Adapting TDMA Arbitration for Measurement-Based Probabilistic Timing Analysis

Miloš Panić[*,†], Jaume Abella[†], Eduardo Quiñones[†], Carles Hernandez[†], Theo Ungerer[⋆], Francisco J. Cazorla[†,‡]

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

email: jaume.abella@bsc.es Telephone: +34 934137171 Fax: +34 934137721

***Abstract*—Critical Real-Time Embedded Systems require functional and timing validation to prove that they will perform their functionalities correctly and in time. For timing validation, a bound to the Worst-Case Execution Time (WCET) for each task is derived and passed as an input to the scheduling algorithm to ensure that tasks execute timely. Bounds to WCET can be derived with deterministic timing analysis (DTA) and probabilistic timing analysis (PTA), each of which relies upon certain predictability properties coming from the hardware/software platform beneath. In particular, specific hardware designs are needed for both DTA and PTA, which challenges their adoption by hardware vendors.**

This paper makes a step towards reconciling the hardware needs of DTA and PTA timing analyses to increase the likelihood of those hardware designs to be adopted by hardware vendors. In particular, we show how Time Division Multiple Access (TDMA), which has been regarded as one of the main DTA-compliant arbitration policies, can be used in the context of PTA and, in particular, of the industrially-friendly Measurement-Based PTA (MBPTA). We show how the execution time measurements taken as input for MBPTA need to be padded to obtain reliable and tight WCET estimates on top of TDMA-arbitrated hardware resources with no further hardware support. Our results show that TDMA delivers tighter WCET estimates than MBPTA-friendly arbitration policies, whereas MBPTA-friendly policies provide higher average performance. Thus, the best policy to choose depends on the particular needs of the end user.

***Index Terms*—worst-case execution time; processor design; arbitration policy; probabilistic analysis; time randomization**

I. Introduction

Timing analysis is a critical step in the development of real-time embedded systems, especially for those implementing some kind of safety- or mission-critical functionality. Timing analysis derives estimates to the Worst-Case Execution Time (WCET), which are combined by the task scheduler with other task information such as deadline, period and priority, to validate that the budget provided to each task is sufficient to satisfy the tasks' execution time needs.

Deterministic-Timing Analysis techniques [53], either static or measurement-based (SDTA and MBDTA), rely on architectures whose response time is deterministic to derive upper-bounds to the access time to each hardware resource. SDTA uses those bounds to create a timing model of the hardware. MBDTA enforces those bounds by means of hardware/software support [42], [4] when collecting the measurements used to estimate the WCET.

Probabilistic Timing Analysis (PTA) [6], [9], [13], [31], [5], which includes its static (SPTA) and measurement-based (MBPTA) variants, builds upon systems whose operation-phase timing behavior can be upper-bounded during the system's analysis phase, either deterministically or probabilistically. Deterministic upper-bounding of requests' access latency to hardware resources is performed similarly to DTA. Probabilistic bounding is performed for time-randomized resources [30] for which it can be associated a probability to each potential latency the resource can take to serve a request.

Both PTA and DTA are challenged by mixed-criticality applications running on multicores since the time it takes a request from a given task to be granted access to a resource depends on the load other co-running tasks put in that resource. Under DTA, this dependence is controlled by advocating for hardware support that isolates tasks against each other, e.g. using TDMA arbitration [25], or allows upper-bounding the maximum impact of contention, e.g. round-robin arbitration. Such isolation is a key enabler for mixed-criticality systems by preventing interferences across criticality levels. Under MBPTA[1], it is required that the impact of contention captured in the measurements taken during the analysis phase of the system upper-bounds, deterministically or probabilistically [30], the impact of contention that can occur during the operation phase of the system. In this line, round-robin arbitrated shared resources, usually deployed in the context of DTA, have also been proven analyzable with MBPTA [20]. However, this is not the case for TDMA arbitrated shared resources.

Contribution. In this paper, which extends our former publication in [40], we analyze in detail TDMA in the context of MBPTA and provide means to allow TDMA resources to be analyzed with MBPTA. This is of high relevance since TDMA arbitration fulfills isolation requirements coming from mixed-criticality applications. Furthermore, we show that TDMA

[1] In this paper we focus only on MBPTA since it has been shown to be closer to industrial practice for timing analysis than Static Probabilistic Timing Analysis (SPTA). Further, MBPTA has been already evaluated with avionics case studies [51].

allows obtaining tighter WCET estimates than round-robin by padding execution time once instead of padding the latency of each request. On the other hand, we show that average performance for TDMA is worse than for the other arbitration policies. To reach these objectives:

1) We analyze the timing characteristics of TDMA in the context of MBPTA from a theoretical perspective. We show that TDMA cannot be directly analyzed with MBPTA. The difficulty lies in the variable (i.e. jittery) nature of the delay that a request incurs to get access to the arbitrated resource and that a probability cannot be assigned to each specific delay value, thus failing to attain the properties required by MBPTA [30].
2) We show that the effect of TDMA on execution time is limited to the duration of a single TDMA window when there is a single TDMA-arbitrated resource for asynchronous requests, as already proven for synchronous ones in [25]. Also, we show that the effect of TDMA for several chained arbitrations is limited to the least-common-multiple of the TDMA windows.
3) Based on the previous appreciation of the limited impact of TDMA on execution time, we apply a simple modification to the application of MBPTA as a means to enable the analysis of TDMA resources. In particular, we *augment* the execution time observations collected when running the task of interest in the target system, which are used as input to MBPTA.
4) We identify how the use of TDMA in the context of MBPTA impacts timing anomalies. Our analysis shows that, if timing anomalies can occur, TDMA cannot trigger new types of timing anomalies but, instead, alter the number of anomalies triggered of the existing types.

Our analysis not only advances the limits on the arbitration policies that can be analyzed with MBPTA without requiring MBPTA-customized designs [20], but also helps promoting *one-design-fits-all* for arbitration policies. The latter makes that different timing analysis techniques are enabled on the same hardware. This increases the impact that the research on time-analyzable hardware may have on chip vendors to adopt such hardware in actual processor designs, hence, reaching the goal of having time-analyzable multicores. Our solution, based on padding, produces 9% lower WCET estimates for TDMA in comparison to the best MBPTA-specific arbitration policy on average. However, average performance for TDMA is 18% worse than that for MBPTA-specific arbitration policies. Therefore, in the context of MBPTA, TDMA can be regarded as the most convenient solution when most of the time budget needs to be devoted to tasks with hard real-time constraints. However, if a large fraction of the time budget needs to be devoted to tasks with soft real-time or non real-time constraints, then work-conserving arbitration policies such as, for instance, random permutations [20], are more convenient due to their higher average performance.

The rest of this paper is organized as follows. Section II introduces how contention is handled under DTA and MBPTA. Section III describes how TDMA impacts execution time. Section IV analyzes TDMA in the context of MBPTA and introduces how to enable its use. Section V analyzes the impact of timing anomalies when TDMA is used together with MBPTA. Section VI provides some results. Related work is presented in Section VII and conclusions in Section VIII.

II. Contention analysis for DTA and MBPTA

Since mid 90's, critical real-time embedded systems have progressively shifted towards an *integrated architecture* paradigm (e.g. IMA [2]) in domains such as avionics: a modular approach in which multiple functions are allocated to a single hardware unit. A key design principle is *incremental qualification*, whereby each component can be subject to qualification in isolation. At the timing level, this requires *time composability* so that the timing properties of a software component in isolation, i.e. its WCET estimate, do not change when the system is integrated. Time composability, therefore, reduces the cost of system development, integration and qualification. It is worth noting that this usually comes at the cost of some overestimation of the WCET.

The advent of multicores challenges achieving time composability, though. This occurs because the access latency to hardware shared resources becomes jittery. In particular, the access latency to a hardware shared resource includes the arbitration delay and the service latency. The former is the time a request spends to get access to the resource. The latter is the time that the request takes to be processed once it is granted access. Both of them may be impacted by (inter-task) contention, especially the arbitration delay. For multicore on-chip resources (the focus of this paper), we aim at providing time composability in the access latency for WCET estimation. Thus, the access latency has to be upper-bounded such that the load that other tasks put on that resource does not exceed the bound used for WCET estimation purposes. This not only prevents avoiding interferences across tasks with mixed criticalities in the timing domain, but also enables incremental development and qualification.

A. SDTA and MBDTA

SDTA [53], [36] abstracts a model of the hardware, which is fed by a representation of the application code, to derive a single WCET estimate. On the contrary, MBDTA [53] makes extensive testing on the target system with stressful, high-coverage input data. The longest observed execution time across all tests is recorded and an engineering margin is added to make safety allowances for the unknown. This engineering margin is extremely difficult to determine in the general case.

Under SDTA, reliable WCET estimates can be attained in the presence of contention by different means:

1) At analysis time, requests are assumed to experience always the worst-case latency in the access to the shared resource [19]. For instance, with round-robin, SDTA assumes that, whenever the request becomes ready, it has the lowest arbitration priority, so it has to wait for all other cores to be arbitrated before getting

access. As analysis-time latencies upper-bound operation ones, the execution time derived at analysis time for the program upper-bounds the impact of the shared resource. More sophisticated methods allow assuming lower contention latencies by analyzing the abstracts states during the timing analysis [18]. Note that, with MBDTA, it is not assumed that requests suffer an upper-bound contention latency but, instead, this is enforced by a specific hardware mechanism [42] making each request be delayed as if it was experiencing the highest contention possible[2].

2) Alternatively, at analysis time, each request is assumed to suffer a fixed impact on its duration. This approach is used by SDTA when applied to TDMA-arbitrated resources by determining the alignment of each request w.r.t. the TDMA window and hence, the delay it suffers until its next available slot.
3) With SDTA, it is possible to carry out a combined timing analysis of all the tasks simultaneously running in the multicore [24]. This may reduce the impact of contention on WCET estimates since only the actual contention generated by the co-running tasks is considered. However, it comes at the cost of losing time composability since any change in the tasks in the workload requires reanalyzing all the tasks in it.

