Design and Integration of Hierarchical-Placement Multi-level Caches for Real-Time Systems

Pedro Benedicte[*†], Carles Hernandez[*], Jaume Abella[*] and Francisco J. Cazorla[*‡]
[*] Barcelona Supercomputing Center [†] Universitat Politècnica de Catalunya [‡] IIIA-CSIC

***Abstract*—Enabling timing analysis in the presence of caches has been pursued by the real-time embedded systems (RTES) community for years due to cache's huge potential to reduce software's worst-case execution time (WCET). However, caches heavily complicate timing analysis due to hard-to-predict access patterns, with few works dealing with time analyzability of multi-level cache hierarchies. For measurement-based timing analysis (MBTA) techniques – widely used in domains such as avionics, automotive, and rail – we propose several cache hierarchies amenable to MBTA. We focus on a probabilistic variant of MBTA (or MBPTA) that requires caches with time-randomized behavior whose execution time variability can be captured in the measurements taken during system's test runs. For this type of caches, we explore and propose different multi-level cache setups. From those, we choose a cost-effective cache hierarchy that we implement and integrate in a 4-core LEON3 RTL processor model and prototype in a FPGA. Our results show that our proposed setup implemented in RTL results in better (reduced) WCET estimates with similar implementation cost and no impact on average performance w.r.t. other MBPTA-amenable setups.**

I. Introduction

The complexity of on-board computing systems in domains such as aerospace, automotive, and rail has steadily increased in recent years due to the innovation in their electronics and software components. Taking the automotive domain as an example, on-board software in cars already comprises more than 100 millions lines of code [9], with its performance requirements expected to rise by two orders of magnitude [5] by 2024. It is widely accepted that timely executing those software functionalities will rely on processors comprising high-performance features. And, undoubtedly, caches are one of the resources with highest impact on performance.

In the real-time domain, the challenge of using complex hardware lies on providing increased performance guarantees (i.e. reduced WCET estimates) – and not just increased average performance as needed in the mainstream market – with hardware features like caches complicating deriving performance guarantees [20]. Time composability, a desired property for WCET estimates, allows deriving those estimates in early design stages with assurance that they remain valid as different software components, which are developed independently, are integrated. This is a fundamental property in increasingly-complex multi-provider software projects in integrated systems like *Integrated Modular Avionics* (IMA) [1] in the avionics domain and AUTOSAR in the automotive domain [6].

In this paper we focus on the most extended timing analysis practice, measurement-based timing analysis (MBTA) [25]. MBTA relies on collecting task's execution time measurements on the target hardware during the system analysis (design) phase under different stressing conditions with guarantees that those conditions capture the worst scenarios that can arise during operation. MBTA can be used for the timing analysis of the highest-criticality tasks, as it has been shown for avionics software [18]. This requires that the user masters all the sources of execution time variability (jitter) in the platform and provides evidence that their operation-time impact on software execution time has been captured in the test campaign carried out during the analysis phase.

While this level of control can be reached with simple hardware, it is hard to maintain in the presence of caches. This occurs because with modulo placement, program's data/code addresses in memory (i.e. the memory mapping) determine the cache sets in which they are mapped (i.e. the cache layout). Despite the system engineer performs many runs during the test campaign, it is hard for him/her to provide evidence of coverage (representativeness) of operation-time cache layouts. Random placement caches make any given address be randomly mapped to different cache sets every time a random seed is changed, effectively breaking the dependence between memory mapping and cache layout. This allows: (i) providing probabilistic arguments on the coverage of those cache layouts that result in high execution times; (ii) dealing with *cache risk patterns*, i.e. cache layouts causing high execution times [20] – one of the main stumbling blocks for the ubiquitous adoption of caches in RTES; and (iii) providing *time composable* WCET estimates in early design stages that remain valid upon integration of other software components.

The current landscape of random caches covers hash-based random-placement caches (hRP) [16] and Random Modulo (RM) [14]. The former hashes a given random seed and the address so that it can be mapped to any set. RM improves the spatial-locality of hRP by generating a *permutation* of the addresses in the same memory page, so that addresses in the same page can be mapped to any random set, though they cannot conflict with each other in cache. However, despite these efforts, it is not yet well understood how to design efficient multi-level time-randomized cache hierarchies and how different randomization policies in each level impact average performance and WCET. Our contributions are as follows:
① We perform a design space exploration of multi-level random cache designs in a cycle-accurate simulator. We explore monolithic designs by applying existing L1 placement policies to both L1 and L2. We show that these policies are not designed for L2 caches and have performance (average/worst-case) or time composability issues.
② To tackle the observed deficiencies, we introduce, for the first time, hierarchical placement designs that solve L2 related issues while still being MB**P**TA (Probabilistic) compliant. Our results show that the proposed hierarchical designs have no negative impact on average performance, improve worst-case results with respect to the monolithic designs, and favor time composability.
③ We implement and integrate the most cost-effective cache hierarchy in a 4-core RTL processor model prototyped in a FPGA. Our results show that it has almost the same performance as modulo placement, provides tight WCET estimates, and enables MBPTA time-analyzability.

