Enabling TDMA Arbitration in the Context of MBPTA

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

***Abstract*—Current timing analysis techniques can be broadly classified into two families: deterministic timing analysis (DTA) and probabilistic timing analysis (PTA). Each family defines a set of properties to be provided (enforced) by the hardware and software platform so that valid Worst-Case Execution Time (WCET) estimates can be derived for programs running on that platform. However, the fact that each family relies on each own set of hardware designs limits their applicability and reduces the chances of those designs being adopted by hardware vendors.**

In this paper we show that Time Division Multiple Access (TDMA), one of the main DTA-compliant arbitration policies, can be made PTA-compliant. To that end, we analyze TDMA in the context of measurement-based PTA (MBPTA) and show that padding execution time observations conveniently leads to trustworthy and tight WCET estimates with MBPTA without introducing any hardware change. In fact, TDMA outperforms round-robin and time-randomized policies in terms of WCET in the context of MBPTA.

I. Introduction

Developing a real-time system requires validating its timing behavior. This can be done by deriving WCET estimates to the execution time of each task, which are passed as input to the scheduler that combines them with other task information such as deadline, period and priority to validate that the budgets provided to each task are sufficient to satisfy the tasks' execution time needs. Deterministic-Timing Analysis techniques [39], both static and measurement-based (SDTA and MBDTA), advocate for time-deterministic architectures. The goal is that the access time to each resource can be upper-bounded so that (1) with SDTA, bounds can be *incorporated* in the analysis and (2) with MBDTA, bounds can be *enforced* in the measurements taken during the analysis phase. Probabilistic Timing Analysis (PTA) [4], [7], [9], [21], [3] supports architectures in which some resources are time-deterministic whereas others are time-randomized [20]. The goal is that resources' impact on execution time can be bounded either with a fixed value (deterministic upper-bounding) or a distribution function (probabilistic upper-bounding) [20].

Mixed-criticality applications running in multicores challenge both timing analysis families, DTA and PTA, because 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 [15], 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 [20], the impact of contention that can occur during the deployment of the system. While round-robin arbitrated shared resources used in the context of DTA have also been proven analyzable with MBPTA [11], this is not the case for TDMA arbitrated shared resources.

Contribution. In this paper we analyze in detail TDMA in the context of MBPTA and provide means to allow TDMA resources to be used together with MBPTA, which is of high relevance since TDMA arbitration fulfils isolation requirements coming from mixed-criticality applications. Furthermore, we show that TDMA allows obtaining tighter WCET estimates than round-robin by padding execution time once instead of padding the latency of each request. 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 PTA [20]. 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 [15]. Also, we show that the effect of TDMA for several chained arbitrations is limited to the least-common-multiple of the TDMA windows. ③ 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.

Our analysis not only advances the limits on the arbitration policies that can be analyzed with MBPTA without requiring MBPTA-customized designs [11], 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 on average than round-robin and MBPTA-specific arbitration policies.

II. Contention analysis for DTA and MBPTA

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 contention, specially the arbitration delay. Several proposals have shown how to handle contention in the access to hardware shared

[1] In this paper we focus only on MBPTA because it has been shown to be close to industrial practice for timing analysis and it has been already evaluated with avionics case studies [37].

resources so that trustworthy WCET estimates can be provided. For on-chip resources, which is the focus of this paper, the goal is providing time composability in the access latency for WCET estimation. This means that access latency can be upper-bounded such that the load that other tasks put on that resource does not exceed the access latency used for WCET estimation purposes for the task under analysis, thus avoiding interferences across tasks with mixed criticalities.

A. SDTA and MBDTA

SDTA [39], [25] 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 makes extensive testing on the target system with stressful, high-coverage input data. From all the tests it is recorded the longest observed execution time 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 trustworthy 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 [10]. 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 deployment ones, the execution time derived at analysis time for the program upper-bounds the impact of the shared resource. 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 [29] 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. And (3) with SDTA, it is possible to carry out a combined timing analysis of all the tasks simultaneously running in the multicore [14]. While this may reduce the impact of contention on WCET estimates, since only the actual contention generated by the co-running tasks is considered, 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

PTA 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. For instance DO-178B/C [34] for avionics sets the maximum allowed failure rate of a system component to 10^{-9} per hour of operation for its highest integrity level. This translates into 10^{-15} exceedance probability for tasks triggered every 10ms [37].

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

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 X

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

nature of the different *contributors* to the execution time of a program [8]. 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., 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, i.e. execution time variability, of all *setv* captured in the end-to-end execution times collected at analysis time upper-bound the jitter of each *setv* when the system is deployed (*deployment phase*). In [20] 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 [9].