B. MBPTA

MBPTA derives a distribution, called probabilistic WCET or pWCET, that associates a probability of exceedance to each WCET value. The exceedance probability, which upper-bounds the probability that a single run of the task exceeds its WCET budget, can be set arbitrarily low in accordance with the requirements of the corresponding safety standard.

MBPTA reaches this goal by relying on end-to-end measurements taken on the platform to derive a WCET distribution, rather than a single WCET estimate per task, as it is the case for SDTA. MBPTA requires understanding and controlling the nature of the different *contributors* to the execution time of a program [10]. These contributors, also known as sources of execution time variability (*setv*), include (i) the initial conditions of hardware and software (e.g. cache state), (ii) those functional units with input-dependent latency (e.g. an integer divider), (iii) the particular addresses where memory objects are placed, (iv) the number of contenders in the access to shared resources, and (v) the execution paths of the program. MBPTA requires that the jitter, a.k.a. execution time variability, of all *setv* captured in the end-to-end execution times collected at analysis time upper-bounds the jitter of each *setv* when the system is deployed (*operation phase*). In [30] it is explained how upper-bounding these *setv* enables collecting execution time observations that can be regarded as independent and identically distributed, as required by MBPTA [13].

Jitter can be upper-bounded *deterministically* [30] by enforcing *setv* to experience a single latency at analysis time lat_{det}^{an} that upper-bounds any latency that the *setv* may take at operation, $lat_{det}^{op,i}$. That is, $\forall i : lat_{det}^{an} \geq lat_{det}^{op,i}$. For instance, enforcing functional units with input-dependent latencies to operate at their highest latency during the analysis phase leads to deterministic upper-bounding, as their latency at analysis time is constant. During operation, real latencies will be equal or lower than those at analysis time.

Jitter can also be upper-bounded *probabilistically* [30] by enforcing the latencies of a *setv* to have a probabilistic distribution at analysis time such that, for any exceedance probability (e.g. 10^{-3}), the latency at analysis time is equal or higher than that of the distribution during operation. For instance, let us assume a random-permutations arbitrated bus [20] shared by N_c cores (with random permutations, on every arbitration window a random permutation of the slots is created so that in every window the contenders access the bus in a random fashion [20]). Further, let us assume that, during operation, the bus is arbitrated only across all cores with pending requests, which are a subset of all N_c cores. In this scenario, the analysis-time delay distribution experienced due to contention upper-bounds that during operation if, at analysis time, arbitration always occurs across N_c cores. This upper-bounding is probabilistic since such delay is not a fixed value but a distribution. Table I(a) shows the probability of getting the bus in a given round under different contender (core) counts, while Table I(b) shows the accumulated probability, that is, the

TABLE I
RANDOM ARBITRATION BUS EXAMPLE.

Number of rounds	contenders 4	3	2
1	0,2500	0,3333	0,5000
2	0,2344	0,2963	0,3750
3	0,2031	0,2222	0,1250
4	0,1563	0,1111	0,0000
5	0,0938	0,0370	0,0000
6	0,0469	0,0000	0,0000
7	0,0156	0,0000	0,0000
8	0,0000	0,0000	0,0000

(a) Probability of getting the bus in a given round

Number of rounds	contenders 4	3	2
1	0,2500	0,3333	0,5000
2	0,4844	0,6296	0,8750
3	0,6875	0,8519	1,0000
4	0,8438	0,9630	1,0000
5	0,9375	1,0000	1,0000
6	0,9844	1,0000	1,0000
7	1,0000	1,0000	1,0000
8	1,0000	1,0000	1,0000

(b) Accumulated prob. of getting the bus in the first X rounds

[2]Without hardware support, measurements need to capture high contention scenarios, but reliability of the WCET estimates is hard to be proven.

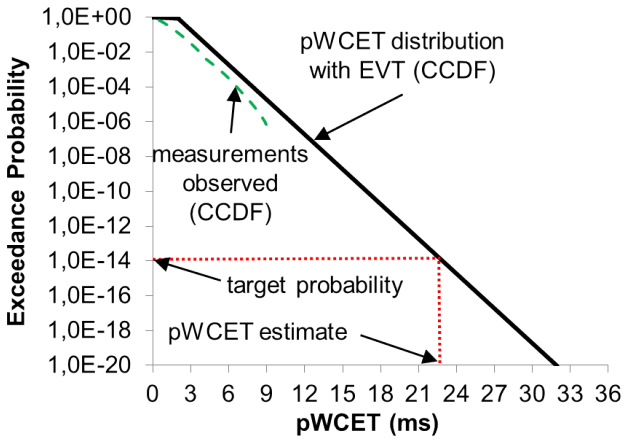

Fig. 1. Example of application of MBPTA for a program with $R = 1,000$. Measurements are taken during the analysis phase. The pWCET distribution derived with EVT holds at operation. In the X-axis ms stands for milliseconds.

probability of getting the bus in any of the first X rounds[3]. We observe that, when all $N_c = 4$ cores are assumed active, as it is the case at analysis time, the accumulated probability of getting the bus is smaller than when the number of cores is 3 or 2. Hence, given that during operation the number of active cores is at most 4, the analysis time contention distribution upper-bounds that experienced during operation, rendering this arbitration policy as MBPTA-analyzable.

Once the requirements of MBPTA are fulfilled, a number (R) of execution time measurements of the program are collected – R is typically in the order of few hundreds or thousands. Then, those R measurements are tested for independence and identical distribution so as to enable the use of Extreme Value Theory (EVT) [32][4]. EVT is a powerful statistical tool to predict the tail of a distribution based on a sample. In the context of MBPTA, EVT is used to predict the distribution of the high execution times (so the pWCET). Execution time measurements are also tested for exponentiality of their uppermost tail so that the pWCET can be approximated with a Gumbel distribution, which has been regarded the appropriate (and very convenient) distribution to model the pWCET [13]. Finally, MBPTA is applied on the R measurements collected. An example of application of MBPTA on a synthetic program is shown in Figure 1, where we depict the complementary cumulative distribution function (CCDF) – also known as tail distribution – of the R measurements (1,000 in the example) as well as the pWCET distribution produced by MBPTA. As explained before, the pWCET value to be used is the one at the desired exceedance probability (10^{-14} per run in the example).

III. TDMA IMPACT ON EXECUTION TIME

TDMA ensures that the load a task puts on shared resources does not affect the WCET of its co-runners [19], thus isolating tasks with different criticality levels. In this section, we make a detailed analysis of TDMA impact on the timing behavior of the application. Without loss of generality, we focus on a bus as the resource arbitrated with TDMA.

We assume canonical TDMA so that it splits time into windows of size w cycles, each of which is further divided into slots of size s. Each bus contender (processor cores in our case) is assigned one such slot in a cyclic fashion. During a given slot, only its owner can send requests. Request duration cannot exceed the slot duration. When a contender has no pending requests, the bus remains idle for that slot even if there are pending requests from other contenders (non-work-conserving approach). We call *tdma-relative cycle* or simply *relative cycle* (cyc_i^{rel}) the cycle in which a request, r_i, becomes ready within the TDMA window. It can be computed as shown in Equation 1, where cyc_i^{abs} stands for the absolute execution cycle.

$$cyc_i^{rel} = cyc_i^{abs} \mod w \tag{1}$$

A. Request Types

We consider a timing-anomaly free architecture [35], [52], [45], [23]. A discussion on the impact of timing anomalies is provided in Section V. A number of definitions have been provided for timing anomalies. In our case, a processor architecture free of timing anomalies refers to an architecture for which an increase in the access latency of a request (belonging to the program) to any resource, e.g. due to contention, cannot result in lower program's execution time.

We consider both synchronous and asynchronous requests. Synchronous requests are blocking. This means that they stall the corresponding pipeline stage until served. In our reference architecture this is the case of load operations that miss in first level (L1) caches and access the second level cache (L2).

Asynchronous requests, instead, are kept in a buffer until served, not stalling any pipeline stage unless the buffer is full. This is the case, for instance, of those processors that do not stall the pipeline on a store (write) operation. Since no instruction in the core has to wait for the results of such write operation, the store operation is put in a store-buffer, which sends the request to the data cache afterwards. The store operation is considered as committed (serviced) when it is sent to the store-buffer. However, the write request may take a variable number of cycles to access the bus. This creates asynchronous accesses to the bus.

Split transactions are used when the target resource for the request, L2 in our case, takes long to answer (e.g. ARM AMBA bus [1] implements them). Instead of holding the bus for tens of cycles, the L2 answers the request with a 'split transaction' command, allowing the other contenders to use the bus while L2 processes the request in background.

B. TDMA impact on execution time for synchronous request

The slot alignment delay (*sad*) for each request defines the time the request has to wait for its slot in a TDMA window so it can be granted access. In the worst case, a request becomes

[3]Note that random permutations works similarly to TDMA but sorting slots randomly within each window. Thus, the maximum arbitration delay is always below two TDMA windows.