II. Background

A. Basics on Timing Analysis

Real-time tasks are assigned a 'criticality' level as part of the safety design process. For instance, in automotive, software elements – and more specifically the safety requirements they are assigned – are attached an Automotive Safety Integrity Level (ASIL). A common

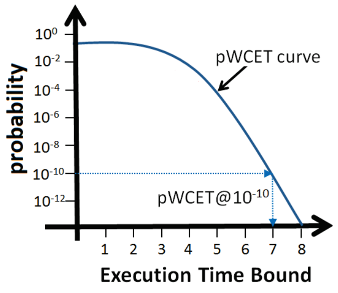


Fig. 1: pWCET distribution

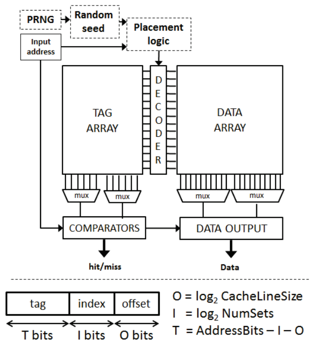


Fig. 2: Random placement cache

misconception is assuming that critical real-time tasks (even most-critical ones) cannot miss a deadline without this causing a system-perceived failure. In reality, safety mechanisms are put in place to reduce the risk to sufficiently low levels of hardware/software faults causing system failures (more specifically to the violation of the safety goals of the system element they belong to). Hence, in short, all hardware/software elements can fail and proper measures are put in place – obviously for high ASIL more stringent measures are used – to detect and correct faults if they occur. For instance, highest-criticality (ASIL-D) random hardware residual faults are considered acceptable if the failure rate is below 10^{-8} per hour with a diagnosis coverage of at least 99%. Following this concept, MBPTA [3] delivers a probabilistic WCET (pWCET) distribution as shown in Figure 1, where for instance the probability that the particular software unit takes longer than 7ms in one run is at most 10^{-10}.

The timing of tasks under analysis needs to be assessed against allocated time budgets early in the design process to take corrective actions with limited effort and time costs [21]. Otherwise, not only regression tests are more complex to design but also, as software gets integrated, finding an overrun late can cause costly system redesigns and even delay product's time to market. Hence, it is desirable for system engineers to have *time composable* timing bounds that are estimated in early design stages and remain valid upon integration of other software components, thus enabling *incremental software integration*. This is a fundamental property in increasingly-complex multi-provider software projects like IMA [1] in avionics or AUTOSAR [6] in automotive domains.

With caches, the relative position of program's memory objects may change across software integrations leading to different cache layouts with arbitrary impact on execution time. This breaks time composability and shifts timing analysis and verification to the latest design stages (when the binary is fixed) with increased risk of failing to meet execution time bounds. In this context, random placement policies together with MBPTA have been shown to enable incremental verification in the presence of cache memories [8]. In particular, random placement policies break the dependence of cache placement on the actual memory addresses, i.e. in each run software experiences random placement of memory objects in cache. As a result, the actual memory addresses are irrelevant for cache placement and the space of potential cache placements is randomly sampled in each run. Since the probability distribution for cache placements observed at analysis matches that during operation, impact of cache placement can be analyzed with MBPTA to produce probabilistic bounds on its impact on execution time. In fact, MBPTA is capable of considering different sources of random variation (e.g. cache placement for multiple caches, random arbitration in buses) simultaneously. However, while random caches remove WCET estimate dependence on memory location of objects, thus relieving the user from controlling memory placement, it has not been explored how the different random placement policies need to be combined into multi-level cache hierarchies. In particular the desired properties are:

1) **WCET reduction** as the main metric to optimize.
2) **Reduced impact on average performance** due to the importance of this metric for mixed-critical scenarios executing tasks with different criticality levels.
3) **Increase time composability** to favor incremental software development as described above.
4) **MBPTA compliance** to reduce the cost of changing existing timing analysis tools.

Next, we present several multi-level cache designs and assess them against these metrics.

