Multi-Level Unified Caches for Probabilistically Time Analysable Real-Time Systems

Leonidas Kosmidis*,†, Jaume Abella†, Eduardo Quiñones†, Francisco J. Cazorla†,‡
* Universitat Politècnica de Catalunya † Barcelona Supercomputing Center
‡ Spanish National Research Council (IIIA-CSIC)

***Abstract*—Caches are key resources in high-end processor architectures to increase performance. In fact, most high-performance processors come equipped with a multi-level cache hierarchy. In terms of guaranteed performance, however, cache hierarchies severely challenge the computation of tight worst-case execution time (WCET) estimates. On the one hand, the analysis of the timing behaviour of a single level of cache is already challenging, particularly for data accesses. On the other hand, unifying data and instructions in each level, makes the problem of cache analysis significantly more complex requiring tracking simultaneously data and instruction accesses to cache.**

In this paper we prove that multi-level cache hierarchies can be used in the context of Probabilistic Timing Analysis and tight WCET estimates can be obtained. Our detailed analysis (1) covers unified data and instruction caches, (2) covers different cache-write policies (write-through and write back), write allocation policies (write-allocate and non-write-allocate) and several inclusion mechanisms (inclusive, non-inclusive and exclusive caches), and (3) scales to an arbitrary number of cache levels. Our results show that the probabilistic WCET (pWCET) estimates provided by our analysis technique effectively benefit from having multi-level caches. For a two-level cache configuration and for EEMBC benchmarks, pWCET reductions are 55% on average (and up to 90%) with respect to a processor with a single level of cache.

I. Introduction

Caches are undoubtedly one of the resources with the highest performance impact in a processor system. Most high performance processors come equipped with two of levels of cache, like the ORACLE UltraSPARC T2, or even three such as the IBM POWER7 or the Intel Core i7. This is also the case of some processors used in the real-time domain such as the ARM Cortex A9 and A15 [4], the Freescale P4080 [13] and the Aeroflex Gaisler NGMP [3].

Cache memories also impact noticeably worst-case execution time (WCET) and have been object of intense study during the last decades by the real-time community. This has motivated researchers to develop models that allow to derive the behaviour of the cache [22], [12], [23], [11], [20], [15] to determine whether cache accesses hit or miss. Those models have been particularly successful for the instruction cache, however, data cache remains a major challenge for static WCET analysis methods due to the difficulty of statically determining the address of each data memory access. The difficulties to determine the addresses of data accesses at run-time compel analysis techniques to make pessimistic assumptions, which in turn result in pessimistic WCET estimates. Multi-level caches aggravate these problems; in fact, to our knowledge, few works deal with multi-level caches and only particular setups of multi-level instruction [15] and data caches [20] have been considered so far, evidencing the dimension of the challenge.

While static timing analysis techniques demand caches that are deterministic in their temporal behaviour, the introduction of Probabilistic Timing Analysis (PTA) techniques [9], [14], [7], [8] changes the requirements to be accomplished by caches. In particular, PTA requires caches to have a *time-randomised behaviour that allows computing the probability of hit or miss of every cache access*. In the case of multi-level caches, this imposes that the events affecting the timing behaviour of an access, such as the outcome of other accesses or inclusivity requests coming from the upper cache levels, must have a probabilistic nature.

In this paper we analyse for the first time the worst-case timing behaviour of multi-level time-randomised caches. Our analysis, which scales to an arbitrary number of cache levels, covers unified data and instruction caches, different write, write-allocation and inclusion policies among the different levels. Our analysis builds upon some of the properties of time randomised caches. First, the particular memory address of an access does not determine the cache set in which it is mapped since the placement in cache is time randomised [16]. And second, the hit/miss probability of a given access to an address $@_x$ only depends on probabilistic events such as whether the accesses between the current and the last access to $@_x$ miss in cache. Based on these properties, our analysis focuses on identifying those events that cause accesses to the different cache levels and their probability of occurrence.

We evaluate 2-level time-randomised cache setups with a unified second level cache (shared among data and instructions). We consider inclusive and non-inclusive caches, as well as write-through and write-back first level caches. Our results prove that (1) multi-level time-randomised caches fulfil the requirements of measurement-based PTA (MBPTA) [8]. We also show how multi-level time-randomised caches decrease execution time by 30% on average. Reduction in terms of WCET estimates is even higher (55% on average) since multi-level caches also reduce the probability of pathological cache behaviour resulting in large execution times, hence execution time variability, with respect to single-level cache setups.

The rest of the paper is organised as follows. Section II provides background on PTA and time-randomised caches. Section III reviews time-randomised single-level caches. Section IV provides models proving the suitability of multi-level time-randomised caches in the context of PTA. Results are presented in Section V. Section VI reviews some related work. Finally, Section VII presents the conclusions of this work.

II. BACKGROUND

This section provides some background on PTA as well as the cache characteristics and assumptions in this paper.

A. Probabilistic Timing Analysis (PTA) Approaches

Probabilistic Timing Analysis (PTA) [14][9][7][8] provides WCET estimates with an associated probability of occurrence, called probabilistic WCET (pWCET) estimates. A pWCET estimate can be exceeded with a given probability, thus leading to a timing failure. This is analogous to the behaviour of hardware, for instance, which may fail with a given probability. In that sense, PTA extends the notion of probability of failure to timing correctness. PTA aims at obtaining pWCET estimates for arbitrarily low probabilities, so that even if that pWCET estimate can be exceeded, it would be exceeded with low probability (e.g. in the region of 10^{-15} per hour of operation, largely below the probability of hardware failures).

PTA can be implemented either in a static (SPTA) [7] or measurement-based (MBPTA) [8] manner. In this paper we focus on MBPTA as it is closer to industrial practice to compute WCET. MBPTA derives pWCET estimates for a program based on a collection of end-to-end observed execution times on a time-randomised architecture for which an ETP can be derived for each instruction. MBPTA applies Extreme Value Theory (EVT) [19], a well-known statistical method that, based on the inverse cumulative distribution function (ICDF) of the observed execution times, provides the probability that the execution time of a given instance of a program exceeds a threshold pWCET estimate.

Under both PTA approaches, the probabilistic timing behaviour of an *execution component* can be represented with an Execution Time Profile (ETP). At the smallest granularity, an execution component represents an access to a resource and at the highest, the entire program. In between, we can find instructions (which may access several resources), basic blocks, functions, etc. An ETP defines the discrete probability distribution function of execution times. Thus, an ETP defines, for the different execution times of a program (or latencies of an instruction), their corresponding probabilities. Hence, the timing behaviour of a program/instruction is described by the probability mass function: $(\overrightarrow{l}, \overrightarrow{p}) = (\{l_1, l_2, ..., l_k\}, \{p_1, p_2, ..., p_k\})$, where p_i is the probability the program/instruction taking latency l_i, accomplishing that $\sum_k^{i=1} p_i = 1$. For instance, in the case of cache read accesses in a simple single-level cache hierarchy, the ETP is as follows: $ETPmemop = (\{l_{hit}, l_{miss}\}, \{p_{hit}, p_{miss}\})$, where l_{hit} and l_{miss} are the hit and miss latencies respectively and p_{hit} and p_{miss} are the probabilities of occurrence of a hit and a miss respectively.

Unlike SPTA, which needs to know the exact ETP of each instruction, *MBPTA, which has been proven suitable for industrial practice [26], only requires the ETP for each instruction to exist*. The fact the ETPs exist ensures that the execution components contributing to the execution time of a program, i.e. instructions, behave as independent and identically distributed (i.i.d.) random variables[1]. Since instruction latencies occur with a given probability (described in their ETPs), all feasible program execution times also occur with a given probability, so an ETP for the whole program also exists. Thus, program execution time can be modelled with an i.i.d. random variable (described by its ETP) and EVT requirements are hence fulfilled [8].