[4]In some cases independence is not strictly needed if maxima are independent or if the dependence is weak [12], [47].

ready one cycle after its slot expires making it wait sad_{tdma} cycles, as defined in Equation 2.

$$sad_{tdma} = (Nc - 1) \times s \quad (2)$$

Note that, without loss of generality and for the sake of simplifying formulation, we have assumed that the access time of a request is one cycle. In the general case, assuming a request latency lat_r, the worst scenario occurs when it becomes ready during its slot $lat_r - 1$ cycles before it elapses, making the request wait $sad_{tdma-gen}$ cycles, as defined in Equation 3.

$$sad_{tdma-gen} = (Nc - 1) \times s + lat_r - 1 \quad (3)$$

The particular *sad* of a request may make it be served right away (so 0 delay) or delayed by up to $(Nc - 1) \times s + lat_r - 1$ cycles. Given that $lat_r \leq s$ so that the request fits in the slot, and assuming the worst case where $lat_r = s$, $sad_{tdma-gen} = w-1$. Therefore, given a program with a single synchronous request r_i, the execution time of the program can vary up to $w - 1$ cycles depending on how r_i aligns with the TDMA window as already shown in [25]. Further, if multiple synchronous requests exist in the program, the execution time variation that the TDMA resource can introduce is still up to $w-1$ cycles as proven in [25]. The intuition behind this effect lies on the fact that a particular *sad* achieves the fastest execution time across the w different *sad* (w different alignments w.r.t. the TDMA window). Under any other *sad*, the program only needs to be stalled by up to $w - 1$ cycles to align with the TDMA window as the fastest *sad*, and execute identically from that point onwards. We refer the interested reader to the work by Kelter et al. [25] for a formal proof.

Let us assume that the program under analysis has a single request r_i to the bus during its whole execution. That request may suffer a slot alignment delay sad_i which can take values within the range: $sad_v = \{0, 1, ..., (Nc-1) \times s\}$. For instance, for a 4-core scenario in which cores access a *TDMA* resource with $s = 2$, and hence $w = 8$, $sad_v = \{6, 5, 4, 3, 2, 1, 0\}$. The particular *sad* suffered by the request depends on the relative cycle in which the request becomes ready, cyc_i^{rel}, which can take w different values. In the particular example above, if the request arrives in any of the 2 cycles of its slot, i.e. $cyc_i^{rel} = 0$ or $cyc_i^{rel} = 1$, it is served immediately, hence $sad = 0$ in both cases. If it arrives in the $cyc_i^{rel} = 3$ then $sad = 6$, whereas if $cyc_i^{rel} = 4$ then $sad = 5$, and so on so forth. Overall, we observe that each value of cyc_i^{rel} affects sad_i and hence program execution time.

C. sad *for Multiple Synchronous Requests*

Let us now assume that the program under analysis, *P*, has several requests to the bus: $R_P = \{r_0, r_1, ..., r_n\}$. Let δ_i^{inj} be the injection delay between a preceding instruction generating request r_{i-1} and the instruction generating request r_i. The injection delay can be measured as the time elapsed since r_{i-1} is fetched into the processor until r_i is fetched. Hence, for the program *P* we have $\Delta_P^{inj} = \{-, \delta_1^{inj}, ..., \delta_n^{inj}\}$.

If the requests generated by those instructions are synchronous (and other pipeline stages do not create stalls), the delay since a request r_{i-1} is served until the next request r_i accesses the bus is fixed, that is δ_i^{bus} is fixed. The inter-request delay for all requests is defined as: $\Delta_P^{bus} = \{-, \delta_1^{bus}, ..., \delta_n^{bus}\}$.

The relative cycle in which the first request becomes ready, cyc_0^{rel}, defines a *sad* scenario in which the alignment that the subsequent requests, $r_i : i > 0$, suffer may vary. This is better illustrated with the example in Figure 2, in which there are 5 requests with $\Delta^{inj} = \{-, 1, 3, 2, 1\}$. In the figure, we use d_i to refer to the cycles in which the request r_i is ready but delayed due to *sad*. We use s_i to refer to the cycle in which the request r_i is served. Each row represents a different cyc_0^{rel} scenario in which r_0 becomes ready in a different relative cycle. Requests can only be served during slots $s0$ (see second row). In the first cyc_0^{rel} scenario, r_0 is ready in cycle 0 and served immediately. r_1 gets ready one cycle later, in cycle 1 (second element in Δ^{inj}), and is served in cycle 1, still during slot $s0$ of window $w0$. r_2 becomes ready 3 cycles after r_1 is served (see third element in Δ^{inj}), so in cycle 4, but has to wait for the next slot $s0$ until cycle 8. r_3 becomes ready 2 cycles after, so in cycle 10, and has to wait until cycle 16 for the next $s0$ slot. Finally r_4, that becomes ready right after r_3, so in cycle 17, is served immediately. Overall, in this first *sad* all requests are served after 18 cycles (from cycle 0 till cycle 17). In the second *sad* r_0 is ready in cycle 1 and served immediately. r_1 gets ready in cycle 2, but has to wait then until cycle 8. r_2 gets ready 3 cycles later, in cycle 11, and has to wait for slot $s0$ until cycle 16. r_3 gets ready in cycle 18, so it has to wait until cycle 24 to be served. Then r_4 gets ready in cycle 25 and is served immediately. Thus, under this *sad*, all requests are served after 25 cycles (from cycle 1 till 26). Overall, each different *sad* takes 18, 25, 24, 23, 22, 21, 20 and 19 cycles respectively.

We observe different sad for all requests that are determined by cyc_0^{rel}. Each of the cyc_0^{rel} different w scenarios determines the *sad* experienced by the remaining requests, leading to a maximum of w execution times for the program.

Observation 1: *The impact of* sad *on a program with different synchronous requests is determined by* cyc_0^{rel}. *Under each different* cyc_0^{rel} *scenario, the* sad *for each request – and hence the impact on the program's execution time – may vary.*

As shown in the previous example, the execution time difference between the fastest and the slowest scenarios is exactly $w - 1$ cycles. In the example, the first scenario takes 18 cycles (from cycle 0 until cycle 17), whereas the second scenario takes 25 cycles (from cycle 1 until cycle 25). This occurs because, in the slowest of both scenarios, it may happen that a given request, r_i, cannot be served as quickly as in the fastest scenario and it experiences some delay. However, such delay is strictly smaller than w, as this is the longest time it may take r_i to reach its next slot. Eventually, this request will align with its slot identically as in the fastest case in up to $w - 1$ cycles, and beyond that point its execution will be identical to that of the fastest case. In this case, we say that both *sad* scenarios become synchronous after r_i.

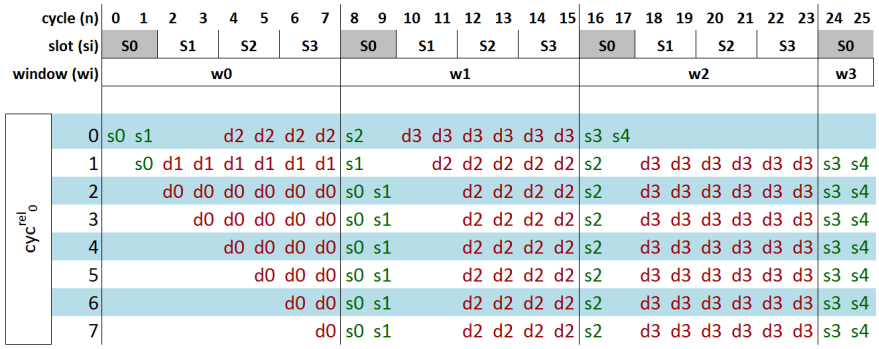

Fig. 2. Example of 5 requests with $\Delta^{inj} = \{-, 1, 3, 2, 1\}$ and their *sad*. d_i are the cycles in which the request is ready but blocked due to *sad*; s_i represents cycle in which the request gets access to the bus. Finally blanks represent the cycles with no pending requests.

Definition 1: *Two* sad *scenarios, sad_a and sad_b, are synchronous starting from a given request r_i when the relative cycles when that request is served is the same under both scenarios, i.e. $cyc_i^{rel,sad_1} = cyc_i^{rel,sad_2}$, regardless of the particular TDMA window in which the request is served. In this scenario, all subsequent requests after r_i suffer the same* sad *under both scenarios.*

Hence, the maximum execution time impact between the different *sad* that a program with synchronous requests may suffer is strictly smaller than a TDMA window (w cycles): up to $w - 1$ cycles to synchronize and identical execution time after synchronizing.

It may also be the case that, in the slowest scenario, r_i is served in the next TDMA window at the beginning of the slot, whereas it is executed at a later relative cycle in the fastest scenario, hence having a lower relative cycle in the slowest scenario than in the fastest one. Eventually, a subsequent request, r_j where $j > i$ may be further delayed until it fully synchronizes with the fastest scenario. In any case, the accumulated delay of all those requests can only be at most $w - 1$ cycles, as this is the maximum time to align their slots identically as in the fastest case.

Observation 2: *The maximum execution time impact between the different* sad *that a program with synchronous requests may suffer is strictly smaller than a TDMA window (w cycles). The execution time difference among two particular r_0 TDMA alignments, i.e. cyc_0^{rel}, is up to $w - 1$ cycles.*

D. sad *for Multiple Asynchronous Requests*

In the case of synchronous requests, the time between requests accessing the bus is fixed, regardless of the particular *sad* each of them suffers. However, this is not the case for asynchronous requests (e.g. stores). Let δ_i^{inj} be the injection delay between a preceding instruction generating request r_{i-1} and the instruction generating request r_i. The injection delay can be measured as the time elapsed since r_{i-1} is fetched into the processor until r_i is fetched. Hence, a program *P* with $n+1$ requests can be represented as $\Delta_P^{inj} = \{-, \delta_1^{inj}, ..., \delta_n^{inj}\}$. If the injection delay is fixed, Δ_P^{inj} for the store operations in P, the access time of those requests to the bus, and hence the time among them, Δ_P^{bus}, may vary depending on the *sad* scenario.