B. Single-level Random Cache Implementations

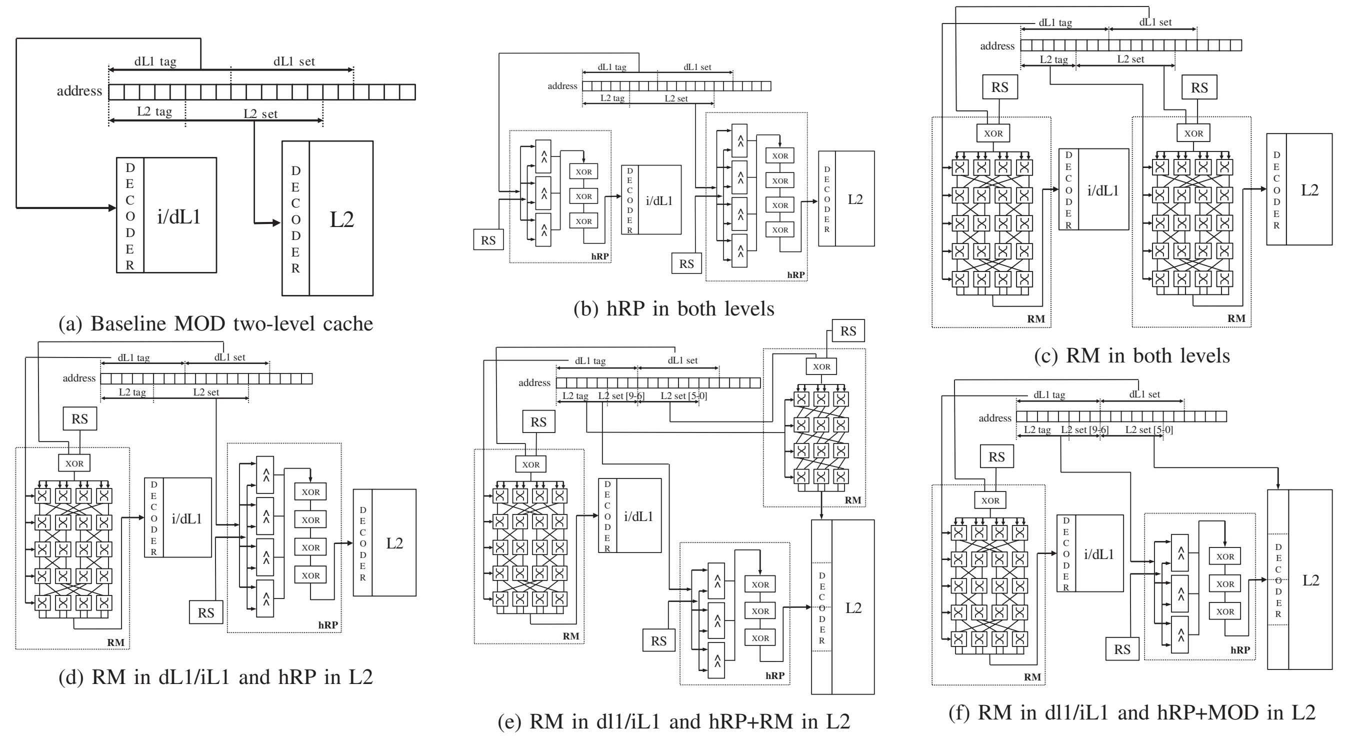
Figure 2 shows a block diagram of a cache and how randomized placement would fit in its overall design. As shown, for the generation of the index – used to feed standard modulo placement – specific logic 'combines' the accessed address and a random number from a pseudo-random number generator (PRNG). State-of-the-art PRNGs deliver value series long enough to exclude repetition in short periods, thereby preventing any potential correlation of events [4].

hRP placement [16] uses a parametric hash function whose input includes the memory address to be accessed and a random seed. It produces the (random) set where the address is placed with that random seed. The hash function combines address bits (factoring out those determining the offset within the cache line) and the random seed. In particular, it uses a set of rotator blocks and XOR gates so that the set chosen for any given address is random. Thus, whether two addresses are placed or not in the same set is a random event. Upon change of the random seed, addresses are randomly and independently mapped into sets. hRP provides homogeneous distribution of addresses across sets, so the probability of each address to be placed in each set is $1/S$, where S is the number of sets.

hRP is used by flushing cache contents and setting a (new) random seed, usually at task execution boundaries[1]. This leads to a random placement of addresses, that holds during the whole execution, so addresses placed in the same set compete for the set space during the whole run, whereas addresses placed in different sets have no conflict in that run. hRP imposes that cache line alignment during analysis and operation is preserved. Thus, objects can be shifted in memory freely at the granularity of cache line size upon integration without impacting (random) placement.

RM placement [14]. Unlike hRP, RM placement breaks the dependence between memory location and cache placement, preserving the advantages in terms of spatial locality as modulo placement does. In particular, RM prevents conflicts between cache lines with identical tag bits, which we refer to as a cache segment. This is achieved by using a random seed, hashed with tag bits (T bits), that determines a random permutation of I (index) bits. Such random permutation changes across addresses by varying T bits and across random seeds. Thus, addresses are placed in random and independent sets across runs. However, two addresses with identical T bits and different I bits are necessarily placed in different cache sets given a fixed random seed. Thus, nearby addresses (those sharing the same T bits) cannot be placed in the same set.

[1]Tasks sharing a cache memory require coordination for seed update and cache flushing. This can be achieved by changing seeds at execution time partition boundaries as described in the context of IMA and proven in [30].