Jitter can be upper-bounded *deterministically* [20] by forcing *setv* to experience a single latency at analysis time lat_{det}^{an} that upper-bounds any latency that the *setv* may take at deployment, $lat_{det}^{dep,i}$. That is, $\forall i : lat_{det}^{an} \geq lat_{det}^{dep,i}$. 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. At deployment, real latencies will be equal or lower than those at analysis time.

Jitter can also be upper-bounded *probabilistically* [20] by forcing 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 at deployment. For instance, let us assume random-permutations arbitrated bus [11] shared by N_c cores. Further assume that at deployment 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 at deployment 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 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 at deployment time the number of active cores is at most 4, the analysis time contention distribution upper-bounds that obtained at deployment time rendering this arbitration policy as MBPTA analyzable.

[2]If no hardware support is in place measurements need to capture high contention scenarios, but trustworthiness of the WCET estimates is hard to be proven.

[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.

III. TDMA IMPACT ON EXECUTION TIME

TDMA ensures that the load a task puts on a shared resources does not affect the WCET of its co-runners [10], 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. 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}) to 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 [24], [38], [32], [13]. A number of definitions have been devised for timing anomalies. In our case, a processor architecture free of timing anomalies refers to an architecture where an increase in the access latency of a request to any resource (e.g., due to contention) can only lead to an equal or higher 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 requester 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 ready one cycle after its slot expires making it wait sad_{tdma} cycles that is defined in Equation 2.

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

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.

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

As shown in Equation 2, the particular *sad* of a request may make it be served right away (so 0 delay) or delayed by up to $(Nc - 1) \times s$ cycles, or in other words, $w - 1$ cycles. 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 [15]. 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 [15]. 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. [15] for a formal proof.

C. 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 1 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).

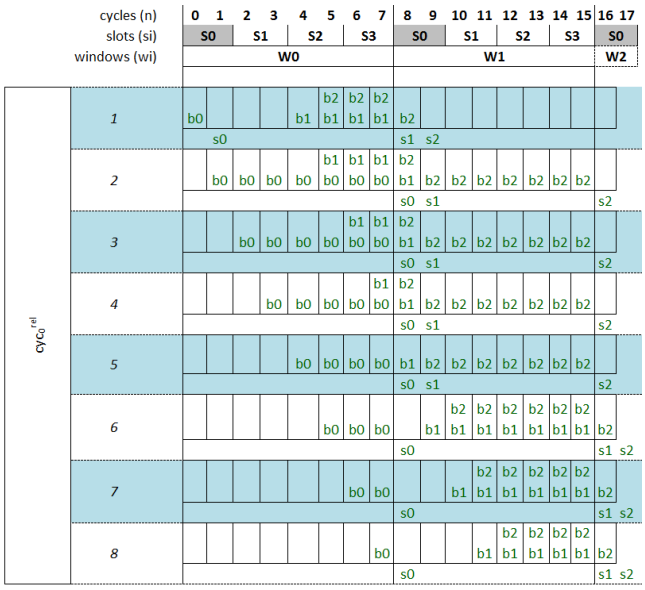

Fig. 1. Example of 3 requests with $\Delta^{inj} = \{-, 4, 1\}$ and their *sad*. b_i are the cycles in which the request is ready but waiting in the buffer due to *sad*; s_i represents cycle in which the request gets access to the bus. Finally blanks represent the cycles with no requests on the bus.

Overall, each different *sad* takes 10, 16, 15, 14, 13, 13, 12 and 11 cycles respectively.

In Figure 1 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 1: *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 1) and any other arbitrary *sad scenario*. Let $shift^{sad}$ be the cycle count difference 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 2: *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.*

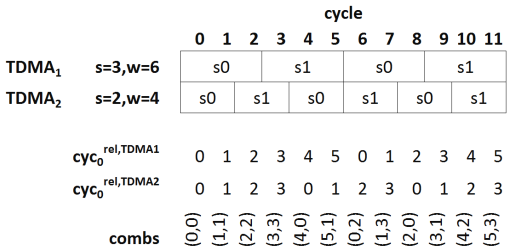

Fig. 2. Different combinations – in a two TDMA-window case – for $cyc_0^{rel,TDMA1}$ and $cyc_0^{rel,TDMA2}$.

D. Multiple TDMA resources

When several TDMA-arbitrated resources are used in the system (e.g., k TDMA resources), at most $lcm(w_1, w_2, ..., w_k)$ different *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 2 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 3: *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.