Dependences among instructions. MBPTA works in the presence of *dependences among instructions*. For instance, the probability of hit/miss of a given access to $@_A$ may depend, for instance, on the number accesses to the cache between $@_A$'s current and previous access. MBPTA only requires those *causal dependences to be either (i) fully systematic*, meaning that they appear every time those instructions are executed, *or (ii) probabilistic*, meaning that there is a probability that the dependence manifests and hence, the execution time of dependent instructions is affected in a probabilistic manner. In both cases a valid ETP can be still derived to describe the timing behaviour of the instructions with the dependences. If each instruction has an ETP, measurements (execution times) obtained by running the program capture probabilistically the effect of any dependence among them. MBPTA imposes the execution time observations obtained by running the program on the target platform to be modelled with i.i.d. random variables. The existence of an ETP per processor instruction ensures that the i.i.d. properties for the execution time observations are fulfilled [7]. Further details can be found in the Annex in [18].

Input-data-dependent memory accesses. MBPTA handles data-dependent memory accesses by making the compiler flagging them so that they are forced to miss in cache, preventing the cache behaviour from being affected by the particular data provided by the user. By means of some 'hint' bits for memory operations already present in most current Instruction Set Architectures (ISA), the processor can be notified of whether a memory operation is input data dependent, preventing cache or other performance improving features from shortening the latency for these instructions [8].

Both PTA methods demand for a new type of cache designs, i.e. time-randomised (TR) caches [16], that guarantee that ETPs exist for all memory accesses since hits/misses occur with a given probability. In this paper we analyse, for the first time, different time-randomised multi-level cache designs, with and without inclusion under different write miss policies as well as unified data and instruction caches.

B. Cache Characteristics and Assumptions

In a multi-level cache design, *inclusivity* of the lower cache levels (those closer to the cores) into the upper cache levels (those closer to memory), imposes that all contents in the

[1]Two random variables are said to be independent if they describe two events such that the occurrence of one event does not have any impact on the occurrence of the other event [10]. Two random variables are said to be identically distributed if they have the same probability distribution [10].

lower level cache are also included in the upper level cache[2]. This implies that, whenever a cache line is evicted from the upper level cache, all cache lines in the lower level cache holding some or all contents of the cache line evicted in the upper level cache, are also evicted. When *exclusivity* is applied[3], cache lines can be stored *only* in one of the two levels involved. When a new cache line is fetched by the processor, it is typically fetched into the lower level and removed from the upper level. When a cache line is evicted from the lower level it is moved up to the next level. *Non-inclusive* caches are those where no constraint is imposed on whether cache lines are stored in upper or lower cache levels. This is a common choice for instruction caches since they are typically read-only and, thus, cache lines can be simply removed on an eviction.

Upper cache levels can be either shared among data or instructions or kept private. While private caches have been regarded as easier to analyse, unified (shared) ones are the most common choice due to their lower overheads. Thus, unlike previous works, we enable for the first time the analysis of unified upper cache levels storing data and instructions.

Write operations introduce complexities in the behaviour of the cache that are handled with a cache-write policy and a write allocation policy. There are two main write policies, namely, *write-through* (WT) and *write back* (WB). In the former, write operations occur in the current cache and are forwarded to the next cache level so that both caches hold consistent data. In WB caches, write operations occur only in the lower level cache, and the update of the next level is postponed until the cache lines containing the *dirty* data are evicted from the lower level cache. There are two write allocation policies. With *write allocate* (WA), on a write miss, data are fetched into cache, as it is the case for read misses, and, once fetched, the write operation occurs. With *no-write allocate* (nWA), on a write miss, the write operation is simply forwarded to the next cache level (or memory). Both WT and WB can use either of these write-allocation policies, but we only consider WB-WA and WT-nWA caches, since they are the most common choices. Though, nothing prevents our analysis to be extended to other combinations.

III. Time Randomised Single-level Caches

There are two sources of determinism in a cache that are randomised for MBPTA to work: the placement and replacement policies. Placement is relevant for direct-mapped and set-associative caches, whereas replacement is relevant for set-associative and fully-associative ones. In [16] authors propose random placement and random replacement policies that are PTA-compliant. The random replacement (RR) policy must ensure that every time a memory request misses in cache, a way in its corresponding cache set is randomly selected and evicted to make room for the new cache line. The random placement policy in [16] proposes a new parametric hash function that makes use of a random number as an input. Such random number can be generated either by hardware or software. The hash function, given a memory address and a random number called random index identifier (RII), provides a unique and constant cache set (mapping) for the address along the execution. If the RII changes, the cache set in which the address is mapped changes as well, so cache contents must be flushed for consistency purposes. Authors in [16] propose changing the RII only across program execution boundaries so that programs can be analysed with end-to-end runs without any further consideration than assuming that the cache is initially empty. The hash function proposed in [16] ensures that, given a memory address and a set of RIIs, the probability of mapping such address to any given cache set is the *same*.

For each access to a first level cache we can derive a probability of hit [16]. In general, for a cache with S sets and W ways, given the sequence $< A_i, B_1, ..., B_k, A_j >$, where A_i and A_j correspond to accesses to the same cache line and no B_l (where $1 \leq l \leq k$) accesses the same cache line as A_j, the probability of A_j to miss in cache can be approximated as:

$$P_{miss_{A_j}}(S,W) = \left(1 - \left(\frac{W-1}{W}\right)^{\sum_{l=1}^{l=k} P_{miss_{B_l}}}\right) \cdot \left(1 - \left(\frac{S-1}{S}\right)^{k}\right) \quad (1)$$

In Equation 1, the first element is due to the random replacement, while the second is the element due to placement. In fact, the first element is the probability of miss in a fully-associative cache with W ways deploying random replacement (*Prob1*). The second part of the equation corresponds to the probability of miss in a direct-mapped cache with S sets (*Prob2*). Given that placement and replacement work independently, the probability of miss in a set-associative cache with S sets and W ways deploying both random placement and replacement can be computed as the product of the probabilities *Prob1* and *Prob2*.

Note, that $P_{miss_{A_j}}$ is an approximation to the actual miss probability of A_j. Annex I, explains how the actual $P_{miss_{A_j}}$ can be derived for a simple example and shows that deriving the actual miss probability requires a more complex formulation than the one we use in this paper and which we leave as future work. For the purpose of proving that MBPTA can be applied, the use of our approximations is enough, since MBPTA only requires hit/miss events to have a probability of occurrence. Note that this is not the case for SPTA, which would require the actual miss probability, or a least a safe upper-bound to it, to be computed.

$P_{miss_{A_j}}$ is used to compute the ETP of each cache access as follows, where l_{hit} and l_{miss} are the cache hit and miss latencies respectively:

$$ETP_{cache} = \{\{l_{hit}, l_{miss}\}, \{1 - P_{miss_{A_j}}(S,W), P_{miss_{A_j}}(S,W)\}\} \quad (2)$$

[2]Note that contents may not be up-to-date in both caches if write operations are not propagated immediately as explained later, but at least an older version of the data is in place in both caches.

[3]We use the term exclusive cache rather than non-inclusive, because the latter just implies that inclusivity is not controlled, which is different than requiring exclusivity.

TABLE I. EVENTS IN A NIC CACHE HIERARCHY WITH WT-NWA L1 AND WB-WA L2

Event id	latency	L1	L2	L2 dirty?	actions	probabilities
1)	Lat_{ld1}	L1 ld hit			(a) Send data from DL1 to the core	$P^{@A}_{L1,hit}$
2)	Lat_{ld2}	L1 ld miss	L2 ld hit		(b) Send data from UL2 to DL1 and the core	$P^{@A}_{L2,hit} \times P^{@A}_{L1,miss}$
3.1)	Lat_{ld3}		L2 ld miss	L2 dirty	(c) Write dirty line to mem, (d) load new line into L2 and (b)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss} \times P^{vctm}_{L2,dirty}$
3.2)	Lat_{ld4}			L2 clean	(d) and (b)	$P^{@A}_{L2,miss} \times P^{vctm}_{L1,miss} \times P^{vctm}_{L2,clean}$
1)	Lat_{st1}	L1 st hit	L2 st hit		(e) write data into L1, (f) write data into L2	$P^{@A}_{L2,hit} \times P^{@A}_{L1,hit}$
2.1)	Lat_{st2}		L2 st miss	L2 dirty	(c), (d), (e) and (f)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,hit} \times P^{vctm}_{L2,dirty}$
2.2)	Lat_{st3}			L2 clean	(d), (e) and (f)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,hit} \times P^{vctm}_{L2,clean}$
3)	Lat_{st4}	L1 st miss	L2 st hit		(f)	$P^{@A}_{L2,hit} \times P^{@A}_{L1,miss}$
4.1)	Lat_{st5}		L2 st miss	L2 dirty	(c), (d) and (f)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss} \times P^{vctm}_{L2,dirty}$
4.2)	Lat_{st6}			L2 clean	(d) and (f)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss} \times P^{vctm}_{L2,clean}$

IV. TIME RANDOMISED MULTI-LEVEL CACHES

In this section we focus first on random-replacement fully-associative multi-level caches for the sake of clarity. In Section IV-C we extend our analysis to random placement featured in set-associative and direct-mapped caches. With the same aim and without loss of generality we focus on a 2-level cache hierarchy. In the first level we find an instruction (IL1) and a data cache (DL1). In the second level we have a unified L2 cache (UL2). We generalise our analysis for more than 2 levels also in section IV-C. We consider a WT-nWA DL1 and a WB-WA UL2 caches, as this is a very common organisation and allows us reasoning about both types of cache-write and write allocation policies. Note that cache-write and write allocation policies are irrelevant for IL1, as its contents are read-only. We provide the analysis for both non-inclusive and inclusive caches, as they are the common case. Later in Annex II we also describe the case of exclusive caches. Considerations related to the hardware implementation are described in section IV-D.