(a) Baseline MOD two-level cache

(b) hRP in both levels

(c) RM in both levels

(d) RM in dL1/iL1 and hRP in L2

(e) RM in dl1/iL1 and hRP+RM in L2

(f) RM in dl1/iL1 and hRP+MOD in L2

Fig. 3: (a) Baseline modulo cache. (b)(c) Monolithic placements. (d)(e)(f) Hierarchical placements. RS stands for Random Seed

RM poses constraints on integration: addresses in a cache segment (same T bits) during analysis must belong to the same cache segment upon integration. Hence, addresses may be shifted at the granularity of 2^{I+O} bytes (the size of a cache segment), which is practically achieved by making cache way size (W_{size}) match that granularity ($W_{size} = 2^{I+O}$), which is further made match memory page size. Thus, at software level objects are aligned at page boundaries and cache ways need to match that size (or be divisors of that size).

III. Multi-level Random Cache Approaches

Time-randomized cache placement policies have been evaluated mainly for single-level cache hierarchies. An exception to this is hRP that has been shown to keep its MBPTA-compliance properties for multi-level caches [17]. However, hRP is the existing random placement policy with lowest average and guaranteed performance. Hence, there is significant room for improvement in multi-level random cache design. In this line, this section presents several approaches that use, individually or in a smartly-combined way, different random placement policies to provide higher-performance MBPTA-compliant multi-level cache designs. For clarity, we use the L2 placement policy as the identifier for the multi-level configuration. See Table I for the list of configurations.

A. L2 Monolithic Placement

MOD setup is the reference setup against which we compare other randomized setups in terms of average performance. This setup uses modulo placement (MOD) – deployed in many multi-level cache designs as placement policy for all cache levels – see Figure 3(a). It determines cache placement based on cache index bits (I bits) and it is not amenable for MBPTA. This is mainly due to MOD deterministic behavior: although conflictive memory alignments can be infrequent, they may occur upon integration with a systematic and pathological nature, resulting in the so-feared (for measurement-based techniques) cache risk patterns.

TABLE I: L1+L2 placement policies. *PL* stands for Page Level

Setup	MOD	hRP	RM	hRP2	hRP+MOD	hRP+RM
L1	MOD	hRP	RM	RM	RM	RM
L2	MOD	hRP	RM	hRP	hRP *PL* + MOD	hRP *PL* + RM

hRP setup. In this setup hRP placement is used for first level data and instruction caches, respectively referred to as dL1 and iL1, and the second level cache (L2), see Figure 3(b). This setup was already considered in [17] given that hRP was the first random placement policy proposed compatible with MBPTA. This design only imposes preserving cache line alignment between analysis and operation phases. However, hRP allows all cache lines to be randomly placed completely independently. Therefore, few cache lines may be placed in the same cache set in L1 caches (either dL1 or iL1) with a relevant probability for pWCET estimation, and also in L2 cache. Thus, while those bad placements occur with relatively low probability, having low impact on average performance, they may lead to large impact in pWCET estimates to account for very bad placements that can occur even with very small working sets.

RM setup. RM placement implements a Benes network (Figure 3(c)) that produces a random permutation of the index bits – XORed with the random seed – being the permutation controlled by the T tag bits. With *RM*, cache-segment alignment must be maintained between analysis and operation: all addresses fitting in a cache segment in the experiments carried out at analysis, must remain in a segment during operation. As explained before, the real-time operating system (RTOS) can easily achieve this goal by matching memory page size with cache segment size, or making page size be a multiplier of cache segment size. In fact, this assumption has already been shown compatible with complex avionics case studies [30].

RM can be soundly used for first level caches whose way size (i.e. cache segment size) is typically equal or smaller than the page size. When the way size is larger than the page size, usually the case for L2 whose size is easily above 128KB-256KB, then *RM* can be used if the RTOS preserves page alignment at that granularity. For instance, if the cache way size is k times larger than the page size, the RTOS should maintain the alignment of pages at $k \times page\_size$ bytes granularity to soundly apply MBPTA. However, dealing with this constraint is unaffordable in practice due to memory fragmentation (the subsequent memory space waste). Further, it also adds complexity to the RTOS. Hence, *RM* can be used in L1 caches, and *hRP* in the L2 instead as presented next.

Summary. Neither MOD nor RM in L2 are MBPTA compliant and defeat achieving time composability. We keep the former for average performance comparison purposes, while we discard the latter. hRP is MBPTA compliant and maintains time composability, and hence, we use it as reference randomization policy for L1 and L2.

