Time-Analysable Non-Partitioned Shared Caches for Real-Time Multicore Systems

Mladen Slijepcevic[†,‡], Leonidas Kosmidis[†,‡], Jaume Abella[†], Eduardo Quiñones[†], Francisco J. Cazorla[†,⋆]
[†]Barcelona Supercomputing Center (BSC-CNS), Barcelona (Spain)
[‡]Universitat Politècnica de Catalunya (UPC), Barcelona (Spain)
[⋆]Spanish National Research Council (IIIA-CSIC), Barcelona (Spain)

ABSTRACT

Shared caches in multicores challenge Worst-Case Execution Time (WCET) estimation due to inter-task interferences. Hardware and software cache partitioning address this issue although they complicate data sharing among tasks and the Operating System (OS) task scheduling and migration. In the context of Probabilistic Timing Analysis (PTA) time-randomised caches are used. We propose a new hardware mechanism to control inter-task interferences in shared time-randomised caches without the need of any hardware or software partitioning. Our proposed mechanism effectively bounds inter-task interferences by limiting the cache eviction frequency of each task, while providing tighter WCET estimates than cache partitioning algorithms. In a 4-core multicore processor setup our proposal improves cache partitioning by 56% in terms of guaranteed performance and 16% in terms of average performance.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design styles—*Cache memories*; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and Embedded Systems

General Terms

Design, Performance

Keywords

Real-time, WCET, Cache memories

1. INTRODUCTION

Many existing processors in the real-time domain comprise one shared, usually last-level, cache (LLC), like the ARM Cortex A9 and A15, the Freescale P4080 and the Aeroflex Gaisler NGMP. While LLC offers high potential for average performance improvement, it challenges worst-case execution time (WCET) estimation, which has made LLC to be studied in the last years by the real-time community [14, 20, 23, 24, 29].

Two main cache design ‘paradigms’ can be found in the literature: conventional timing analysis techniques (either static or measurement-based) [31] usually rely on caches that are deterministic in their temporal behaviour (e.g., caches deploying modulo placement and LRU replacement). Meanwhile, Probabilistic Timing Analysis (PTA) techniques [4, 5, 7, 8, 11] rely on caches with a *time-randomised behaviour in which hit and miss events have an associated probability for every cache access.* The main difference between time-deterministic (TD) and time-randomised (TR) caches, is that in a TD cache each memory address is mapped into a fixed cache set and way. That is, certain bits of the address (called the index) *determine* the cache set in which the address is mapped, while the way is determined by the replacement policy. In TR caches, however, each address can be mapped to any set (randomly chosen on each execution) and way (randomly chosen on every eviction), since random replacement and placement policies are used [15].

Software cache partitioning [14,20,23,29] and hardware cache partitioning [24] have been used so far to control inter-task interaction in the LLC. The former, which can only be used together with TD caches, maps data/code of each task in non-consecutive memory locations so that data/code are mapped into the desired cache sets. This solution, however, may introduce fragmentation in the use of memory and may require significant changes in the memory management. The latter, which can be used together with both TD and TR caches, relies on deploying hardware support to force each task to use a subset of the ways of set-associative caches. In this manner, tasks can be mapped to different ways preventing their interaction. Both solutions are affected by the fact that tasks may have shared pages or libraries, since cache partitioning does not allow tasks to share data on-chip. Task scheduling is also affected by both hardware and software cache partitioning. In the case of hardware partitioning, partition flushing is required to keep consistency when tasks do not always use the same partition. In the case of software partitioning, two tasks using the same partition cannot be run simultaneously.

This paper overcomes the limitations of cache partitioning by enabling the estimation of trustworthy and tight WCET estimates for systems equipped with fully-shared (non-partitioned) LLCs. The principle behind our proposal is that, while in a TD LLC interferences depend on when (time) and where (the particular cache in which) misses occur, a TR LLC cache removes any dependence on the particular addresses accessed and its assigned cache set. This makes that the LLC interferences that a task suffers only depend on how often (frequency) its co-runner tasks miss in cache and not the particular address generating the miss. As a result, *in a TR LLC controlling eviction frequency, by delaying when misses are served for each task, is enough to trustworthily upper-bound the maximum effect that such task may have on other co-running ones in the LLC.* That is, the WCET estimated for a task τ_i is trustworthy regardless of its particular co-runner tasks as long as their aggregated miss frequency – and so their eviction frequency – is below the predefined *miss frequency threshold*, MT_i, for which τ_i's WCET estimate is computed. Based on this analysis we propose a simple hardware mechanism that limits the miss frequency of tasks in each core at analysis and deployment time in a manner that probabilistic upper-bounds can be obtained for the effect in the LLC of one task on the other co-running tasks. Our approach removes cache partitioning constraints while making WCET estimates tighter. This increases the average and guaranteed performance that can be obtained. In a 4-core multicore processor setup our result show that EFL improves cache partitioning by 56% in terms of guaranteed performance and 16% in terms of average performance.