In order to illustrate this scenario we assume a program with $\Delta_P^{inj} = \{-, 4, 1\}$ in which all operations are stores. Stores are sent to a 2-entry store buffer from where they access a TDMA-arbitrated bus. Figure 3 shows the timing of the different requests depending on the relative ready cycle of r_0. Note that requests are considered as completed once they are sent to the store buffer. For instance, in the first scenario ($cyc_0^{rel} = 1$, so the first shaded row), r_0 becomes ready in cycle 0 in which it is buffered (b_0) and it is served in cycle 1 (s_0). r_1 becomes ready 4 cycles after that, and it is put in the buffer b_1 until the next slot for the core starts in cycle 8. Once r_1 is in the buffer, in cycle 4, it is considered completed, so in cycle 5 r_2 is processed, i.e. also sent to the buffer b_2. Once the slot for this core starts in the second TDMA window, r_1 and r_2 are served consecutively in cycles 8 and 9. Thus, it takes 10 cycles to send all requests (from cycle 0 till 9). In the second scenario ($cyc_0^{rel} = 2$) r_0 enters the store buffer in cycle 1 and cannot be sent to the bus in cycle 2 because the $S0$ slot has elapsed. r_1 is queued in cycle 5 and the store buffer is full. Thus, although r_2 gets ready in cycle 6, it cannot enter the buffer until an entry is released, which occurs in cycle 8 when r_0 is sent to the bus. Then r_1 is sent in cycle 9 and r_2 has to wait until cycle 16 to be granted access to the bus. Thus, it takes 16 cycles to send all requests (from cycle 1 till 16). Overall, each different *sad* takes 10, 16, 15, 14, 13, 13, 12 and 11 cycles respectively.

In Figure 3 we observe that the number of different *sad* scenarios, impacting both the *sad* of the different requests and the program execution time, is limited to $w - 1$. This leads to the following observation.

Observation 3: *The impact of* sad *on a program with different asynchronous requests is determined by cyc_0^{rel}. Under each different cyc_0^{rel} scenario, the* sad *for each request – and hence the impact on the program's execution time – may vary.*

As with synchronous requests, the execution time difference between different *sad* can only be up to $w - 1$ cycles. To illustrate it, let us take as a reference the scenario executing the fastest (e.g. first scenario in Figure 3) and any other arbitrary *sad scenario*. Let $shift^{sad}$ be the cycle count difference

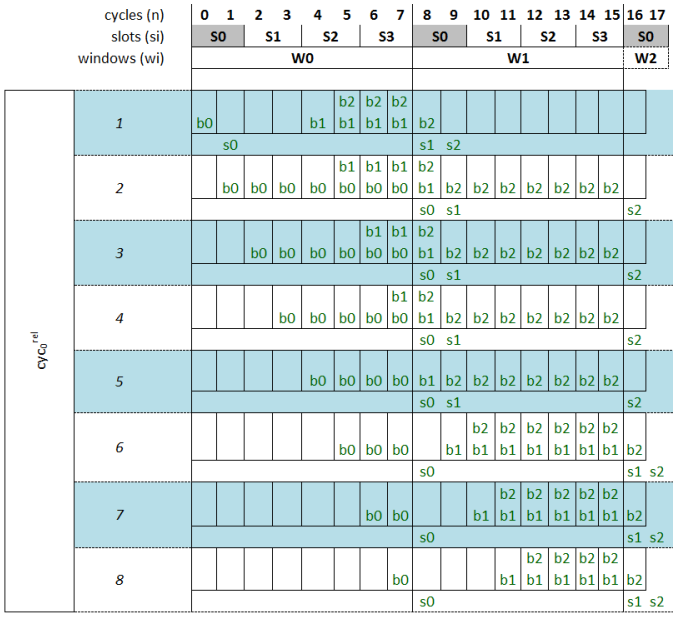

Fig. 3. Example of 3 requests with $\Delta^{inj} = \{-, 4, 1\}$ and their *sad*. b_i are the cycles in which the request is ready but waiting in the buffer due to *sad*; s_i represents cycle in which the request gets access to the bus. Finally blanks represent the cycles with no requests on the bus.

between both scenarios – $sad_{fastest}$ and sad_{slow} – such that sad_{slow} synchronizes with $sad_{fastest}$ as described in Section III-B. By construction, $shift^{sad} < w$ given that there are w different *sad* where the $shift^{sad}$ for the $w-1$ slowest ones w.r.t. the fastest one is 1, 2,... $w-1$ cycles respectively. Eventually, in sad_{slow} requests can wait $shift^{sad}$ cycles and execute identically as in $sad_{fastest}$, or it may be the case that they execute faster because during those $shift^{sad}$ cycles some requests find an available slot. This reasoning applies to each request individually given that, although they are injected synchronously (e.g. instructions are fetched synchronously), they access the bus asynchronously due to some buffering mechanism (e.g. requests are buffered in the store buffer without stalling the fetch stage). Thus, given that all requests can be served as in the fastest case if they get delayed by $shift^{sad}$ cycles, the execution time would be increased by $shift^{sad}$ at most, where $shift^{sad} < w$. If any request is served earlier, this cannot increase the execution time further because we rely on a processor free of timing anomalies. Hence, sad_{slow} can only take up to $shift^{sad} < w$ more cycles than $sad_{fastest}$.

Observation 4: *The maximum execution time impact between the different* sad *that a program with asynchronous requests may suffer is strictly smaller than a TDMA window (w cycles). The execution time difference among two particular r_0 TDMA alignments, i.e. cyc_0^{rel}, is up to $w-1$ cycles.*

E. Multiple TDMA resources

When several TDMA-arbitrated resources are used in the system (e.g., k TDMA resources), with each resource i with its own TDMA window w_i, at most $lcm(w_1, w_2, ..., w_k)$ different

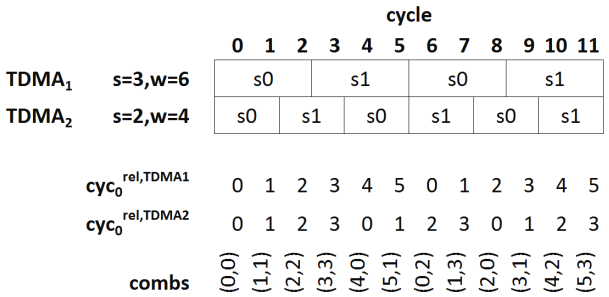

Fig. 4. Different combinations – in a two TDMA-window case – for $cyc_0^{rel,TDMA1}$ and $cyc_0^{rel,TDMA2}$.

sad scenarios across all k TDMA resources exist, where lcm stands for the *least common multiple*.

The key factor in determining the impact of crossing k TDMA resources is the relative cycle in which the first request, r_0, becomes ready across all TDMA windows. Hence, for the case of two TDMA windows, there are a total of $lcm(w_1, w_2)$ combinations of $cyc_0^{rel,tdma1}$ and $cyc_0^{rel,tdma2}$. For instance, in Figure 4 we have an example with two TDMA resources, each one with two slots. Slots in the first and second TDMA resource have 3 and 2 cycles respectively. Thus, $w_{tdma1} = 6$ and $w_{tdma2} = 4$. This leads to a total of $lcm(w_{tdma1}, w_{tdma2}) = 12$ different *sad*, shown in the last row. In this case, we consider all those 12 *sad* scenarios. Based on the arguments given before, the execution time in the worst *sad* scenario is at most 11 cycles worse (slower) than in the best *sad*, as this is the longest time needed to align the slots across both TDMA resources. Thus, the same rationale used for a single TDMA resource can be applied in this case.

Observation 5: *When multiple TDMA resources are used, those TDMA resources can create execution time variations of up to $lcm(w_1, w_2, ..., w_k) - 1$ cycles due to* sad.

F. Other considerations

Split Requests. As explained before, some requests to the bus are split. For example, a L1 cache miss may require a split request to access first the L2 cache, get a response indicating it misses in L2, and some time later get the data back with the second part of the request that has been split. In any case these two requests originated by the split mechanism are either synchronous or asynchronous and the same observations presented in previous sections for independent synchronous and asynchronous requests apply in this case.

Variable injection rate. Let $fc(I_{ri})$ be the cycle in which the instruction generating a request to the bus is fetched. So far, in our discussion, we have assumed a fixed injection rate across *sad* scenarios. That is, $\delta_i^{inj} = fc(I_{ri}) - fc(I_{ri-1})$ is the same for any consecutive pair of instructions under any two *sad* scenarios. In reality, however, if under some scenarios the instructions between I_{r_i} and $I_{r_{i-1}}$, block the pipeline after the execution of $I_{r_{i-1}}$ such that I_{r_i} cannot be fetched, then δ_i^{inj} varies across *sad* scenarios. However, δ_i^{inj} is determined by a combination of synchronous and asynchronous events: pipeline

stalls bring the synchronous component whereas buffering capabilities of the pipeline bring the asynchronous component. Hence, delaying timing events by at most the cycles between the current *sad* and the one leading to the fastest execution is enough to have the same execution behavior from that point onwards. Anything occurring with a delay shorter than that cannot lead to a longer execution time in a processor free of timing anomalies. Overall, the maximum impact on execution time of TDMA is limited to $lcm(w_1, w_2, ..., w_k) - 1$ cycles.

IV. TDMA IN THE CONTEXT OF MBPTA

In this section, we show how TDMA affects WCET estimation under MBPTA. We start by introducing the particular timing characteristics of MBPTA-compliant processors.

A. Timing of MBPTA-Compliant Processors

DTA-compliant processors experience deterministic latencies in the different resources and hence, execution time can be regarded as deterministic given a set of initial conditions. This occurs because each event leads to a single (deterministic) outcome and so, a single processor state can be reached. This is not the case for MBPTA-compliant processors, in which a number of random events may alter the execution time, thus leading to a different number of states, each of which is reached with a given probability as shown in [27]. We refer to those states as *probabilistic processor states*.