B. L2 Hierarchical Placement

Next we propose hierarchical designs based on multiple policies to get the best of each policy.

hRP2 setup. This setup combines the advantages of RM in L1 caches to preserve spatial locality, and the flexibility of hRP in L2, avoiding posing undue constraints on memory object alignment, see Figure 3(d). Thus, qualitatively, this setup is far more convenient than those presented so far by smartly combining in different cache levels appropriate random placement policies. However, hRP has been shown to have low but non-negligible hardware cost, due to the expensive barrel shifters followed by a tree of XOR gates, whose number and size – and so impact on area – grows significantly with the number of address bits to handle. Therefore, we examine other hybrid solutions for L2 caches.

hRP+RM setup. This setup – that uses RM in L1 caches – performs hRP at page size in L2 and RM within page size, as seen in Figure 3(e). We build on the observation that, as L2 cache ways are conceptually split into cache segments, hRP can be used to randomly select the cache segment where an address is placed and RM to select the set within the segment. This setup requires preserving page alignment between analysis and operation phases. However, such constraint is already imposed by L1 caches, so constraints remain the same as for any other setup using RM in L1 caches.

This hierarchical design has a positive impact in the implementation cost of the L2. First, hRP only operates on tag (T) bits instead of on $T + I$ bits. RM, instead, randomizes placement within page boundaries thus operating on the remaining I bits. However, RM is much cheaper than hRP in terms of area. This is further detailed later in Section V and in Table III. While the impact on the critical path is roughly null, hRP logic becomes the critical path for large caches (larger than L2 caches evaluated in this work). Hence, the hybrid solution would also mitigate delay issues in those cases.

As this design uses RM at the page level, the low performance of hRP is mitigated. The other side of the coin is that it can be the case that two pages are randomly mapped to the same cache segment. The fact that we use RM inside L2 segments reduces the likelihood that pages (segments) evict each other's lines systematically.

hRP+MOD setup. While the previous setup reduces the hardware overhead of L2 compared to those with hRP in L2, the hardware for L2 cache placement must still accommodate hRP across cache segments and RM within segments. This overhead can be further reduced by removing RM from L2 cache segments (and sticking to MOD), see Figure 3(f).

This approach reduces hardware complexity, but the degree of randomization achieved in L2 also decreases. While it has been shown that higher degrees of randomization lead to less abrupt execution time variations (and thus to lower pWCET estimates) [29], the fact that addresses go through RM placement and random replacement (RR) in L1 caches, hRP across cache segments in L2, and RR in L2 (if more than 1 way is used per core), already brings high degrees of randomization. Hence, we regard this setup as a good trade-off between randomization achieved and hardware cost. In particular, this setup decreases transistor count slightly and may reduce cache placement latency for caches with large number of L2 cache sets due to the decreased logic depth.

IV. Evaluation on a Simulator

We build on SoCLib [28] to model a pipelined in-order processor resembling the LEON3+ design in [13]. We model 16KB 4-way dL1 and iL1 caches per core, with 16 and 32B/line respectively, and a shared 128KB 4-way L2 cache that is partitioned so that each core receives an independent L2 cache way of 32KB. The dL1 is write-through no write-allocate, so store operations are always forwarded to the L2 cache and do not fetch data to dL1 on a miss. The L2 is write-back write-allocate, so on a store miss, the cache line is fetched into L2 – and modified. Caches are non-inclusive, so no control is exercised on whether cache lines must or must not reside in any particular cache memory. Bus arbitration implements random permutations [15]. Random permutations are also used to arbitrate memory requests (L2 cache misses). Memory latency is 16 cycles to serve a request and 27 cycles until the next request is served [23].

We evaluate a large subset of the EEMBC automotive [24] suite comprising common critical real-time applications in automotive systems and MediaBench [19] comprising embedded applications such as multimedia and communications[2].

Results. Our designs aim at (a) maintain MBPTA compliance; (b) reduce pWCET estimates w.r.t. simpler MBPTA-compliant designs; (c) obtain comparable performance to non-randomized (and hence non MBPTA-compliant) modulo+LRU based multilevel cache hierarchies; and (d) preserve time composability.

To assess MBPTA compliance, we have followed the approach proposed in [17] checking that cache events preserve its random/probabilistic nature. Our results – not shown for space constraints – show that an address can be randomly mapped to any dL1 (iL1) and L2 set. Further, independence and identical distribution tests on execution times are passed [3].

To derive pWCET estimates, and obtain solid average performance results, we carry out 500 runs for each benchmark-setup pair. pWCET estimates are shown for an exceedance threshold of 10^{-12} per run, since they are enough for the highest criticality software [30].

MediaBench. In Table II columns 2-3 show the pWCET estimates obtained with each placement policy normalized to the monolithic setup *hRP*. We observe that hierarchical setups consistently reduce the pWCET estimates of *hRP*, by 28% and 34% for hRP+MOD and hRP+RM respectively.