DAC ’14, June 01 - 05 2014, San Francisco, CA, USA
Copyright 2014 ACM 978-1-4503-2730-5/14/06...$15.00.
http://dx.doi.org/10.1145/2593069.2593235.

2. BACKGROUND AND RELATED WORK

2.1 Probabilistic Timing Analysis (PTA)

Probabilistic Timing Analysis (PTA) [4, 5, 7, 8, 11] provides WCET estimates with an associated exceedance probability, called probabilistic WCET (pWCET) estimates. A pWCET estimate can be exceeded leading to a timing failure. In that respect, PTA extends the notion of probability of failure to timing correctness. This is similar to the behaviour of hardware, which is regarded to fail with a given probability. PTA provides pWCET estimates for arbitrarily low probabilities, so that even if a pWCET estimate can be exceeded, it would be exceeded with an upper-bounded low probability (e.g., 10^{-15} per hour), largely below the probability of hardware failures.

We focus on measurement-based PTA (MBPTA) [7] as it is the closest PTA method to industrial practice. MBPTA derives pWCET estimates based on end-to-end execution time observations collected during the *analysis stage*. The execution times are collected on a time-randomised MBPTA-compliant architecture. MBPTA then applies Extreme Value Theory (EVT) [18], which upper-bounds the tail of the complementary cumulative distribution function (CCDF) of those observed execution times, thus providing the probability that the execution time of one run of the program during the *deployment stage* exceeds a given pWCET estimate.

With PTA, the probabilistic timing behaviour of an instruction can be represented with an Execution Time Profile (ETP). An ETP defines the discrete probability distribution function of the execution times of the instruction. It is described by the pair of vectors: $(\vec{l}, \vec{p}) = (\{l_1, l_2, ..., l_k\}, \{p_1, p_2, ..., p_k\})$, where p_i is the probability of the instruction taking latency l_i, accomplishing that $\sum_{i=1}^{k} p_i = 1$.

The existence of an ETP for each dynamic (executed) instruction enables the use of MBPTA [2]: the fact the ETPs exist ensures that dynamic instructions behave as random variables. Note that an ETP can be understood as a concrete representation of a random variable, where the different latencies that the instruction may take represent the possible outcomes of the random variable, and the probability of those latencies are the probability of occurrence of the outcomes of the random variable.

Instructions may have *causal dependences* among them [2], such as data or control. Despite *causal dependences*, independence and identical-distribution (i.i.d.) properties[1] still hold across full program runs [2]. This is so because the timing effect of those dependences can be captured by the measurements taken at analysis time. Further, at analysis time the timing effect of the dependences upper-bounds their effect at deployment time. MBPTA has been shown to work in the presence of data dependences among instructions [17] and in the presence of control-flow dependences, i.e. programs with several execution paths [7] and with real avionics applications [30]. We refer the reader to those papers for more details on MBPTA.

The requirements that MBPTA imposes on the timing behaviour of each dynamic instruction can be achieved by either randomising (e.g., using TR caches) or upper-bounding (e.g., variable-latency ALUs) the timing behaviour of all hardware resources that may introduce timing variations by themselves, such that observed instruction latencies during the analysis stage upper-bound probabilistically or deterministically those that can appear during the deployment stage.

2.2 Controlling Cache Inter-task Interferences

Classifying cache accesses as hits and misses in non-shared caches has been already deemed as a complex process subject to some degree of pessimism for those accesses for which it is hard to determine whether they will hit or miss [10, 12, 19, 22, 27, 28]. Thus, extending this process to the coordinated analysis of several tasks simultaneously sharing a LLC is even harder [6, 32]. This is so because any shift in the execution time of any task (e.g., due to accesses whose outcome cannot be accurately predicted) would lead quickly to highly pessimistic assumptions for the other tasks, that must account for all potential time alignments across all tasks. Moreover, combined WCET analysis breaks time composability since any change in the task scheduling or the upgrade of any software component invalidates the WCET estimates of all tasks.

Cache partitioning removes the need for multitask cache analysis by ensuring that tasks are assigned a distinct cache portion. Software cache partitioning is done through memory coloring [14, 20, 23, 29]. By enforcing programs' data/code to be allocated in certain memory addresses (pages) it can be controlled the cache sets in which they are allocated. Hardware cache partitioning assigns different cache ways or cache banks to the different co-running tasks such in a way that contention is prevented [24].

Cache partitioning complicates task scheduling and data sharing. For instance, let us assume a 4-core processor deploying LLC and executing three tasks, τ_A, τ_B and τ_C. Further assume that, the code and data of each task are mapped in memory such in a way that τ_A and τ_B are mapped on the same cache sets and τ_C is mapped to different cache sets. Under this scenario we observe the following problems with software cache partitioning.

- The scheduler has to prevent τ_A and τ_B from running simultaneously because they would interfere each other in cache.
- It is non-obvious how to manage read-write shared data among τ_A and τ_C since data cannot be simply replicated in cache without challenging functional correctness.

