analysis for multi-core architectures. In *RTNS*, 2014.
[18] H. Shah et al. Timing anomalies in multi-core architectures due to the interference on the shared resources. In *ASP-DAC*, 2014.
[19] A. N. Udipi et al. Towards scalable, energy-efficient, bus-based on-chip networks. In *HPCA*, 2010.
[20] D. Wang et al. DRAMsim: a memory system simulator. *SIGARCH Comput. Archit. News*, 2005.