In terms of average performance, columns 4-6 show that the three hierarchical setups obtain comparable results to those of the deterministic approach (MOD+LRU), only up to 2% worse. This is so because bad placements occur seldom even for the worst setups, so average results hide outliers. We repeated the same analysis for executions resulting in the *highest 5% miss counts*, as they shape the tail of the execution time distribution, and hence WCET [3]. Our results show that hRP achieves worst results than hierarchical approaches: with hRP, by allowing each cache line to be placed randomly and independently in L2, any pair of cache lines can, in

[2]We excluded those benchmarks that we could not make work in our platform: `aifirf`, `aiifft`, `idctrn` (EEMBC); `g721`, `mpeg` (MediaBench).

TABLE II: MediaBench results

	pWCET vs hRP		avg perf vs MOD				pWCET vs hRP		avg perf vs MOD		
	hRP MOD	hRP RM	hRP	hRP MOD	hRP RM		hRP MOD	hRP RM	hRP	hRP MOD	hRP RM
ad.d	0.08	0.05	1.01	1.00	1.00	m.m	0.99	0.89	1.01	1.01	1.02
ad.e	0.25	0.18	1.01	1.01	1.00	m.o	0.48	0.49	1.00	1.00	1.00
ep.d	0.89	0.90	0.99	1.00	0.99	m.t	0.87	0.83	1.01	1.01	1.01
ep.e	0.90	0.65	1.00	1.00	1.00	pe.d	0.71	0.72	1.01	1.00	1.00
gs.d	0.94	1.00	1.00	1.00	1.00	pe.e	0.65	0.64	1.01	1.00	1.00
gs.e	0.75	0.75	1.01	1.01	1.01	pg.d	0.73	0.75	1.01	1.00	1.00
jp.d	0.84	0.77	1.02	1.02	1.02	pg.e	0.69	0.16	1.02	1.00	1.01
jp.e	0.91	1.10	1.01	1.02	1.01	rast	0.87	0.76	1.00	1.00	1.00

the worst cases, be placed in the same set and produce high miss counts, and hence execution times. This explains why hierarchical placements improve hRP for pWCET (columns 2-3).

EEMBC Automotive trends are similar to MediaBench. We observed similar average execution time for all placements, and hRP being the worst policy in terms of worst-case execution time. Results are omitted since they bring no further insights.

Summary. hRP+RM and hRP+MOD avoid some systematic effects of hRP, which reduces their L2 miss rate (and execution time) for pathological cases w.r.t. hRP. Also, it cannot be claimed whether hRP+RM or hRP+MOD is superior, since our results show that conclusions change across different benchmarks. Moreover, although the cost of implementing hRP+RM is only slightly higher than that of implementing hRP+MOD in L2, hRP+MOD can be regarded as an effective setup. Also, this combination is interesting because it synergistically combines 3 different placement policies: RM in L1 caches, hRP across L2 cache segments, and modulo inside L2 cache segments. For these reasons, we implemented this setup in RTL.

V. Design Validation:RTL implementation

To validate simulation results we implemented hRP+MOD setup in a LEON3-based RTL prototype and synthesized it in a Terasic Stratix-IV board able to operate at 100Mhz.

Processor Model. The architecture of the baseline processor is analogous to the one simulated with per-core iL1 and dL1, and a shared L2. Cores are connected to the L2 through an on-chip bus implementing a random arbitration policy [27] and a memory controller is placed after the L2 to forward requests to the off-chip DDR2 RAM memory. L1 caches are 16KB 4-way and the L2 is 128KB 4-way. To ease timing analyzability, the shared L2 cache supports partitioning, where one way is assigned to each core. The integration of random placement requires including one placement function per each of the L1 private caches so in total 8 placement functions are required for the 4 cores. The L2 cache module requires using a different seed per core to allow cores having independent probabilistic timing behavior. Note that despite the L2 is partitioned, each core may read shared data from the other ways (while only the master core is entitled to evict data) so 4 independent placement functions are required for the L2, each one using the seed of the corresponding core, hence the importance of mitigating area overhead in L2 placement.

Area overhead. Table III shows the area and delay overhead introduced by the different random placement functions considered in this study. As shown, RM requires significantly lower area than hRP since it uses a permutation network consisting of few pass transistors per index bit. The actual permutation carried out is driven by XORing address bits and seed random bits. Instead, hRP requires combining the seed random bits and the address by means of barrel shifters and several levels of XOR gates [16]. The hierarchical random placement implementation reduces area overheads of hRP by roughly 50%. This, coupled with the good performance results it provides, confirms the hierarchical approach as the most suitable option to implement random placement in L2 caches.

TABLE III: Hardware cost and delay

	dL1		L2		
	hRP	RM	hRP	hRP+MOD	hRP+RM
Trans. Count	49488	240	49440	24360	24600
Delay (ns)	0.65	0.26	0.52	0.52	0.52

In terms of overall hardware occupancy, the baseline design occupied 70% of the resources in the FPGA, whereas the design including the random placement in all L1 caches and L2 occupies less than 72%, thus showing that all cache modifications required to achieve MBPTA-compliance incur very low overheads.