In the remaining of this section we refer to the probability of an $< event >$, such as a hit or a miss, in each cache level as: $P^{<op>}_{<L>,<event>}$, where L is the cache level, i.e. IL1, DL1 or UL2, $< op >$ is the type of access, i.e. load (ld) [4] and store (st), or the address of the access when the type of operation does not affect its probability. Each probability is provided under a cache configuration that determines the write, allocation and inclusivity policies.

For computing the probabilities in the probability vector of the ETP of every memory operation, we consider the following sequence of accesses: $< I_0^{mop@A}\ I_1^{ld1}\ I_2^{st1}\ I_3^{ld2}...\ I_r...\ I_s^{stm}...\ I_t^{ldn}\ I_{t+1}^{mop@A} >$, where the subindex is the instruction id, the superindex indicates the number of load and store operation, and where $I_{t+1}^{mop@A}$ is the memory operation whose hit/miss probability we want to derive for each cache organisation. Note that mop stands for any memory operation (either ld or st). We use it in those cases where the particular type of memory operation being considered is irrelevant. The load and store operations between the accesses to $@_A$ access different cache lines in DL1, IL1 and UL2 to those where $@_A$ and the instruction loading $@_A$ are stored.

[4] A load can be an instruction load sent by the instruction cache or a data load sent by the data cache.

A. No Inclusivity Control (NIC)

When no inclusivity control is used among the different cache levels, the hit accesses and the evictions carried out in one level have no impact on previous or next cache levels. Table I shows the different events in each level of the cache, the actions taken in the cache on that event and the associated probability.

When a load access hits in DL1 (1), which happens with a probability $P^{@A}_{DL1,hit}$, data are sent to the core. In case of miss in DL1 and hit in UL2 (2), data are sent from UL2 to DL1 and the core. Given that the RIIs used for each cache are different as explained later, their placement functions are different and thus, the events 'hit in DL1' and 'hit in UL2' are independent. Hence, the probability of both events to occur can be obtained by multiplying their respective probabilities. In case of a miss in both DL1 and UL2 (3), data are loaded from memory to UL2 and DL1, and sent to the core. If the line evicted is dirty[5] (3.1), it is written back to memory before the new line is loaded from memory to UL2. If it is clean (3.2) no line is written back. Note that the instruction cache (IL1) events are the same as the load events for the DL1.

Stores update DL1 when they hit and are always forwarded to UL2. If they hit in UL2, UL2 is updated (1). In case of miss in UL2 (2), first a cache line (victim) is selected for eviction and, if it is dirty (2.1), which happens with a probability $P_{L2,dirty}(vctm)$, it is written back to memory. Then, the new line is fetched from memory into UL2 and it is updated with the data carried out by the store operation. If the line was clean (2.2), the same actions, but writing the victim to memory, are carried out.

In case of miss in both DL1 and UL2 (3), the line is written into UL2, but not brought to DL1. In case of miss in both (4.1 and 4.2), the evicted line is written to memory in case it is dirty, the new line is brought into UL2 and updated with the new data.

The most important appreciation from Table I is that *all the events that may potentially happen in all caches have an associated probability* ($P_{UL2,hit/miss}(@)$, $P_{IL1,hit/miss}(@)$, $P_{DL1,hit/miss}(@)$ and $P_{UL2,dirty}(@)$). Next, we derive how

[5] Write operations in a WB cache make cache lines to be inconsistent with upper levels in the memory hierarchy, so on an eviction their contents must be updated in upper levels. Those lines are referred to as *dirty lines*.

TABLE II. EVENTS IN AN INCLUSIVE CACHE HIERARCHY WITH WT-NWA L1 AND WB-WA L2 AND THEIR ASSOCIATED PROBABILITIES

Event id	latency	L1	L2	L2 dirty?	actions	probabilities
1)	Lat_{ld1}	L1 ld hit			(a) Send data from L1 to the core	$P^{@A}_{L1,hit}$
2)	Lat_{ld2}	L1 ld miss	L2 ld hit		(b) Send data from UL2 to L1 and (a)	$P^{@A}_{L2,hit} \times P^{@A}_{L1,miss}$
3)			L2 ld miss		(c) Check inclusivity of evicted line	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss}$
3.1)	Lat_{ld3}			L2 dirty	(d) write dirty line (victim) to mem, (e) load new line into L2, (b) and (a)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss} \times P^{vctm}_{L2,dirty}$
3.2)	Lat_{ld4}			L2 clean	(e), (b) and (a)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss} \times P^{vctm}_{L2,clean}$
1)	Lat_{st1}	L1 st hit			(f) write data into L1, (g) write data into L2	$P^{@A}_{L1,hit}$
2)	Lat_{st2}	L1 st miss	L2 st hit		(g)	$P^{@A}_{L2,hit} \times P^{@A}_{L1,miss}$
3)			L2 st miss		(c)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss}$
3.1)	Lat_{st3}			L2 dirty	(d), (e) and (g)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss} \times P^{vctm}_{L2,dirty}$
3.2)	Lat_{st4}			L2 clean	(e) and (g)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss} \times P^{vctm}_{L2,clean}$

those probabilities can be approximated. Note, however, that, assuming that core operations can be analysed probabilistically, the fact that cache events are probabilistic makes the application of MBPTA correct, since EVT makes no assumption on the particular probability distribution function of the events under consideration or on whether they depend on each other.

A.1. $P_{DL1,miss}(@)$. The miss probability of an access to $@_A$ in a WT-nWA DL1, $P_{DL1,miss}(@_A)$, with a WB-WA UL2 with no inclusivity control is affected by the intermediate accesses carried out in between that access to $@_A$ and the previous access to $@_A$. In particular, $P_{DL1,miss}(@_A)$ is given by the number of memory operations between both accesses to $@_A$, and the probability of miss of each access. The higher the number of accesses and their respective miss probabilities, the higher the probability of mis of the second access to $@_A$, which can be approximated as:

$$P_{DL1,miss}(@_A) = 1 - \left(\frac{W_{DL1}-1}{W_{DL1}}\right)^{\sum_{i=1}^{n+m} P_{DL1,miss}(DL1acc_i)}$$

where W_{DL1} is the number of ways in DL1[6], $n+m$ is the number of intermediate loads and stores, and $P_{DL1,miss}(DL1acc_i)$ their associated miss probabilities.

A.2. $P_{UL2,miss}(@)$. The probability of missing in UL2 for a given access to address $@_A$ (regardless of whether it is a data or instruction address) is given by the number of accesses performed to UL2 between the current access and the previous access to $@_A$. This includes both, UL2 instruction accesses coming from the IL1 and UL2 data accesses coming from DL1 as shown in the formula below, where W_{UL2} is the number of lines in the UL2.

$$P_{UL2,miss}(@_A) = 1 - \left(\left(\frac{W_{UL2}-1}{W_{UL2}}\right)^{\sum_{i=1}^{t} P_{miss,IL1}(I_i) \times P_{miss,UL2}(I_i)} \times \left(\frac{W_{UL2}-1}{W_{UL2}}\right)^{\sum_{i=1}^{n+m} P_{miss,DL1}(mop_i) \times P_{miss,UL2}(mop_i)} \right)$$

[6]Recall that in this section we focus on fully-associative caches, so the number of cache ways of the cache is also the number of cache lines.