We illustrate, through a synthetic example, how those different states influence the latency between different bus requests. We consider a processor in which instructions take a fixed latency and where memory operations are all loads. Load operations access a time-randomized data cache [28], which is the only source of execution time variability (the instruction cache is assumed perfect)[5]. The total latency of a load that misses in cache, in the absence of any contention, is 100 cycles: 1 cycle to access cache, 1 cycle to traverse the bus and 98 cycles to fetch data. Note that, in this simple example, we assume no contention to send data from memory to the core. In this first experiment, we also consider that, whenever a load misses in cache, main memory is reached through a bus that *creates no contention*. Let us assume that the program under analysis has the following sequence of instructions $I = \{ld_0, i_0, i_1, ld_1, i_2, ld_2, i_3, i_4, i_5, i_6, i_7, ld_3\}$. Further assume that ld_0 always misses in cache and the other three load operations — ld_1, ld_2 and ld_3 — have an associated hit probability of 75%, although the actual value of those probabilities is irrelevant for the example. Other core instructions — i_0, i_1,... i_7 — do not access the data cache and have a fixed 1-cycle latency.

In this architecture, load operations generate a new probabilistic state in the execution, as shown in Figure 5. Every access leads to two possible probabilistic states (hit or miss), each with an associated probability. In that respect, there is a probability for each of the 8 possible combinations of hit-miss outcomes

[5]These assumptions simplify the discussion in this section. In Section VI we consider a multicore processor with time-randomized data and instruction caches.

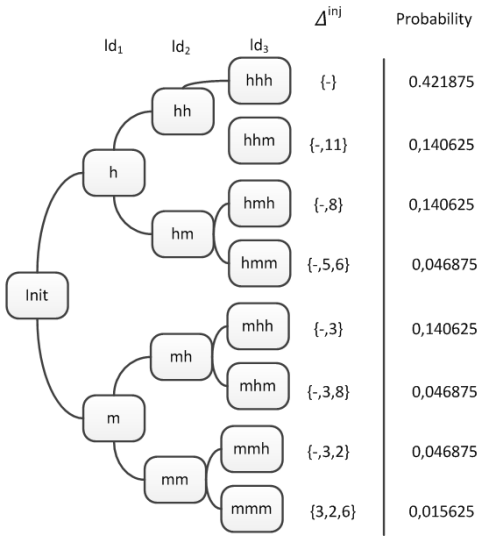

Fig. 5. Different probabilistic states in which the processor may be after the execution of each of the 3 loads in the example.

of the 3 load instructions ($hhh, hhm, hmm, ..., mmm$), which can be easily derived (e.g. $0.75 \cdot 0.25 \cdot 0.75 = 0.140625$ for the hmh case). Interestingly, any execution of the program leads to a single state out of those 8 probabilistic processor states, and for each of them, the delay among requests is fixed. Moreover, each such state (and set of delays among requests) occurs with a given probability. For instance, for the sequence mhm, which occurs with a probability of 0.046875, $\Delta^{inj} = \{-, 3, 8\}$ since 3 cycles elapse between ld_0 and ld_1, in which i_0 and i_1 are executed, and ld_1 requires an extra cycle to access cache. Analogously, 8 cycles elapse between ld_1 and ld_3 to execute seven 1-cycle instructions before ld_3 accesses cache.

In a second experiment, instead of assuming a no-contention bus, we assume TDMA arbitration for the bus that is shared among 4 cores. For TDMA, the slot for each core is $s = 2$ cycles with windows of $w = 8$ cycles. The execution time of the program under each probabilistic state is affected by the bus contention. Hence, the observations made in Section III for the impact of TDMA on execution time are to be considered for *each probabilistic state* in a MBPTA-compliant processor.

B. TDMA analysis with MBPTA

As explained in Section III-B, a shared resource implementing a TDMA arbitration policy may introduce execution time variations of up to $w - 1$ cycles, where w is the window size. From the point of view of MBPTA, the *sad* suffered by each request is, indeed, a *setv*. Hence, *sad* for TDMA is ruled by the same principles as other *setv*: its jitter has to be upper-bounded deterministically or probabilistically.

Observation 6: *In the absence of MBPTA-specific support, by default TDMA is not analyzable with MBPTA because one cannot prove that the delay experienced by each request (and hence the whole program) at analysis time, due to the alignment*

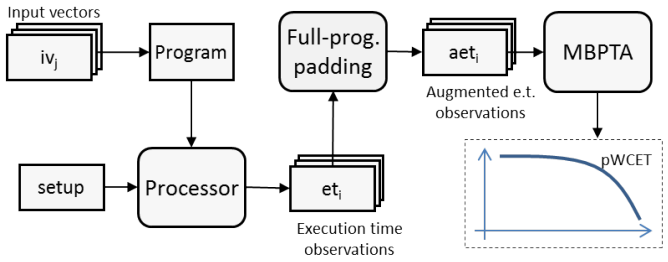

Fig. 6. Full-program padding in the context of MBPTA.

with the TDMA slots, upper-bounds the impact of TDMA during operation.

In the case of MBPTA, we have shown that each probabilistic state leads to a different Δ^{inj}, thus making the impact of the TDMA slot alignment different for each such states. Intuitively, one should consider the TDMA *sad* alignment individually for each probabilistic state to account for TDMA impact in execution time. However, this may be overly expensive since the number of probabilistic states grows exponentially with the number of probabilistic events [27], [5]. A different approach is needed to account for TDMA impact on execution time and pWCET estimates.

C. Per-request padding

One possible and intuitive solution to deal with TDMA jitter, i.e. *sad* alignment, is to assume, during the analysis, that each request suffers its worst *sad* delay (see Equation 2). Under MBPTA, this would mean having a hardware mechanism, such as the worst-case mode [42] so that, during the analysis phase, the program is run in isolation and each of its requests is artificially delayed by sad_{tdma} cycles. Given that sad_{tdma} is higher than any *sad* a request can suffer, the execution time observations collected at analysis time upper-bound the impact of the jitter caused by *TDMA*.

In essence, this approach consists in *deterministically* upper-bounding *sad* per request. However, it has been shown that assuming that each request suffers sad_{tdma} has an overly pessimistic impact on execution time [19]. Hence, this approach effectively enables the use of MBPTA with TDMA resources, but at the cost of pessimistically increasing pWCET estimates.

D. Full-program padding

We rely on the knowledge acquired in Section III on the maximum impact that TDMA can incur in the execution time of a program to propose a solution that has minimum impact on pWCET estimates. In particular, we show that the maximum impact that the alignment w.r.t. the TDMA window that a program can suffer is limited to w, so the maximum difference in execution time (i.e. jitter) between two runs of the same program due to TDMA is limited to $w-1$ cycles when one TDMA-arbitrated resource is used and $lcm(w_1, w_2, ..., w_k) - 1$ when $k > 1$ TDMA resources are used.

Hence, we could increase the execution time observations obtained at analysis time by $w-1$ cycles without breaking MBPTA compliance and reliably upper-bound the effect of TDMA alignment in the execution time. The process is as depicted in Figure 6. MBPTA [13], [31] performs several runs of the program under analysis on the target platform for a set of input vectors, labeled as iv_j in Figure 6. These runs are done under a setup in which the seeds for the hardware random generators, as well as other setup parameters, are properly initialized by the system software. As a result of this step, several execution time observations (et_i) are obtained. With the full-program padding approach, there is no need to control the *sad* for each run. Each of et_i is augmented, leading to a set of augmented execution time observations, as shown in Equation 4, where k is the number of TDMA-arbitrated resources.

$$aet_i = \begin{cases} et_i + w - 1 & \text{if } \; k = 1 \\ et_i + lcm(w_1, w_2, ..., w_k) - 1 & \text{if } k > 1 \end{cases} \quad (4)$$

The augmented observations, which deterministically upper-bound the maximum impact of TDMA *sad* alignment, are passed as input to MBPTA that obtains a pWCET estimate reliably upper-bounding the impact of TDMA *sad*. Note that augmenting all observations may be pessimistic since the actual *sad* experienced might not be the fastest one. However, as shown later in our evaluation, such pessimism is irrelevant in practice.

V. Impact of Timing Anomalies

As explained in Section III-A, our analysis focuses on processor designs free of timing anomalies. Although the definitions of timing anomalies are abundant [35], [52], [45], [54], here we stick to the following (informal) definition: a timing anomaly occurs when, by advancing the time when a instruction request is served, the execution time of the program increases. In other words, a local speedup turns out to produce a global slowdown.

Architectures free of timing anomalies have been referred to as *fully timing compositional architectures* according to the classification in [54]. This is the assumption for the architecture considered so far. Instead, if timing anomalies can occur but they cannot trigger domino effects, then the architecture is classified as a *compositional architecture with constant-bounded effects* according to [54]. In those cases, one can account for the impact of timing anomalies by counting how many times they can occur and multiplying that number by the maximum impact of a timing anomaly in execution time. Finally, *non-compositional architectures* in [54] are those that may experience timing anomalies and domino effects. Authors in [54] detail some means to make those architectures behave as *compositional architectures with constant-bounded effects*.

How to account for the impact of timing anomalies is beyond the scope of this paper. However, in this section we analyze how the use of TDMA arbitration may influence the occurrence of timing anomalies. In particular, we use an example of a timing anomaly and illustrate how the use of TDMA cannot create new types of anomalies, but create or eliminate some

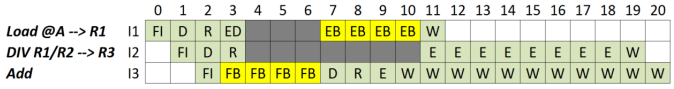

Fig. 7. Regular execution. (FI stands for fetch from IL1, FB for bus access during fetch, D for decode, R for register read, E for execute, ED for DL1 access during execute, EB for bus access during execute and W for write-back). Grey boxes correspond to bus stalls and yellow boxes to bus accesses. In this case i_3 occupies the bus delaying both i_1 and i_2.