Critical Path. RM is faster than hRP-based approaches since the latency of the latter is mainly determined by the depth of the XOR gates tree employed to combine address and random bits. In the case of RM, the XOR gates tree must produce the bits required for configuring the permutation network while for hRP, XOR gates are combined to produce the randomized index itself. For the L1 we see that RM outperforms hRP being able to reduce its latency by $2.46\times$. For the L2, hRP delay is lower than for the L1 since fewer XOR gates are required to produce a wider cache index. However, hierarchical implementations do not necessarily reduce the delay since, despite fewer bits are combined to produce the random index, since this index has fewer bits, more XOR gates are required to produce the output. While in our implementation of the hierarchical approach the two effects compensate each other, thus making latency remain the same, different implementations may provide slightly different results.

Overall, the hierarchical implementation (hRP+MOD/RM) decreases area overheads and reduces the number of critical paths ($\approx 3X$), which in this case correspond to the index bits, w.r.t. the hRP implementation. The latter significantly mitigates the impact that process variations have on the maximum achievable frequency [7]. In particular, for hRP+MOD, the L1 access latency slightly increases by two XOR gates w.r.t. modulo placement. For the L2, hRP+MOD causes a larger impact on critical path due to the higher complexity of its design (XOR gates and barrel shifters). Still, this impact was not enough to decrease the maximum operating frequency.

Performance Validation. To validate performance results of the hRP+MOD setup, we run the EEMBC automotive benchmark suite in the FPGA prototype. Our platform does not implement a floating-point unit so we excluded those benchmarks using FP operations. Also, MediaBench requires some I/O RTOS support that is not yet available for this particular configuration, so we did not include them. We made 500 runs of each benchmark and averaged hit ratios, and compared the implemented hRP+MOD and the default MOD against results in the simulator. Results (not shown for space constraints) reveal that the FPGA implementation of hRP+MOD shows almost the same behavior observed in the simulation evaluation. Hit rates are quite high for all EEMBC, specially for the L1 but also for the L2, proving the effectiveness of these placement policies.

Average and Worst-Case Performance Results. The first bar in each pair in Figure 4 shows that hRP+MOD achieves very similar average performance to that of MOD: 1% worse on average and up to 3%, making hRP+MOD very competitive in terms of average performance. For WCET estimation, we build on current industrial practice for WCET analysis on real boards that takes as WCET estimate a margin (e.g. 20%) over the high watermark (HWM) execution time [30]. The second bar in each pair in Figure 4

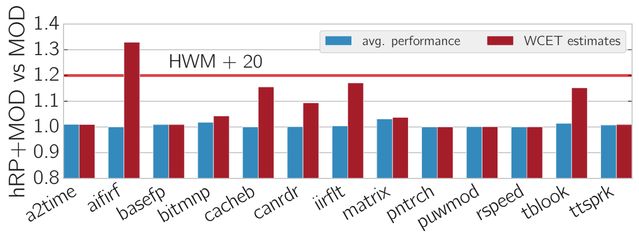


Fig. 4: Average performance results (left bar) and pWCET results (right bar) of hRP+MOD w.r.t. MOD

shows the pWCET estimate obtained for hRP+MOD w.r.t. to the highest execution time observed for MOD (i.e. the HWM). For all benchmarks we observe that the pWCET estimate is above the HWM (as expected) and for all benchmarks but one the pWCET estimate with hRP+MOD is below HWM+20% obtained for MOD. Hence, hRP+MOD helps reducing WCET estimates w.r.t. current practice while increasing the confidence on estimates w.r.t. just increasing the HWM by a fudge factor of 20%. On average, pWCET estimates are just 8% over the HWM (12 percentage points below HWM+20%).

VI. Related Work on Random Caches

In [11] authors propose a pseudo-random hash function to distribute the data across sets and thus, make the cache performance less sensitive to different placements compared to conventional modulo placement. Topham [12] also explores different pseudo-random hashing functions to reduce conflict misses. With skewed associative caches [10], each way uses a distinct hash function for randomized placement across banks, which reduces conflict misses for programs that process large matrices. A commonality of these solutions is that placement uses only the address of the access. As a result, for a given memory layout a single placement is produced across all runs of the program. This poses the same limitations for MBPTA as conventional deterministic architectures based on modulo placement: time composability is lost since performance changes arbitrarily if memory addresses change upon integration.

RM caches [14] improve performance over random hRP while maintaining MBPTA compliance. Random caches (RM and hRP) in the context of MBPTA have been shown to require to control the number of runs to carry out [26], [22] to reach statistically relevant results. This has been shown achievable [22], [2]. RM caches perform better on the worst case scenarios than caches with hRP because they avoid some pathological cases by construction, such as conflicts across lines in the same page. Enhanced RM is an improvement over RM [29] that homogenizes the distribution of addresses across sets. We implemented this policy both in the simulator and the FPGA, but omitted the results since the difference w.r.t. RM was marginal.