The exponent in the first element in the formula above accounts for the effect of instruction misses (between both accesses to $@_A$) in the IL1 (and hence accesses to the UL2) that also miss in UL2. The exponent in the second element is the probability of a memory operation to miss in the DL1 and UL2. We can simplify the above formula as $P_{UL2,miss}(@_A) = 1 - \left(\frac{W_{UL2}-1}{W_{UL2}}\right)^{\sum_{i=1}^{k} P_{UL2,miss}(L2acc_i)}$, where the exponent is the accumulated miss probability of all UL2 accesses between the current and the previous access to $@_A$.

A.3. $P_{UL2,dirty}$. When $@_A$ misses in UL2, a random line in the corresponding set is selected for eviction. There is a probability that the selected line is dirty. While hit and miss probabilities only depend on the accesses in between the current and the previous accesses to a particular address, $P_{UL2,dirty}$ depends on all past accesses since the beginning of the execution of the program (assuming an initial empty cache state). Therefore, and only for approximating $P_{UL2,dirty}$ we consider the following sequence: $<I_0^{mop@_0}, I_1^{mop@_1}, ..., I_{i-1}^{mop@_{i-1}}, I_i^{mop@_A}>$, in which we assume that $I_i^{mop@_A}$ misses in cache, resulting in a cache line being randomly evicted. We obtain $P_{UL2,dirty}(I_i)$, that is, the probability of a dirty line to be evicted when I_i is executed, as the fraction of dirty lines in the cache set where $@_A$ is when the second access to $@_A$ occurs. In other words, the accumulated probability that each line in that set has not been evicted from cache since it was last accessed by a store operation.

$$P_{UL2,dirty}(I_i) = \sum_{j=0}^{i-1} P^{dirty}_{surv}(@_j, i)$$

In the equation above, $P^{dirty}_{surv}(@_j, i)$ is the probability of instruction I_j to leave a dirty line in cache and this line to be still present when instruction I_i (the one accessing $@_A$) is executed. It is defined as follows:

$$P^{dirty}_{surv}(@_j, i) = \begin{cases} 0 & \text{if } isload(I_j) \\ \left(\frac{W_{L2}-1}{W_{L2}}\right)^{\sum_{k=j+1}^{i-1} P_{miss}(@_k)} & \text{if } isstore(I_j) \end{cases}$$

In the equation above, $isload(I_k)$ is true when I_k is a load instruction, similarly $isstore(I_k)$ is true when I_k is a store. Lines dirtied by stores survive with a given probability that depends on the number of misses between them and the access to $@_A$.

B. Inclusive Caches

The main difference between inclusive caches with respect to caches without inclusivity control (NIC) is that UL2 evictions may require evicting some cache lines in DL1. If a line being evicted from UL2 is present in DL1, it is also evicted from DL1. Note that inclusivity is typically deployed only for data caches since this simplifies hardware design to deal with either write operations in write-through caches or dirty lines evicted in write-back ones. Instruction caches typically do not support any type of write operation (other than filling cache lines when fetched), so there is no need for making them inclusive.

Table II shows the different events that may happen to a cache access and their associated probability. On an UL2 miss, a victim is selected to be evicted. In addition to checking whether it was dirty, in which case it is written back to memory, it must be checked whether that line is present in DL1 (in fact in all L1 inclusive caches), in which case it is invalidated from the corresponding L1 cache. As DL1 is WT, the invalidation consists simply in setting an 'invalid' bit (no further transaction is initiated). Therefore, we consider that the latency of checking for invalidations is the same regardless of whether a line is finally invalidated. How to deal with WB caches and dirty lines is later described in Section IV-C. We also observe that in inclusive caches the event 'DL1 hit and UL2 miss' is not possible, since all DL1 contents are also present in the UL2.

As for NIC caches, the events that may potentially happen in the different caches have an associated probability ($P_{UL2,hit/miss}(@)$, $P_{IL1,hit/miss}(@)$, $P_{DL1,hit/miss}(@)$ and $P_{UL2,dirty}(@)$). The value of some of those probabilities change with respect to the NIC case as detailed next.

B.1. $P_{DL1,miss}(@)$. The probability of miss in DL1 of an access to $@_A$, $P_{DL1,miss}(@_A)$, with an inclusive UL2 is affected by the accesses carried out in between that access to $@_A$ and its previous access, and the probability of miss of those intermediate accesses. There are two types of accesses to the DL1 that can happen. First, *memory accesses (MA)* between the two accesses to $@_A$ sent from the core to the DL1. And second, *Data Inclusivity Requests (DIR)* sent from the UL2 to DL1 due to the UL2 accesses between the two accesses to $@_A$ that evict lines from UL2, thus causing a subsequent eviction in DL1 of the line evicted from UL2.

The number of data memory accesses sent to UL2 from the core are given by $P_{MAmiss} = \sum_{i=1}^{k} P_{DL1,miss}(mop_i)$, where mop_i are the loads and stores in between the two accesses to $@_A$ and $P_{DL1,miss}(mop_i)$ is the miss probability of each access computed as for single-level caches. Analogously, instruction accesses sent to UL2 are given by $P_{Imiss} = \sum_{i=1}^{k} P_{IL1,miss}(I_i)$, where I_i stands for any instruction in between the two accesses to $@_A$ and $P_{IL1,miss}(I_i)$ is the miss probability of each such instruction computed as for single-level caches.

When a data or instruction access between two accesses to $@_A$ causes a UL2 miss, this generates a UL2 eviction which can evict $@_A$ from UL2, which would imply removing $@_A$ from DL1 to keep inclusivity. The number of DIRs is given by number load and store operations between two accesses to $@_A$ that miss in DL1 plus the number of instructions fetched between those two accesses to $@_A$ that miss in IL1. Note that though the UL2 is inclusive of DL1, hits to DL1 are ensured to also hit in UL2, hence not generating any eviction that could evict the line in UL2 where $@_A$ is. This is particularly important for store operations that access UL2 regardless of whether they hit DL1. We approximate the accumulated probability due to inclusivity evictions as follows:

$$P_{DIRev} = \sum_{i=1}^{n+m}\left(P_{DL1,miss}(mop_i)\times P_{UL2,miss}(mop_i)\times\frac{1}{W_{UL2}}\right)+ \sum_{i=1}^{t}\left(P_{IL1,miss}(I_i)\times P_{UL2,miss}(I_i)\times\frac{1}{W_{UL2}}\right)$$

where $n+m$ is the number of loads and stores and t the number of instructions between both accesses to $@_A$. The first element in the first row of the equation is the probability that any data access in between two accesses to $@_A$ misses in the data cache. The second element is the probability that those DL1-missing accesses, that access the UL2, miss in the UL2. The last element is the probability that each evicted L2 cache line contains $@_A$. The second row of the formula is analogous for IL1.

Overall, the miss probability of $@_A$ in DL1 can be approximated as $P_{DL1,miss}(@_A) = 1 - \left(\frac{W_{DL1}-1}{W_{DL1}}\right)^{P_{MAmiss}+P_{DIRev}}$.

B.2. $P_{UL2,miss}(@)$. There is a probability that an access to $@_A$ in the UL2 (regardless of whether it is a data or an instruction access), and so its $P_{UL2,miss}(@_A)$, is affected by the inclusivity policy. An instruction with instruction address (i.e. Program Counter) $@_A$ accesses UL2 if it misses in IL1. Since DL1 contents are included in UL2, the access $@_A$ cannot hit in those W_{DL1} lines of the UL2 keeping the contents of the DL1. Similarly, a DL1 miss cannot hit in the UL2 lines keeping the contents of DL1. Hence, $@_A$ can only hit in $W_{UL2} - W_{DL1}$ lines, which we call $W_{UL2nonDL1}$ being its miss probability approximated by:

$$P_{UL2,miss}(@_A) = 1 - \left(\frac{W_{UL2nonDL1}-1}{W_{UL2nonDL1}}\right)^{\sum_{i=1}^{k} P_{UL2,miss}(L2acc_i)}$$

where the exponent is the miss probability of the accesses between $@_A$ and its last access.