E. 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 $I_{r_{i-1}}$, after the execution of $I_{r_{i-1}}$ block the pipeline 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 [17]. 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 [18], which is the only source of execution time variability (the instruction cache is assumed perfect)[4]. 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 3. 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 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 can only lead to one 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, $\Delta^{inj} = \{-, 3, 8\}$

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

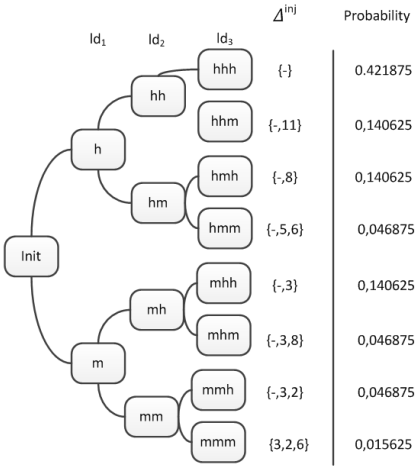

Fig. 3. Different probabilistic states in which the processor may be after the execution of each of the 3 loads in the example.

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 7 1-cycle instructions before ld_3 accesses cache.

In a second experiment, instead of assuming a no-contention bus, we use 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 4: *In the absence of MBPTA-specific support, TDMA is not by default 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 with the TDMA slots upper-bounds the impact of TDMA at deployment.*

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 [17], [3]. A different approach is needed to account MBPTA impact on execution time and pWCET estimates.

C. 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

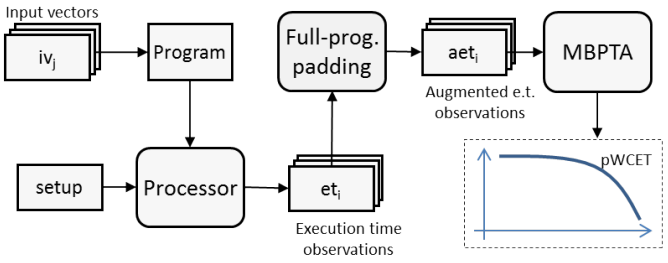

Fig. 4. Full-program padding in the context of MBPTA.

impact that the alignment with respect to 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 trustworthily upper-bound the effect of TDMA alignment in the execution time. The process is as depicted in Figure 4. MBPTA [9], [21] performs several runs of the program under analysis on the target platform for a set of input vectors, labeled as iv_j in Figure 4. 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 trustworthily 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. Results

In this section we first introduce the evaluation framework. Next, we examine how TDMA *sad* impacts execution time. Finally, we compare 3 arbitration policies: TDMA, IARA (interference-aware resource arbiter) based on round-robin [29] and a MBPTA-specific randomized arbitration policy called random permutations [11].

A. Evaluation Framework

Processor setup. We use a cycle-accurate modified version of the SoCLib [23] framework modeling a multicore processor as the one shown in Figure 5. We use 3-stage in-order execution cores. Caches implement random placement and random replacement[5]. First level data (DL1) and instruction

[5]Time-randomized caches have been shown effective in conjunction with MBPTA [18], [19]. Although they are not the object of this paper, it is worth mentioning that their use in the context of MBPTA has been regarded as risky in [31]. However, authors in [2], [26] provide detailed arguments about those concerns and why time-randomized caches can be used safely.

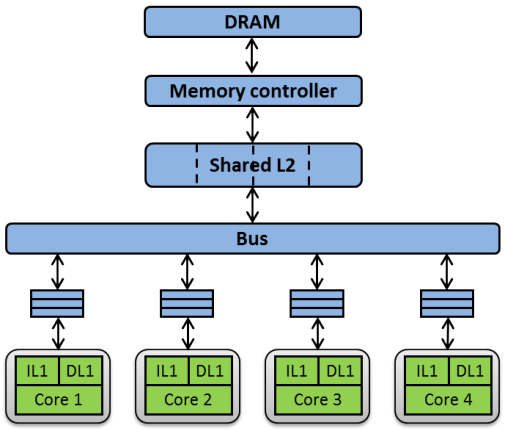

Fig. 5. Schematic of the multicore processor considered.

(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 processor like ARM A9 or Aeroflex NGMP, so that each core has exclusive access to 2 ways. This prevents contention in the cache as it is hard to model. These cache designs have been shown to be MBPTA compliant [18], [19], [37]. Cache 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.

We use a time-analyzable memory controller [28] 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 [12]. 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 [11], IARA based on round-robin [29], [10] and TDMA. The particular policy used in each experiment is indicated conveniently.

Benchmarks. We consider the EEMBC Autobench benchmarks [30], which is a well-known suite reflecting the current real-world demand of some automotive embedded systems.

When computing pWCET estimates, we collected 1,000 execution times for each benchmark, which proved to be enough for MBPTA [9]. The observations collected in all the experiments passed the independence and identical distribution tests as required by MBPTA [9].