VII. Conclusions

While cache memories (in particular multi-level cache hierarchies) offer benefits for RTES, they challenge timing analysis. Some studies show that MBPTA combined with time-randomized caches facilitate factoring in the impact of caches in WCET estimates. However, those studies mostly focus on single-level cache hierarchies. In this paper, we propose several multi-level configurations and implement them on a simulator. These configurations include both, monolithic and new hierarchical solutions. Finally, we implement the most cost-effective hierarchical configuration in an FPGA, and compare it against a conventional deterministic cache. Our results show that this solution results in negligible average performance degradation and improved (reduced) WCET estimates, while preserving time composability.

VIII. Acknowledgment

This work has been partially funded by the Spanish Ministry of Economy under TIN2015-65316-P, Carles Hernandez's TIN2014-60404-JIN (jointly with FEDER funds) and Jaume Abella's Ramon y Cajal grant RYC-2013-14717. Pedro Benedicte is funded by the Spanish Ministry of Education FPU15/01394.

References

[1] *ARINC Specification 653: Avionics Application Software Standard Standard Interface, Part 1 and 4*, 2012.
[2] J. Abella et al. Heart of Gold: Making the improbable happen to extend coverage in probabilistic timing analysis. In *ECRTS*, 2014.
[3] J. Abella et al. Measurement-based worst-case execution time estimation using the coefficient of variation. *ACM Trans. Des. Autom. Electron. Syst.*, 22(4):72:1–72:29, June 2017.
[4] I. Agirre et al. IEC-61508 SIL 3 Compliant Pseudo-Random Number Generators for Probabilistic Timing Analysis. In *DSD*, 2015.
[5] ARM. ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade, 2015. https://www.arm.com/about/newsroom/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade
[6] AUTOSAR. *Technical Overview V2.0.1*, 2006.
[7] K. Bowman et al. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of Solid-State Circuits*, 2002.
[8] F. J. Cazorla et al. PROXIMA: improving measurement-based timing analysis through randomisation and probabilistic analysis. In *DSD*, 2016.
[9] R. N. Charette. This car runs on code. In *IEEE Spectrum online*, 2009.
[10] F. Bodin et al. Skewed associativity improves program performance and enhances predictability. *IEEE Trans. on Comp.*, 1997.
[11] M. Schlansker et al. Randomization and associativity in the design of placement-insensitive caches. *HP Tech Report HPL-93-41*, 1993.
[12] N. Topham et al. Randomized cache placement for eliminating conflicts. *IEEE Trans. Comput.*, 48, February 1999.
[13] C. Hernandez et al. Towards making a LEON3 multicore compatible with probabilistic timing analysis. In *DASIA*, 2015.
[14] C. Hernandez et al. Random modulo: a new processor cache design for real-time critical systems. In *DAC*, 2016.
[15] J. Jalle et al. Bus designs for time-probabilistic multicore processors. In *DATE*, 2014.
[16] L. Kosmidis et al. A cache design for probabilistically analysable real-time systems. In *DATE*, 2013.
[17] L. Kosmidis et al. Multi-level unified caches for probabilistically time analysable real-time systems. In *RTSS*, 2013.
[18] S. Law et al. Achieving appropriate test coverage for reliable measurement-based timing analysis. In *ECRTS*, 2016.
[19] C. Lee et al. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO*, MICRO 30, 1997.
[20] E. Mezzetti et al. Attacking the sources of unpredictability in the instruction cache behavior. In *RTNS*, 2008.
[21] E. Mezzetti et al. A rapid cache-aware procedure positioning optimization to favor incremental development. In *19th RTAS*, 2013.
[22] E. Mezzetti et al. Randomized caches can be pretty useful to hard real-time systems. *Leibniz Transactions on Embedded Systems*, 2(1), 2015.
[23] M. Paolieri et al. *An Analyzable Memory Controller for Hard Real-Time CMPs* . Embedded System Letters (ESL), 2009.
[24] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.
[25] R. Wilhelm et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM TECS*, 7:1–53, May 2008.
[26] J. Reineke. Randomized caches considered harmful in hard real-time systems. *Leibniz Transactions on Embedded Systems*, 1(1), 2014.
[27] M. Slijepcevic et al. Design and implementation of a fair credit-based bandwidth sharing scheme for buses. In *52nd DAC*, 2017.
[28] SoCLib. -, 2003-2012. http://www.soclib.fr/trac/dev.
[29] D. Trilla et al. Resilient random modulo cache memories for probabilistically-analyzable real-time systems. In *IOLTS*, 2016.
[30] F. Wartel et al. Timing analysis of an avionics case study on complex hardware/software platforms. In *DATE*, 2015.