B.3. $P_{UL2,dirty}$. It is not affected by the inclusivity control, so it remains as described for the NIC case.

C. Generalisation of the Latency/Probability Cache Model

For the sake of simplicity, we have assumed that all accesses in the different sequences we have used to describe the hit/miss probability of each event, i.e.

$<I_0^{mop@A}\ I_1^{ld1}\ I_2^{st1}\ I_3^{ld2}...\ I_r...\ I_s^{stm}...\ I_t^{ldn}\ I_{t+1}^{mop@A}>$ and $< I_0^{mop@_0}, I_1^{mop@_1}, ..., I_{i-1}^{mop@_{i-1}}, I_i^{mop@_A} >$, go to a different cache line each, but the first and last access to $@_A$ that access the same line. Considering the case in which intermediate accesses may access the same cache line addresses just makes the computation of probability approximations more complex, since events are probabilistically dependent; however, *the events affecting the timing behaviour of the cache are still probabilistic as required for the application of MBPTA* (see Annex I). Recall, that MBPTA makes no assumption on the probability distribution function of any random event. Note also, that the probabilities computed assuming that accesses are assumed to go to different addresses represent an upperbound of the actual probabilities when they may go to the same address line. This is so, because when two accesses go to the same address, their reuse distance reduces and so do their miss probabilities.

Besides that, there are several dimensions in which our model can be generalised. First, random placement since in our analysis above we consider only random replacement. Second, different cache line sizes between different levels. Third, considering more than 2 cache levels. And fourth, different inclusivity arrangements between different cache levels.

Random Placement. Random placement requires taking into account the probability that any of the k different (unique) accesses between the two accesses to $@_A$ access the set where $@_A$ is placed. This is given by the formula $\left(\frac{S-1}{S}\right)^k$, where S is the number of sets. As we have done for the random replacement in previous sections, for each cache level we can compute the number of those potential accesses. For the IL1 these are the number of unique instruction requests between the two accesses to $@_A$. For the DL1 we have the number of unique memory operations plus the number of unique data inclusivity requests. The UL2 is accessed as many times as the number of DL1 and IL1 misses are experienced. The effect of the placement on the probability of hit of accesses can be multiplied by the effect of replacement computed in previous sections as both are independent [16].

Different cache line sizes. So far we have considered the case where all caches use the same cache line size. However, it may be the case that cache lines in the upper level are larger than those in the lower levels. Let us assume that UL2 lines are *q* times larger than those of the DL1 and IL1. There are two main ways in which different cache line sizes in each level affect the probability of hit/miss of each access. First, the distribution of accesses on the different cache lines changes. For instance, while accesses $@_B$ and $@_C$ in the sequence $< @_A @_B @_C @_A >$ access different cache lines under a line size setup, they can access the same line if the cache line size is increased. In the latter case, the probability of evicting $@_A$ is smaller since $@_C$ will always hit and hence, will never produce an eviction. This simply reduces the miss probability of $@_A$. Note, however, that if the cache size remains constant, increasing the line size implies reducing the number of sets or ways, which will increase the probability of miss of $@_A$. In any case, hit/miss events remain probabilistic regardless of the cache line sizes and hence, analysable with MBPTA. Second, for some inclusivity control policies, when a dirty line is evicted from UL2, the contents of that line have to be evicted from DL1. If UL2 lines are larger than DL1 ones and hence, contain several DL1 lines, this would produce a potentially larger number of invalidations, reducing the probability of hit of DL1 accesses.

Several cache levels. For setups with several (more than 2) cache levels, the only difference in our analysis consists in taking into account, when analysing a given cache level L_i, the accesses that any other cache level can introduce on L_i. This depends on the inclusivity arrangement selected and the write-miss and allocation policies of each level. This would make the number of events to consider higher, but each event would be still fully probabilistic. As a result, an ETP for each cache access still exists, and hence, MBPTA can be applied.

Inclusivity Arrangements. Similarly, to the previous case, inclusivity policies affect the number of accesses that a given program does to the different cache levels. It also affects, the actual size available in a given level. For instance, when the DL1 is inclusive of the UL2, every miss in DL1 that becomes an access to the UL2, can only hit in the UL2 lines not devoted to keep DL1 information. As we have seen, this can easily be taken into account in the analysis. Analogously, if DL1 is WB and inclusive, dirty lines may be evicted. This may have an impact in latency. To consider this, one should split the case of DL1 evictions into 2 subcases considering whether a dirty line from DL1 is evicted or not. Deriving such probability of dirtiness in DL1 is analogous to the case of UL2.

Overall, the effect of all these variations is purely probabilistic and therefore, analysable with MBPTA. MBPTA does not need to compute ETPs and hence, it is enough those events to be probabilistic to ensure ETPs exist, which is in turn enough to apply MBPTA.

D. Hardware Considerations

In random placement multi-level caches it is important guaranteeing that placement choices in every cache are independent to prevent any correlation between the random events in the different caches. This is achieved by simply using different RII values for each cache. Therefore, those addresses conflicting in a particular cache set in a first level cache, thus producing some misses, are very unlikely to be placed in the same set in the second level cache so that the same conflicts do not repeat.

Regarding the overhead of the random placement and replacement caches, it has been shown to be low with respect to modulo-placement LRU-replacement: as shown in [16] the overhead in area and access time is very small and its relative impact further decreases for large caches such as UL2 ones.

V. EVALUATION

A. Experimental Framework

We focus on a single-core pipelined processor architecture in which instructions are fetched from the time-randomised instruction cache and sent to the decode stage. Once decoded, instructions are executed in a fixed latency and finally sent to the write-back stage. Our pipeline, similar to the LEON4 [1], incorporates bypasses to remove pipeline stalls due to dependences across instructions. During the execution stage, the time-randomised data cache is accessed. We model 4KB, 32-byte line, 4-way set-associative instruction (IL1) and data caches (DL1), both deploying random replacement and random placement [16]. The UL2 is a unified cache keeping data and instructions. It is 128KB with 32-byle lines and 8-way set associativity. The UL2 access latency is 10 cycles and the latency to access memory 100. The DL1 deploys WT-nWA policies and the UL2 is WB-WA.

We use several inclusivity arrangements: a first setup where we make DL1 inclusive of the UL2 (*L1-L2inc*) and a second setup in which we do not exercise any inclusivity policy (*L1-L2nic*). We also evaluate the effect of making the DL1 write-back when inclusivity is exercised (*L1-L2wb*). DL1 and IL1 caches are connected to the UL2 through fully-dedicated bidirectional buses, whose access latency can be bounded using the technique presented in [24].

The objective of our analysis is to effectively reduce the pWCET estimates that can be obtained for programs when several levels of cache are used. Of course, only on those cases in which the average execution time of the program reduces when several levels of cache are deployed, we can expect some reduction in the pWCET. As a reference point we use a setup with a single level of cache in which DL1 and IL1 have the same size they have in the other setups, i.e. 4KB.

We use the EEMBC Autobench benchmark suite [25], which is a well-known suite reflecting the current real-world demand of some automotive embedded systems.

B. Compliance with MBPTA requirements

On the one hand, the core architecture presented in previous section has been shown to be MBPTA compliant [16]. On the other hand, we can derive a probability of every event affecting the timing behaviour of a cache access (see Annex I), or approximate such probabilities (see Section IV) . As a result, for each instruction an ETP exists, which makes our multi-level cache processor architecture MBPTA-compliant by construction. This is so, because (1) the latency of each instruction can be modelled with i.i.d. random variables, and (2) the execution of a sequence of instructions leads to another ETP, i.e. random variable, which at the coarsest granularity level represents the ETP of the program.

We apply statistical tests to show that the execution times of the program fulfil i.i.d. requirements. To that end, we made 1,000 runs of each program in our multi-level cache platform, which proven to be enough runs according to MBPTA [8]. We test independence with the Wald-Wolfowitz test [6] using a 5% significance level (a typical value for this type of tests). If the absolute outcome obtained after running this test is below 1.96 independence hypothesis cannot be rejected. For identical distribution, we use the two-sample Kolmogorov-Smirnov identical distribution test [5] as described in [8]. For 5% significance, the outcome provided by the test should be above the threshold (0.05) to indicate identical distribution, and non-identical distribution otherwise.