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

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

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

$w_{bus} = 8$ (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. 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"* columns in Table II show 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.

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."* columns in Table II show 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. Performance 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 [11].
- **IARA**. Bus latency is always 8 cycles (4 cores x 2-cycle latency). Memory latency is always 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.

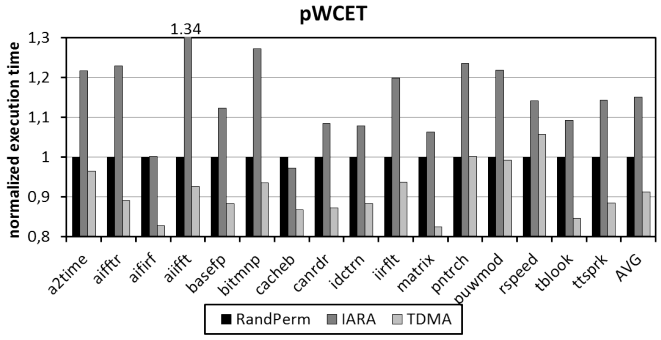

Fig. 6. pWCET estimates for a cutoff probability of 10^{-15} normalized w.r.t. time-randomized arbitration.

Since we use non-work-conserving versions of all arbitration policies, the task under analysis can only access the resources in its slot (time-randomized and TDMA). Those slots in which it is not granted access, the task cannot access the shared resource even if it is idle. This makes irrelevant what is run in the other cores or whether they are idle. Thus, pWCET estimates are time-composable (do not depend on the co-runners as needed to isolate across different criticalities), and results can be obtained keeping the other cores idle.

pWCET estimates. Figure 6 shows the pWCET estimates for each benchmark. We use a cutoff probability of 10^{-15} per activation as it has been shown appropriate to use in some domains as avionics [37]. Results have been normalized with respect to the time-randomized bus. IARA is 15% worse than random permutations on average. IARA is 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 last in one permutation and first in the next one. However, some cycles elapse since the data reach 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 wait for a later slot that will not show up 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 performance (and so WCET estimates) is 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 [36]. Still, results are quite consistent across benchmarks.

VI. RELATED WORK

Several works analyze, from a SDTA point of view, the impact of on-chip bus arbitration policies, specially TDMA [33], [15] and round-robin [29], on WCET. In [15] 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 [29] an analysis of the delay that every request can suffer when accessing a round-robin arbitrated resource is carried out.

More complex inter-connection architectures such as meshes [35] or rings [27] 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 [16] (TTA) implements timing-predictable communication by means of customized TDMA schedules. Other approaches like T-CREST [35] deliver low complexity TDMA-based NoCs with global schedule that enable straightforward WCET analysis. For round-robin several studies [5], [6] propose offering several levels of round-robin arbitration for asymmetric upper-bound delay (*ubd*) so that high priority tasks may enjoy lower *ubd*. In [14], [10] authors present a comparison of TDMA and round-robin for SDTA and MBDTA considering different metrics.

For MBPTA several specific arbitration policies have been proposed which includes random lottery access [22] and random permutations [11], 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.

VII. 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 trustworthy 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-friendly 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 TIN2012-34557 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] J. Abella et al. Heart of gold: Making the improbable happen to extend coverage in probabilistic timing analysis. In *ECRTS*, 2014.

[3] S. Altmeyer and R. Davis. On the correctness, optimality and precision of static probabilistic timing analysis. In *DATE*, 2014.

[4] G. Bernat, A. Colin, and S. Petters. WCET analysis of probabilistic hard real-time systems. In *RTSS*, 2002.

[5] R. Bourgade et al. MBBA: a multi-bandwidth bus arbiter for hard real-time. In *EMC*, 2010.

[6] R. Bourgade et al. Predictable bus arbitration schemes for heterogeneous time-critical workloads running on multicore processors. In *ETFA*, 2011.

[7] F. Cazorla et al. Proartis: Probabilistically analysable real-time systems. *ACM TECS*, 2012.

[8] F. Cazorla et al. Upper-bounding program execution time with extreme value theory. In *WCET workshop*, 2013.

[9] L. Cucu-Grosjean et al. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS*, 2012.

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

[11] J. Jalle et al. Bus designs for time-probabilistic multicore processors. In *DATE*, 2014.

[12] JEDEC. *DDR2 SDRAM Specification JEDEC Standard No. JESD79-2E*, April 2008.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[34] RTCA and EUROCAE. *DO-178B / ED-12B, Software Considerations in Airborne Systems and Equipment Certification*, 1992.

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

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

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

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

[39] 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.