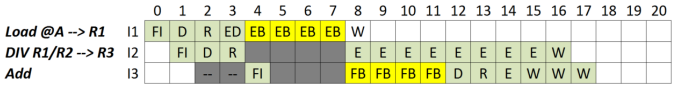

Fig. 8. Timing anomaly that occurs when delaying i_3 by 2 cycles, which results in a decrease of the overall execution time. (FI stands for fetch from IL1, FB for bus access during fetch, D for decode, R for register read, E for execute, ED for DL1 access during execute, EB for bus access during execute and W for write-back). Grey boxes correspond to bus stalls and yellow boxes to bus accesses.

that could already occur without TDMA. Finally, we discuss how these anomalies are avoided in our processor design.

A. An Example of Timing Anomaly

TDMA, as any other arbitration policy for shared resources, influences the arrival time of requests to the target device. For instance, TDMA implemented in a bus has an impact on the arrival time of the requests of the cores to the second level cache (L2) that sits in the other side of the bus. For instance, TDMA may, indirectly, reorder L2 or memory requests issued by a particular core. Let us assume that our processor issues L2 instruction accesses and L2 data accesses in order locally. That is, all instruction accesses occur in order (and so it is the case for data accesses), and the core always prioritizes the request from the oldest instruction when choosing among instruction and data requests. This leads to prioritizing globally the ready request from the oldest instruction. However, requests can be issued out-of-order if a request from a given instruction becomes ready after a request from a younger instruction has already been granted access to the bus.

Let us assume a program where 3 instructions are executed as shown in Figure 7. i_1 is a load to address $@A$, which is not present in first level data cache (DL1). Thus, i_1 fetches the instruction from the first level instruction cache (IL1) in cycle 0 (FI), decodes it in cycle 1 (D), reads its operands in cycle 2 (R), accesses DL1 in cycle 3 (ED, execute-DL1) and misses, waits for the bus to be available during 3 cycles, and then it spends 4 cycles to access L2 and retrieve the data (EB, execute-bus), which is placed in register $R1$. i_2 is a division that reads $R1$, so until the data is not read from L2 (cycle 11), it cannot start its execution. The division takes 8 cycles to execute (E) and the result is written back to the register file (W) in cycle 19. Finally, i_3 is an addition that misses in IL1. Thus, it accesses IL1 in cycle 2 and misses, and sends a request to the bus in cycle 3 (FB, fetch-bus). Since no further request is ready in cycle 3, it is granted access and gets the instruction from L2, thus keeping the bus busy until cycle 6. Then, it continues its execution and writes back the result in cycle 20

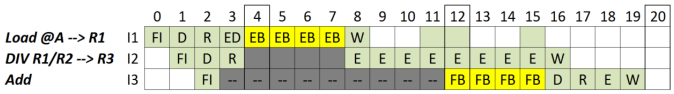

Fig. 9. Example in which a TDMA window of size $w = 8$ cycles (two slots of 4 cycles each) prevents the timing anomaly presented in Figure 8 from occurring. (FI stands for fetch from IL1, FB for bus access during fetch, D for decode, R for register read, E for execute, ED for DL1 access during execute, EB for bus access during execute and W for write-back). Grey boxes correspond to bus stalls and yellow boxes to bus accesses.

to preserve in-order finalization of instructions. Overall, the program takes 21 cycles to execute. If we artificially enforce i_3 to start 2 cycles later (in cycle 4), the bus access from i_1 is ready before the one from i_3, so i_1 is not delayed, as illustrated in Figure 8. Hence, i_2 completes its execution earlier, and the program finishes in cycle 17. Thus, by delaying i_3 the program executes 3 cycles faster, so revealing a timing anomaly.

B. Example in the Context of TDMA

If we assume a TDMA bus shared across 2 cores with a window of 8 cycles (4-cycle slots per core), this timing anomaly may not occur. This is illustrated in Figure 9, where requests are allowed to start in cycles 4, 12, 20, etc. As shown, in cycle 4 requests from both i_1 and i_3 are ready, but i_1 is granted access since it is the oldest instruction. Instead, i_3 needs to wait until cycle 12 to access the bus. This makes the program finish in cycle 19. In this example, by delaying i_3 start time we could never execute the program faster, so the timing anomaly cannot occur.

Note that other programs and/or time alignments of the TDMA window could still allow requests to be reordered, which could trigger some timing anomalies. However, since TDMA arbitration can only introduce delays, it may only avoid or create some reordering scenarios, thus creating or eliminating some timing anomalies of a type already existing without TDMA (e.g. due to request reordering).

If timing anomalies can occur due to, for instance, contention in the use of some other resources (e.g. an arithmetic unit), TDMA could also remove or trigger some of them due to the potential delay TDMA could introduce in the execution of some instructions, which could arrive at the conflictive resource in different order. However, TDMA delays do not reorder events that could not occur in arbitrary order. For instance, fetch and write-back occur always in-order, and therefore, TDMA cannot alter such behavior that is already enforced by hardware means.

C. Avoiding Timing Anomalies

In our particular design, this timing anomaly is avoided by not allowing instruction misses access the bus until all older instructions with data accesses (loads and stores) have reached their execution stage, so that any request reaching the L2 cache does it always in the program order. This is basically the same solution illustrated in Figure 8, where i_3 execution is delayed until all older requests (the one by i_1) are issued to the bus and hence, access order to shared resources is preserved. If other timing anomalies can occur due to, for instance, contention

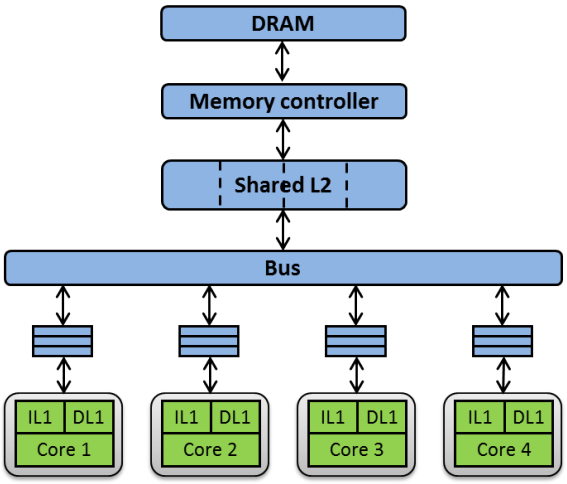

Fig. 10. Schematic of the multicore processor considered.

in some other resources (e.g. an arithmetic unit), there are two ways to also avoid them by construction: replicating those resources so that no contention can occur or making them serve a request only when no request from older instructions may arrive to the resource.

VI. Results

In this section we first introduce the evaluation framework, then we examine how TDMA *sad* impacts execution time and finally we compare 3 arbitration policies: TDMA, IARA (interference-aware resource arbiter) based on round-robin [42] and a MBPTA-specific randomized arbitration policy called random permutations [20].

A. Evaluation Framework

Processor setup. We use a cycle-accurate modified version of the SoCLib [34] framework modeling a multicore processor as the one shown in Figure 10. We use 3-stage in-order execution cores. Caches implement random placement and random replacement[6]. DL1 and IL1 caches are 8KB, 4-way with 32-byte lines. DL1 is write-through. The L2 is 128KB, 8-way with 32-byte lines. The L2 deploys cache partitioning, in particular way-partitioning as implemented in real processors like ARM A9 or Cobham NGMP [11], so that each core has exclusive access to 2 ways. This prevents contention in the cache as it is hard to account for in WCET estimates. These cache designs have been proven MBPTA compliant [28], [29], [51]. Cache access latencies are 1 cycle for DL1/IL1 and 2 cycles for L2. Note that L2 turnaround time can be typically around 10 cycles due to 2 bus traversals to send the request and receive its corresponding answer. There are two independent buses to send requests from cores to L2 and to send answers from L2 back to the cores. Both buses have a 2-cycle latency once access is granted.

[6]Time-randomized caches have been shown to provide the randomization properties required by MBPTA [28], [29]. The potential implications of random placement on MBPTA have been discussed in [44], [3], [37].

TABLE II
Workloads for average performance evaluation. *C0* corresponds to the benchmark under consideration. *C1*, *C2* and *C3* correspond to the benchmarks running in the other cores.

C0	C1	C2	C3
a2time	aiifft	iirflt	tblook
aifftr	idctrn	pntrch	idctrn
aifirf	pntrch	canrdr	a2time
aiifft	cacheb	cacheb	aifirf
basefp	cacheb	basefp	aifftr
bitmnp	rspeed	bitmnp	idctrn
cacheb	rspeed	canrdr	bitmnp
canrdr	iirflt	puwmod	ttsprk
idctrn	aiifft	basefp	iirflt
iirflt	a2time	idctrn	cacheb
matrix	matrix	basefp	a2time
pntrch	aiifft	idctrn	canrdr
puwmod	idctrn	basefp	matrix
rspeed	bitmnp	matrix	aifirf
tblook	ttsprk	matrix	matrix
ttsprk	ttsprk	a2time	aiifft

We use a time-analyzable memory controller [41] with per-request queues. We assume a CPU frequency of 800MHz and DDR2-800E SDRAM with the memory controller implementing close-page and interleaved-bank policies, which delivers 16-cycles access latency and 27-cycles inter-access latency [22]. Thus, an access completes in 16 cycles once it is granted access to memory, but the next access has to wait 11 extra cycles to start to allow the page accessed to be closed. This typically leads to memory latencies around 100 cycles due to contention and access delay.

In our experiments, to control the access to both the bus and memory controller, we deploy three different arbitration policies: random permutations [20], IARA based on round-robin [42], [19] and TDMA. The particular policy used in each experiment is indicated conveniently.

Although the particular setup used in this work has not been explicitly validated against real hardware, a non-MBPTA-compliant configuration with modulo placement and LRU replacement in all caches, and FIFO bus and memory controller arbitration has been assessed against the Cobham NGMP processor for the Space Domain [11] showing performance differences between 1% and 5% for different microbenchmarks and relevant applications used by the European Space Agency [21]. Therefore, we are confident on the conclusions reached based on the results obtained with this simulator.