TABLE III. INDEPENDENCE AND IDENTICAL DISTRIBUTION TESTS RESULTS (OUTCOME INDEPENDENCE TEST / OUTCOME I.D. TEST).

Benchmarks	L1-L2 INC	L1-L2 NIC	L1-L2 WB	L1 (only)
a2time	0.03/0.29	0.83/0.41	0.46/0.44	0.90/0.49
aifftr	0.71/0.74	0.95/0.33	0.82/0.59	1.19/0.33
aifirf	0.40/0.11	1.04/0.20	0.13/0.94	1.04/0.79
aiifft	0.68/0.32	0.50/0.41	0.96/0.17	1.09/0.94
cacheb	0.63/0.93	1.11/0.72	1.20/0.35	0.79/0.66
canrdr	0.79/0.16	0.75/0.54	1.00/0.37	0.32/0.91
iirflt	0.96/0.85	0.68/0.41	0.78/0.50	0.07/0.22
puwmod	1.39/0.67	0.99/0.25	0.94/0.75	0.30/0.71
rspeed	0.47/0.43	1.33/0.51	0.91/0.24	1.35/0.42
tblook	1.33/0.92	0.52/0.86	0.34/0.26	0.76/0.44
ttsprk	0.19/0.43	0.89/0.52	0.21/0.42	0.67/0.63

Table III shows the results of both tests for all EEMBC benchmarks under all multi-level cache configurations. As expected, both tests are passed in all cases so independence and identical distribution hypotheses cannot be rejected. This statistical results reinforce the probabilistic analysis we did in Section IV on the timing behaviour of multi-level caches.

C. Reduction in pWCET Estimates

We consider an exceedance probability of 10^{-15} per run. Our selection of the exceedance probability, i.e. the probability that an instance of a task misses its deadline, is based on the observation that for the aerospace commercial industry at the highest integrity level (DAL-A) the maximum allowed failure rate in a piece of software is 10^{-9} per hour of operation [2]. In current implementations, the highest frequency at which a task can be released is 20 milliseconds (180,000 times per hour) [2]. Hence, the highest allowed failure rate per task activation is 5.56×10^{-15}, which is above our exceedance probability.

The objective of our analysis is to effectively enable the use of multi-level caches such that significant reductions can be obtained in the pWCET estimates derived by MBPTA. The reduction that can be obtained depends on each application and in the particular use of cache that the application does. Applications requiring little cache space are very unlikely to benefit from having a UL2 cache in place in terms of average performance and pWCET estimates.

Figure 1 shows the average performance that each EEMBC obtains when the different cache setups are deployed. All results are normalised to the *single-level* cache setup. We observe that some benchmarks are quite insensitive to having

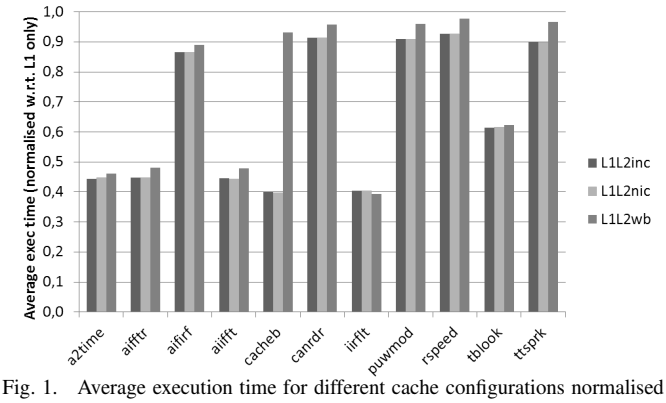

Fig. 1. Average execution time for different cache configurations normalised to the single-cache level setup

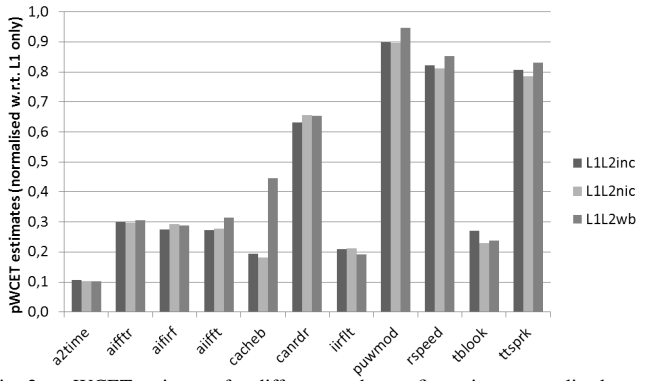

Fig. 2. pWCET estimates for different cache configurations normalised w.r.t. to the single-cache level setup

a two-level cache hierarchy achieving a small execution time reduction. Those benchmarks – `aifirf`, `canrdr`, `puwmod`, `rspeed` and `ttsprk` – achieve an average performance reduction in the range 5%-15%. This is mainly due to the fact that those benchmarks have a small code footprint and data working set that fit in IL1 and DL1 respectively. The rest of the benchmarks significantly benefit from having a UL2 cache. `a2time`, `aifftr`, `aiifft`, `cacheb`, `iirflt` and `tblook` achieve execution time reductions in the range 40%-60%.

It must be also noted that the difference between inclusive and non-inclusive caches is negligible. However, whenever the DL1 is WB and WA, execution time increases. This is particularly noticeable for `cacheb`, whose execution time doubles for the WB-WA configuration. The reason is as follows: when the DL1 is WT-nWA, store instructions can be served without stalling the pipeline. However, under a WB-WA configuration, if store operations miss in DL1, the pipeline is stalled until data are fetched into DL1. In general, this has a relatively low effect on benchmarks whose most store operations hit in DL1, thus not causing any stall. Only few store operations miss in DL1 and increase execution time. However, the store operations in `cacheb` often miss in DL1 and UL2, so they stall the pipeline for long periods of time, thus increasing execution time noticeably.

Figure 2 shows the pWCET estimates obtained for every EEMBC benchmark under each cache setup, normalised to the pWCET estimates for the single-level cache setup.

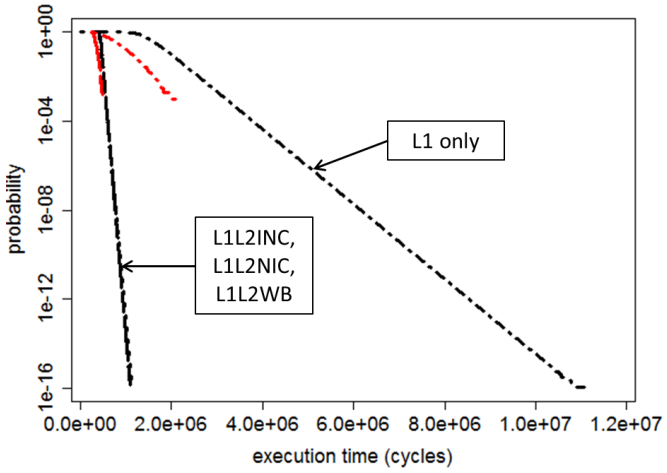

Fig. 3. pWCET distributions (black lines) and actual measurements (red lines), i.e. ICDF, for `a2time`.

pWCET reductions obtained with multi-level caches are more significant than those in terms of average performance. The average performance reduction is around 30% whereas the average pWCET reduction is around 55%. The reason for this behaviour lies on the fact that random placement may map different cache lines to the same set with a relatively high probability in L1 caches, thus causing misses and increasing execution time with non-negligible probability. This effect basically increases the probabilities of high execution times, so MBPTA accounts for that deriving Gumbel distributions [8] with a lower slope, which basically increases pWCET estimates as the exceedance probability decreases. This effect is detailed in the next section through particular examples. On the other hand, whenever a UL2 cache is in place, those conflicts that may arise in L1 caches in few executions have low impact in execution time because UL2 latency is much lower than that of main memory (10 versus 100 cycles). Moreover, since random placement functions in different caches are independent, L1 conflicts are extremely unlikely to also occur in UL2. Overall, whenever a UL2 cache is used, both execution time and execution time variation decrease, thus leading to much lower pWCET estimates since the Gumbel distribution slope is sharper.

D. Detailed pWCET Analysis

This section analyses in detail the effect of UL2 caches in pWCET estimates by considering exemplary benchmarks. In particular, we consider `a2time` and `canrdr`. The pWCET estimates we obtain are shown in Figures 3 and 4 respectively. In those figures we also show inverse cumulative distribution functions (ICDF)[7].

For both benchmarks, the observed execution times (in red and reaching only probabilities down to 10^{-3}) exhibit little

[7]The probability distribution function (or PDF) gives the probability of each execution time to occur. The cumulative distribution function (CDF) accumulates probabilities and the inverse CDF or ICDF, is computed as *1-CDF*.

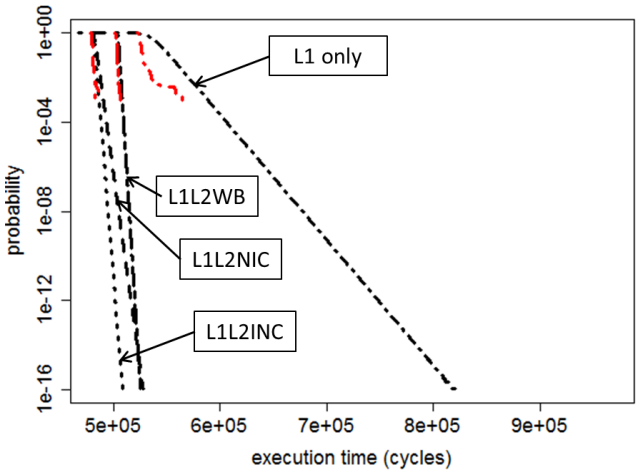

Fig. 4. pWCET distributions and actual measurements for `canrdr`.

variability when the UL2 cache is in place. This leads to sharp slopes for the Gumbel distributions describing the pWCET estimates for those benchmarks. In the case of `a2time` all setups with a UL2 have very similar average performance and pWCET distributions are practically identical. In the case of `canrdr`, execution time when UL2 is used exhibits somewhat higher (still low) variation. This creates some pessimism for pWCET estimates, as shown for the L1-L2 NIC setup, whose average performance resembles that of the L1-L2 INC setup, but whose pWCET estimate for an exceedance probability of 10^{-15} per run resembles that of the L1-L2 WB setup.

Finally, we observe how execution time observations for the *single-level* setup for both benchmark are higher and exhibit much higher variability. This leads to a right-shifted pWCET distribution with lower slope and thus, significantly higher pWCET estimates.

VI. Related Work

Timing analysis of systems equipped with cache memories is abundant [22], [12], [23], [11], [20], [15], [8], [7], [16], [17]. However, to the best of our knowledge, multi-level cache hierarchies are deemed as hard to analyse and few works have considered them [15], [20]. In [15] authors focus on instruction memory accesses on a 2-level non-unified deterministic cache architecture, while in [20] authors focus on data memory accesses on a non-unified cache hierarchy.

One commonality of all the approaches above is that they work on deterministic caches. One of the main characteristics of deterministic caches is that the particular addresses in which objects (i.e. code and data) are located plays a key role in cache performance. This makes that static cache analysis techniques have to deal with an increasingly complex challenge, namely, determining the run time addresses of each access, in addition to having an accurate model of the underlying hardware, i.e. the cache in our case [27]. While such information can be obtained for relatively simple programs, deriving run-time addresses, in particular for data accesses, can be regarded as unattainable for industrial-size applications [21]. Furthermore, hardware efficiency imposes some constraints on cache design, such as using unified second-level caches for data and instructions, considering different inclusion policies, and dealing with write-through and write-back caches as well as write-allocate and non-write-allocate caches.

PTA [9], [8], [7] responds to the need of enabling complex hardware in the context of Critical Real-Time Embedded Systems by obtaining trustworthy WCET estimates at low cost [8]. However, so far only single-level caches have been considered [8], [7], [16], [17]. In this paper we go over this limitation by enabling the use of multi-level caches in the context of PTA. In particular, we show how this can be done with arbitrary cache hierarchies including unified caches, different inclusion, cache-write and allocation policies.

VII. Conclusions

The increasing demand for performance in Critical Real-Time Embedded Systems (CRTES) pushes for the adoption of high-performance features such as multi-level cache hierarchies. However, deriving trustworthy and tight execution time upper-bounds in the presence of such features is deemed as expensive – if at all doable.– Therefore, there is a need for low-cost industrial-viable means to determine trustworthy and tight Worst-Case Execution Time (WCET) estimates in the presence of multi-level caches.

The advent of Probabilistic Timing Analysis (PTA) together with time-randomised caches has enabled the use of single-level cache memories in an industrial context at low cost. In this paper, we prove that multi-level time-randomised caches are also PTA-compliant by showing that the probabilities of the different events exist. In particular, and for the first time, we enable the use of unified data and instruction second-level caches, implementing different inclusion, cache-write and write-allocation policies without impacting the cost of the WCET estimation. Our results show that 55% average pWCET reductions can be achieved by enabling the use of multi-level caches for CRTES, which obviously decreases the hardware required to schedule critical tasks in CRTES.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the PROARTIS Project (www.proartis-project.eu), grant agreement no 249100. This work has also been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557 and the HiPEAC Network of Excellence. Leonidas Kosmidis is funded by the Spanish Ministry of Education under the FPU grant AP2010-4208. Eduardo Quiñones is partially funded by the Spanish Ministry of Science and Innovation under the Juan de la Cierva grant JCI2009-05455.

References

[1] *NGMP Preliminary Datasheet Version 1.6, August 2011 http://microelectronics.esa.int/ngmp/LEON4-NGMP-DRAFT-1-6.pdf.*

[2] Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. *ARP4761*, 2001.

[3] Aeroflex Gaisler. *Quad Core LEON4 SPARC V8 Processor - LEON4-NGMP-DRAFT - Data Sheet and Users Manual*, 2011.

[4] ARM Ltd. Cortex-A series, 2013. http://www.arm.com/products/processors/cortex-a/index.php.

[5] Sarah Boslaugh and Paul Andrew Watters. *Statistics in a nutshell*. O'Reilly Media, Inc., 2008.

[6] J.V. Bradley. *Distribution-Free Statistical Tests*. Prentice-Hall, 1968.

[7] F.J. Cazorla, E. Qui nones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim. Proartis: Probabilistically analysable real-time systems. *ACM TECS*, 2012.

[8] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Quinones, and F.J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS*, 2012.

[9] S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *RTSS*, 2001.

[10] W. Feller. *An introduction to Probability Theory and Its Applications*. John Willer and Sons, 1996.

[11] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise wcet determination for a real-life processor. *EMSOFT*, 2001.

[12] C. Ferdinand and R. Wilhelm. Fast and Efficient Cache Behavior Prediction for Real-Time Systems. *Real-Time System*, XVII:131–181, 1999.

[13] Freescale Semiconductors. P4 series. P4080 multicore processor (white paper), 2008. http://www.freescale.com/files/netcomm/doc/fact_sheet/QorIQ_P4080.pdf.

[14] 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.

[15] Damien Hardy and Isabelle Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the 2008 Real-Time Systems Symposium*, RTSS '08, 2008.

[16] L. Kosmidis, J. Abella, E. Quinones, and F.J. Cazorla. A cache design for probabilistically analysable real-time systems. In *DATE*, 2013.

[17] L. Kosmidis, C. Curtsinger, E. Quinones, J. Abella, E. Berger, and F.J. Cazorla. Probabilistic timing analysis on conventional cache designs. In *DATE*, 2013.

[18] L. Kosmidis, T. Vardanega, J. Abella, E. Quinones, and F.J. Cazorla. Applying measurement-based probabilistic timing analysis to buffer resources. In *WCET workshop*, 2013.

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

[20] B. Lesage, D. Hardy, and I. Puaut. WCET analysis of multi-level set-associative data caches. *WCET Workshop*, 2009.

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

[22] F. Mueller. Predicting instruction cache behavior. *Language, Compilers and Tools for Real-Time Systems*, 1994.

[23] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems - Special issue on worst-case execution-time analysis archive*, 2000.

[24] M. Paolieri, E. Quinones, F.J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009.

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

[26] F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quinones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, T. Vardanega, and F.J. Cazorla. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *SIES*, 2013.

[27] Wilhelm R. 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.