Benchmarks. We consider the EEMBC Autobench benchmarks [43], a well-known suite reflecting the current real-world demand of some automotive embedded systems.

In order to derive pWCET estimates, we collected 1,000 execution times for each benchmark, which have been shown enough for MBPTA [13] application. The observations collected in all the experiments passed the independence and identical distribution tests as required by MBPTA [13].

For these experiments focused on deriving reliable pWCET bounds, it is assumed that each request of the task under

TABLE III
MAXIMUM EXEC. TIME VARIATIONS DUE TO TDMA *sad*.

Bench.	TDMA bus only	TDMA bus and mem.ctrl.
a2time	7	215
aifftr	7	215
aifirf	7	111
aiifft	7	215
basefp	7	215
bitmnp	7	215
cacheb	7	111
canrdr	7	111
idctrn	7	215
iirflt	7	111
matrix	7	215
pntrch	7	111
puwmod	7	111
rspeed	7	111
tblook	7	111
ttsprk	7	111

analysis is arbitrated against requests from the other cores. While in reality this is not always the case, since the contention on the average case is lower, this ensures that derived estimates are actually an upper-bound to the execution time of the task when it runs during operation regardless of the load contender tasks put on the bus and memory.

In order to study average performance, for each EEMBC Autobench benchmark we created one workload with 3 other randomly selected benchmarks. The benchmark in the first core is the one under analysis while the other three in cores 1, 2 and 3 are contending benchmarks. We perform 100 runs collecting execution times of the benchmark in core 0. In the experiments, benchmarks in cores 1, 2 and 3 are restarted if they finish before the program under analysis (core 0). The particular workloads evaluated are shown in Table II, where column *C0* corresponds to the benchmark being analyzed.

B. Impact of TDMA sad on Execution Time

In this section we empirically confirm that the impact of TDMA resources is at most w cycles when a single TDMA resource is used and $lcm(w_1, w_2, ..., w_k)$ cycles for k TDMA resources.

Single TDMA resource. For this experiment we use a TDMA-arbitrated bus to access L2. Bus latency is 2 cycles and $w_{bus} = 8$ (4 slots for the 4 cores, each slot of $s = 2$ cycles). The responses from the L2, which is assumed perfect (i.e. all accesses hit) arrive in a fixed latency of 2 cycles. DL1/IL1 cache memories are always initialized with the same seeds so that the random events produced are exactly the same across all experiments. In this way, the only *setv* is the *sad* for the bus. We run 8 experiments with the 8 different *sad* for each benchmark. The *"TDMA bus only"* column in Table III shows the maximum execution time variation observed for each benchmark. As shown, *all* benchmarks observe exactly a maximum difference of $w_{bus} - 1 = 7$ cycles. In fact, we have corroborated that execution times for the 7 slowest *sad* of each benchmark are exactly 1, 2, 3, 4, 5, 6 and 7 cycles higher than that of the fastest *sad*. This means that, in all runs, at some point requests get delayed until they align (synchronize) with TDMA as in the fastest case, and then execution continues identically.

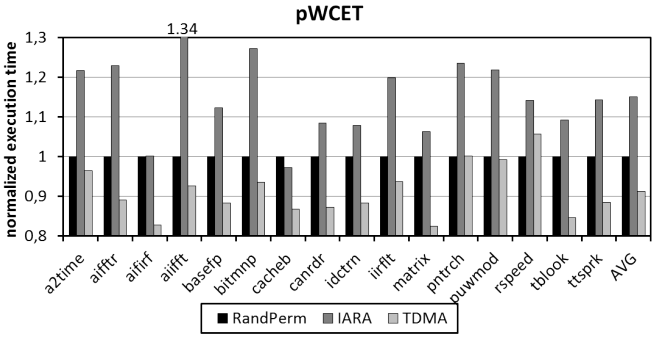

Fig. 11. pWCET estimates for a cutoff probability of 10^{-15} normalized w.r.t. time-randomized arbitration.

Multiple TDMA resources. For this experiment we use the original processor setup. We have 3 TDMA resources: the buses to reach L2 and get answers from it, and the memory controller. Both buses have $w_{bus} = 8$, 2-cycle slots. The memory controller has 27-cycle slots, so $w_{memctrl} = 108$ cycles due to the 4 contender cores. Thus, $lcm(8, 8, 108) = 216$. Experiments are run as before, fixing seeds for caches so that execution time variations are produced only due to the alignment with TDMA resources. We have run 216 experiments for each benchmark with the 216 different *sad*. The *"TDMA bus and mem. ctrl."* column in Table III shows the maximum execution time variation observed for each benchmark. As shown, such difference is at most $lcm(8, 8, 108) - 1 = 215$ cycles, thus further corroborating our hypothesis. In fact, in 7 out of the 16 benchmarks such difference is exactly 215 cycles. In the other 9 cases it is 111 cycles. Those 111 cycles come from the fact that the memory controller window is much larger than the bus one, and in some cases it is enough to align with such window to get identical or near-identical timing behavior as in the fastest case. This explains $w_{memctrl} - 1 = 107$ cycles. The other 4 cycles correspond exactly to the misalignment of the TDMA bus windows after $w_{memctrl} = 108$ cycles.

C. Worst-Case Performance (pWCET) Comparison

We evaluate arbitration policies in terms of worst-case performance, which is measured with the probabilistic WCET estimates provided by MBPTA. In all experiments, we use the same arbitration policy in the buses and in the memory controller. Seeds for the caches are initialized randomly on each run. We use the following setup for each policy:

- *Time-randomized*. We use random permutations arbitration, with which on every arbitration window a random permutation of the slots is created so that in every window the contenders access the bus in a random fashion [20].
- *IARA* uses as baseline arbitration round-robin. In the worst case, the bus latency of an access is: 3 x 2 cycles of delay that the request may suffer waiting for the requests of the other cores, plus its own 2 cycles to access the bus.

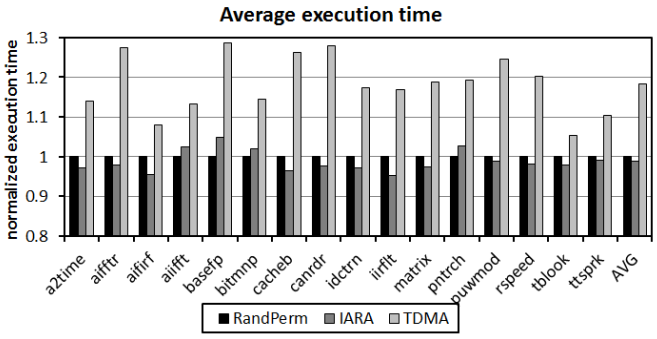

Fig. 12. Average execution time normalized w.r.t. time-randomized arbitration.

Likewise, the longest memory latency is 97 cycles due to the 3 slots for the other cores (3 x 27) and the 16-cycle access of the current request.

- *TDMA*. With TDMA experiments are run assuming always an arbitrary *sad*. We use full-program padding increasing the observations passed to MBPTA by 215 cycles.

In order to ensure that pWCET estimates are time-composable (i.e. do not depend on the co-runners as needed to isolate across different criticalities), for these experiments we use non-work-conserving versions of all arbitration policies. Hence, the task under analysis can only access the resources in its slots (time-randomized and TDMA). In those slots in which it is not granted access, the task cannot access the shared resource even if it is idle.

Figure 11 shows the pWCET estimates for each benchmark. We use a cutoff probability of 10^{-15} per activation as it has been shown appropriate in some industrial case studies [51]. Results have been normalized with respect to the time-randomized bus.

We observe that IARA is 15% worse than random permutations on average. IARA is, in fact, the worst policy since it assumes each request to experience its worst-case latency. TDMA is 9% better on average than random permutations because TDMA slots for a given core are homogeneously distributed in time, thus leaving some time between consecutive slots. Conversely, random permutations may lead, with relatively high probability, to consecutive slots assigned for a given core in the memory controller because it is granted access last in one permutation and first in the next one. However, some cycles elapse since the data reaches the core for a load request until the next request (either a load or a store) from this core reaches the memory controller. This is enough to miss its opportunity and the request has to wait for a later slot that will not arise until the next permutation. Overall, although the average time between slots for random permutations and TDMA is the same, under random permutations some slots cannot be used and so pWCET estimates are affected.

Differences for individual benchmarks w.r.t. the average case occur due to the random variations that affect measurements, which may lead to higher or lower tightness in some cases [49]. Still, results are quite consistent across benchmarks.

D. Average performance

Figure 12 shows the average execution time for the 100 runs of each configuration for each benchmark in its respective workload assuming operation conditions. Results have been normalized with respect to the time-randomized bus for the sake of readability. As shown, TDMA performs clearly worse than random permutations (*RandPerm* in the plot) and IARA. In particular, TDMA performs 18% worse than *RandPerm* and 19% worse than IARA on average. This occurs because *RandPerm* and IARA assume worst contention at analysis, but actual contention during operation, and actual contention is typically low. Thus, pWCET estimates hold, but tasks are not isolated, thus contending in the use of shared resources. Potential contention is upper-bounded at analysis time, but tasks are allowed to contend freely during operation (work-conserving arbitration). Conversely, TDMA enforces isolated time slots for each task, in such a way that the timing behavior observed at analysis *matches* the one observed during operation. Thus, the requests of any task can be stalled since they have to wait for the corresponding slot to access the shared resource, even if the resource is idle (non-work-conserving arbitration).

In summary, TDMA is the best choice in terms of pWCET estimates, whereas *RandPerm* is the second best choice. In terms of average performance *RandPerm* and IARA are the best choices by far.

In the light of this results, if the most critical resource is the time budget available for hard real-time tasks, then TDMA is the most convenient arbitration policy despite its relatively low average performance. Instead, if the time budget for hard real-time tasks is large (e.g. because few tasks have hard real-time constraints), then *RandPerm* is the most convenient solution. Finally, IARA is only the most convenient policy if virtually all tasks have soft or non real-time constraints since its average performance is marginally better than that of *RandPerm* while offering significantly worse pWCET estimates.

VII. Related work

Several works analyze, from a SDTA point of view, the impact of on-chip bus arbitration policies, especially TDMA [46], [25] and round-robin [42], [18], on WCET. In [25] an analysis and evaluation of a TDMA arbitrated bus under the context of SDTA considering both, architectures with and without timing anomalies, is performed. In [42] an analysis of the delay that every request can suffer when accessing a round-robin arbitrated resource is carried out. Jacobs et al. [18] focus on extending SDTA to consider multicore contention for event-driven arbitration policies such as round-robin by accounting for it in the abstract states kept during the analysis. This allows deriving WCET bounds for in-order and out-of-order processors, valid for any contender or suited for specific contention scenarios.

More complex inter-connection architectures such as meshes [48] or rings [39] based on the use of TDMA and round-robin have also been shown to be analyzable with SDTA techniques. For the TDMA case, the Time-Triggered Architecture [26] (TTA) implements time-predictable communication by

means of customized TDMA schedules. Other approaches like T-CREST [48] deliver low complexity TDMA-based NoCs with global schedule that enable straightforward WCET analysis. For round-robin, several studies [7], [8] propose offering several levels of round-robin arbitration for asymmetric upper-bound delay (*ubd*) so that high priority tasks may enjoy lower *ubd*. In [24], [19] authors present a comparison of TDMA and round-robin for SDTA and MBDTA considering different metrics.

Probabilistic timing analysis on non-MBPTA-compliant architectures has been considered in several works [6], [15], [16], [14], [38] including WCET estimation and scheduling techniques. Those solutions cannot be applied in the context of MBPTA since they do not provide means to relate the conditions experienced during timing analysis with those during operation (e.g. memory placement, shared resource contention, etc). Among those works, we notice that Palopoli et al. [38] explore solutions related to ours, although at the task scheduling level. Authors reason about the timing behavior of tasks building on the temporal isolation provided by resource reservation in a similar way to how we reason about temporal isolation provided by TDMA at shared resource request level.

For MBPTA, several specific arbitration policies have been proposed, which include random lottery access [33] and random permutations [20], both based on the idea of introducing some type of randomization when granting access to the different contenders. With the lottery bus, on every (slot) round the grant is given randomly to one of the resource contenders. With random permutations, on every window a random permutation of the slots is assigned so that in every window the contenders access the bus in a random fashion. To the best of our knowledge, this is the first attempt to analyze the benefits of TDMA, a DTA-amenable arbitration policy, in the context of MBPTA.

More complex inter-connection architectures have been considered in the context of MBPTA such as trees [50], which provide advantages over buses for large core counts at the expense of some additional hardware complexity. Those designs also consider variations of the arbitration policies to meet mixed-criticality constraints. While not specifically devised for MBPTA-compliant architectures, novel arbitration policies [17] have been devised for mixed-criticality environments with multi-level arbiters, to some extent similar to those of the MBPTA-compliant tree in [50], and able to adapt dynamically to different scenarios.

VIII. Conclusions

Different types of timing analyses impose heterogeneous constraints on hardware designs, so chip vendors have to face the challenge of deciding which timing analysis to support (if any). Hence, proving that the same hardware design can be used to obtain reliable and tight WCET estimates with different families of timing analyses is of prominent importance to increase the chance of those hardware designs being realized.

In this paper we prove that shared resources implementing TDMA arbitration, which meet mixed-criticality systems requirements, can be analyzed in the context of MBPTA. We introduce small changes to the application of MBPTA with which WCET estimates obtained are 9% lower on average than those obtained with MBPTA-compliant designs.

Acknowledgments

The research leading to these results has been funded by the EU FP7 under grant agreement no. 611085 (PROXIMA) and 287519 (parMERASA). This work has also been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P and the HiPEAC Network of Excellence. Miloš Panić is funded by the Spanish Ministry of Education under the FPU grant FPU12/05966. Jaume Abella has been partially supported by the MINECO under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717.

References

[1] *AMBA Bus Specification. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php*.
[2] Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. *ARP4761*, 2001.
[3] J. Abella et al. Heart of gold: Making the improbable happen to extend coverage in probabilistic timing analysis. In *ECRTS*, 2014.
[4] J. Abella et al. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *SIES*, 2015.
[5] S. Altmeyer and R. Davis. On the correctness, optimality and precision of static probabilistic timing analysis. In *DATE*, 2014.
[6] G. Bernat, A. Colin, and S. Petters. WCET analysis of probabilistic hard real-time systems. In *RTSS*, 2002.
[7] R. Bourgade et al. MBBA: a multi-bandwidth bus arbiter for hard real-time. In *EMC*, 2010.
[8] R. Bourgade et al. Predictable bus arbitration schemes for heterogeneous time-critical workloads running on multicore processors. In *ETFA*, 2011.
[9] F. Cazorla et al. Proartis: Probabilistically analysable real-time systems. *ACM TECS*, 2012.
[10] F. Cazorla et al. Upper-bounding program execution time with extreme value theory. In *WCET workshop*, 2013.
[11] Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor - LEON4-NGMP-DRAFT - Data Sheet and User's Manual*, 2011.
[12] S. Coles. *An Introduction to Statistical Modeling of Extreme Values*. Springer, 2001.
[13] L. Cucu-Grosjean et al. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS*, 2012.
[14] J. Díaz, D. Garcia, K. Kim, C. Lee, L. Bello, L. J.M., and O. Mirabella. Stochastic analysis of periodic real-time systems. In *the 23rd IEEE Real-Time Systems Symposium (RTSS02)*, pages 289–300, 2002.
[15] Edgar S and Burns A. Statistical analysis of WCET for scheduling. In *the 22nd IEEE Real-Time Systems Symposium (RTSS01)*, pages 215–225, 2001.
[16] J. Hansen, S. Hissam, and G. A. Moreno. Statistical-based wcet estimation and validation. In *the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.
[17] M. Hassan and H. Patel. Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, April 2016.
[18] M. Jacobs, S. Hahn, and S. Hack. WCET analysis for multi-core processors with shared buses and event-driven bus arbitration. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, pages 193–202, 2015.
[19] J. Jalle et al. Deconstructing bus access control policies for real-time multicores. In *SIES*, 2013.
[20] J. Jalle et al. Bus designs for time-probabilistic multicore processors. In *DATE*, 2014.
[21] J. Jalle et al. Validating a timing simulator for the NGMP multicore processor. In *DASIA*, 2016.
[22] JEDEC. *DDR2 SDRAM Specification JEDEC Standard No. JESD79-2E*, April 2008.

[23] A. Kadlec et al. Avoiding timing anomalies using code transformations. In *ISORC*, 2010.

[24] T. Kelter et al. Bus-aware multicore WCET analysis through TDMA offset bounds. In *ECRTS*, 2011.

[25] T. Kelter et al. Static analysis of multi-core TDMA resource arbitration delays. *Real-Time Systems*, 50(2):185–229, 2014.

[26] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1), 2003.

[27] L. Kosmidis et al. Applying measurement-based probabilistic timing analysis to buffer resources. In *WCET workshop*, 2013.

[28] L. Kosmidis et al. A cache design for probabilistically analysable real-time systems. In *DATE*, 2013.

[29] L. Kosmidis et al. Multi-level unified caches for probabilistically time analysable real-time systems. In *RTSS*, 2013.

[30] L. Kosmidis et al. Probabilistic timing analysis and its impact on processor architecture. In *Euromicro DSD*, 2014.

[31] L. Kosmidis et al. PUB: Path upper-bounding for measurement-based probabilistic timing analysis. In *ECRTS*, 2014.

[32] S. Kotz and S. Nadarajah. *Extreme value distributions: theory and applications*. World Scientific, 2000.

[33] K. Lahiri et al. LOTTERYBUS: a new high-performance communication architecture for system-on-chip designs. In *DAC*, 2001.

[34] LiP6. SoCLib. www.soclib.fr/trac/dev.

[35] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *RTSS*, 1999.

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

[37] E. Mezzetti et al. Randomized caches can be pretty useful to hard real-time systems. *LITES*, 2(1), 2015.

[38] L. Palopoli, D. Fontanelli, L. Abeni, and B. V. Fras. An analytical solution for probabilistic guarantees of reservation based soft real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):640–653, March 2016.

[39] M. Panic et al. On-chip ring network designs for hard-real time systems. In *RTNS*, 2013.

[40] M. Panic et al. Enabling TDMA arbitration in the context of MBPTA. In *DSD*, 2015.

[41] M. Paolieri et al. *An Analyzable Memory Controller for Hard Real-Time CMPs* . Embedded System Letters (ESL), 2009.

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

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

[44] J. Reineke. Randomized caches considered harmful in hard real-time systems. *LITES*, 1(1):03:1–03:13, 2014.

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

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

[47] L. Santinelli, J. Morio, G. Dufour, and D. Jacquemart. On the sustainability of the extreme value theory for WCET estimation. In *14th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2014.

[48] M. Schoeberl et al. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *NOCS*, 2012.

[49] M. Slijepcevic et al. DTM: Degraded test mode for fault-aware probabilistic timing analysis. In *ECRTS*, 2013.

[50] M. Slijepcevic et al. pTNoC: Probabilistically time-analyzable tree-based NoC for mixed-criticality systems. In *DSD*, 2016.

[51] F. Wartel et al. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *SIES*, 2013.

[52] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *ICQS*, 2005.

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

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