ANNEX I. ACTUAL HIT/MISS PROBABILITIES IN TIME-RANDOMISED CACHES

As indicated in Section III, hit/miss probabilities provided in this paper are an approximation to the actual ones. To illustrate how actual probabilities can be derived, we use an example where a sequence of accesses $<A_1, B_1, A_2, B_2>$ access cache lines A and B in a fully-associative cache with 4 cache lines. Figure 5 depicts the sequence of accesses, the different events that can occur and their probabilities. At the bottom of the figure probabilities for each sequence of events are provided.

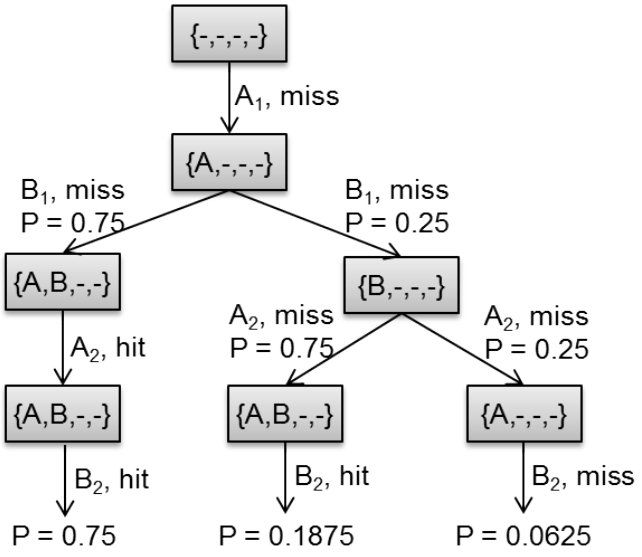

Fig. 5. Access outcome and cache state for the access sequence $<A_1, B_1, A_2, B_2>$.

A_1 always misses in cache and A is in cache after this access. Then, B_1 misses in cache, but two different cache states may be reached: with a probability of 0.75 B replaces any of the empty lines and with a probability of 0.25 (1 out of 4) B replaces A so that B is the only valid line in cache. A_2 accesses cache and hits if the cache contents were $\{A, B, -, -\}$, which occurs in one of the two different sequences of events at this stage. If A_2 hits (leftmost path in the graph), cache state remains the same ($\{A, B, -, -\}$), otherwise it may happen again that A replaces an empty line (0.75 probability) or B (0.25 probability). Finally, B_2 hits in cache in two of the three different sequences of outcomes.

Overall, if we compute the actual probabilities from the graph, we obtain the probability vectors in Table IV. As shown, the probability vector for B_2 differs across the exact computation and the approximation provided by Equation 1. This is so because the probabilities of hit and miss across different sequences of events differ. However, although those probabilities are not independent, whether a hit or a miss occurs depends solely on random events, so properties needed by MBPTA [8] (i.i.d. end-to-end program execution times) are fulfilled.

Note that, as opposed to SPTA [7], MBPTA does not

TABLE IV. PROBABILITY VECTORS FOR THE ACCESSES IN THE SEQUENCE $<A_1, B_1, A_2, B_2>$ FOR A FULLY-ASSOCIATIVE 4-ENTRY

Access	Prob. vector real $\{p_{hit}, p_{miss}\}$	Prob. vector Equation 1 $\{p_{hit}, p_{miss}\}$
A_1	$\{0.0, 1.0\}$	$\{0.0, 1.0\}$
B_1	$\{0.0, 1.0\}$	$\{0.0, 1.0\}$
A_2	$\{0.75, 0.25\}$	$\{0.75, 0.25\}$
B_2	$\{0.9375, 0.0625\}$	$\{0.9306, 0.0694\}$

TABLE V. PROBABILITIES OF EXPERIENCING 0, 1 AND 2 HITS IN THE SEQUENCE $<A_1, B_1, A_2, B_2>$.

	Real	Convolution
P(0 hits)	0.0625	0.0156
P(1 hit)	0.1875	0.2813
P(2 hits)	0.75	0.7031

need to determine the actual probabilities. SPTA, indeed, needs actual probabilities to be upper-bounded and those probabilities must be independent so that they can be convolved to obtain the probability distribution for the whole program. Obtaining actual probabilities can be done in two main ways: (i) Performing an 'infinite' number of runs and measuring actual probabilities, or (ii) Computing the probability of each particular cache state left by the sequence of hits and misses for previous accesses, and accumulating the probabilities for those cache states where the current cache access would result in a hit/miss, as we do in our example. Unfortunately, even if exact probabilities are obtained, they cannot be convolved, so if SPTA is to be used, a way is needed to compute probability vectors that upper-bound those under any sequence of events for each access, i.e. by making sure that the miss probability used is equal or higher than the miss probability under any sequence of events [7].

For the sake of illustration, we provide in Table V the probabilities of experiencing 0, 1 and 2 hits when executing the sequence $<A_1, B_1, A_2, B_2>$ (i) directly from the probability graph in Figure 5 and (ii) by convolving the probability vectors in Table IV. For instance, the actual (real) probability of having exactly one hit is 0.1875, which occurs when A_2 misses and B_2 hits. When applying convolutions, such probability of having exactly one hit is obtained as the addition of the probabilities of (1) A_2 missing and B_2 hitting ($0.25 \cdot 0.9375 = 0.2344$) and (2) A_2 hitting and B_2 missing ($0.75 \cdot 0.0625 = 0.0469$), which indeed cannot occur. As expected, probabilities obtained with convolutions neither match nor upper-bound real ones (e.g., SPTA would underestimate the probability of having 0 hits, which is the one leading to the highest execution time).

Overall, deriving actual probabilities would require a more complex formulation than the one we have used in this paper to derive approximations. The purpose of deriving the probability approximations is proving that hit and miss events occur with a given probability, which is a sufficient condition for enabling the application of MBPTA in multilevel caches.

ANNEX II. EXCLUSIVE CACHES

In this section we briefly discuss some considerations for exclusive caches. In exclusive caches, contents cannot be replicated across multiple caches. Therefore, on a DL1 and UL2 miss, the new line is fetched from memory to DL1, the line evicted from DL1 is moved to UL2, and the line evicted from UL2 is invalidated (if clean) or written back to memory (if dirty). Analogously, on a DL1 miss and UL2 hit, the line from UL2 is moved to DL1. One line from DL1 is evicted and placed in UL2, which can also produce a cascade eviction since placement functions in DL1 and UL2 are independent and so, although both lines (the one fetched and the one evicted) are placed into the same set in DL1, they are very unlikely to be placed in the same set in UL2. In general, exclusive caches are not common because of the number of cache line transfers on a DL1 miss.

Exclusive caches are ill-advised in combination with WT policy, as it is the case of the DL1 considered in this work. This is so because on a DL1 store hit, the write operation is sent to UL2, where it is guaranteed to miss due to the exclusivity constraint. This would enforce evicting such particular cache line from DL1 to put it in UL2 (potentially causing a UL2 eviction) or sending the write operation straight to memory, thus jeopardising performance and power due to the increased number of memory accesses.

The events that may potentially happen in the different caches have an associated probability ($P_{UL2,hit/miss}(@)$, $P_{IL1,hit/miss}(@)$, $P_{DL1,hit/miss}(@)$ and $P_{UL2,dirty}(@)$). The values of some of those probabilities change with respect to the non-inclusive and inclusive cases as detailed next.

C.1. $P_{DL1,miss}(@)$. It is not affected by the inclusivity control, so it remains as described for the non-inclusive case, since UL2 cannot produce any eviction in DL1.

C.2. $P_{UL2,miss}(@)$. There is a probability that an access $@_A$ to the UL2, and so its $P_{UL2,miss}(@_A)$, is affected by the exclusivity policy. On a DL1 miss, data can be found in any UL2 line. Since DL1 and UL2 are exclusive, a miss in UL2 occurs if and only if data are not in the $W_{UL2} + W_{DL1}$ lines of DL1 and UL2 together, being its miss probability approximation:

$$P_{UL2,miss}(@_A) = 1 - \left(\frac{W_{UL2} + W_{DL1} - 1}{W_{UL2} + W_{DL1}}\right)^{\sum_{i=1}^{k} P_{UL2,miss}(L2acc_i)}$$

where the exponent is the miss probability of the accesses between $@_A$ and its last access.

C.3. $P_{UL2,dirty}$. It is not affected by the inclusivity control, so it remains as described for the other cases.