Hardware cache partitioning experiences similar problems for data sharing, but different ones for scheduling. Scheduling problems arise when a task, τ_A, uses a given LLC partition, $Part_i$. Some dirty cache lines may remain in $Part_i$ after τ_A is scheduled out. Whenever τ_A is run again, either it is assigned $Part_i$ or it is given a different cache partition. In the latter case $Part_i$ must be flushed before τ_A is scheduled in for consistency.

3. PROBABILISTICALLY CONTROLLING EVICTION FREQUENCY IN A TR LLC

3.1 Inter-task Interferences in a TD LLC

Tasks can interfere each other in non-obvious ways in a shared TD LLC. The main features of two co-running tasks, τ_A and τ_X, that shape their interferences in cache are (1) their memory mapping (i.e. the memory addresses where their code/data are mapped), which determines the cache sets τ_A and τ_X use; (2) the miss frequency in the data cache and the instruction cache of both tasks. The higher the miss rate of a task in the data and instruction caches (assuming a two-level cache hierarchy), the higher its access frequency to LLC and the higher the chances it changes the LLC state and evicts other tasks' data in the LLC; and (3) the their accesses interleave, that is, the particular order in which accesses occur to the LLC which affects τ_A and τ_X behaviour in cache.

It is challenging to control, either by hardware or software means, all three factors due to the high frequency at which cache access occur and the complex interactions among tasks sharing a LLC. In this scenario, LLC partitioning, either software or hardware, is the most feasible way to enable the use of a shared TD LLC in real-time multicore systems. However, as explained before, partitioning challenges task scheduling, data sharing and task migration.

[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 [9]. Two random variables are said to be identically distributed if they have the same probability distribution [9].

3.2 Introduction to TR caches

TR caches [15] randomise the behaviour of the replacement and placement policies. The placement policy selects, based on some bits of the address, the set to access for a given address; while the replacement policy selects, in the event of a miss in a given set, the victim to be evicted. We use *Evict-On-Miss* (EoM) random replacement, which on the event of a miss, randomly selects a victim line from the target set to be evicted, making it analysable with MBPTA. The random placement policy described in [15] deploys a parametric hash function that uses as inputs the address accessed and a random number, called random index identifier or *RII*. Given a memory address and a RII, the hash function 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. If the RII changes across program execution boundaries, programs can be analysed with end-to-end runs simply by assuming that the cache is initially empty. The hash function proposed in [15] ensures that given a memory address and a set of RIIs, the probability of mapping such address to any particular cache set is the same.

In a TR EoM cache with S sets and W ways, hit and miss events are probabilistic. Given the sequence of accesses $seq_1 =< A_i, B_1, ..., B_k, A_j >$, starting from an empty cache state, with A_i and A_j accessing the same cache line, with no B_l (where $1 \leq l \leq k$) accessing the same cache line as A_i and each B_l accessing a different cache line, the miss probability of A_j can be approximated as [15]:

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

The first element in Equation 1 is the probability of miss in a fully-associative cache with W ways deploying EoM random replacement. The base $\left(\frac{W-1}{W}\right)$ is the probability of A_j not to be evicted when a single eviction is performed. The exponent is the addition of the miss probabilities of the elements in between the two accesses to A, which gives a measure of the number of evictions. This first element represents the exact miss probability for a fully-associative cache under the conditions presented above. It becomes an approximation when those conditions change, e.g. when B_l accesses repeat and/or the initial cache state is not empty. However, this is irrelevant for MBPTA, since what really matters is that each access has a probability of hit/miss rather than the particular value of that probability.

The second element in Equation 1 approximates the probability of miss in a direct-mapped cache with S sets. 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 these probabilities.

Equation 1 provides an intuition on the fact that in a TR cache the key factors affecting the probability of hit of an address A is the number of different accesses carried out in between its current (A_j) and previous access (A_i), called reuse distance of A_j, and the miss probabilities of those accesses.

3.3 Inter-task interferences in a TR LLC

As identified in the previous section there are several features affecting the timing behaviour of a task in a TD LLC. Next, we present each of those features for a TR LLC.

Data/Code memory mapping: Under MBPTA the software unit under study is executed enough times according to MBPTA's convergence criteria to ensure representativity of the results. The number of runs required for different benchmarks [7, 15] and an avionics case study [30] ranges between 300 and 1,000. In each run, a new RII is generated, which makes random placement map each address to a new randomly chosen cache set. This removes the dependence between the memory address of an access and the cache set in which it is mapped.

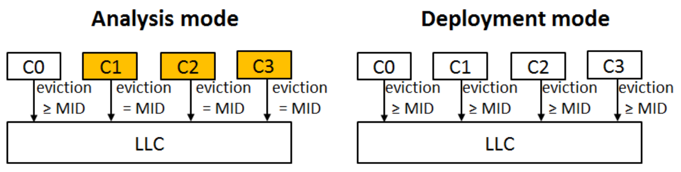

Figure 1: Operation mode for each core at analysis and deployment time. The task under analysis is run in core 0 (C0) at analysis time.

Access and miss frequency: In a TD LLC hit accesses modify the *LRU stack* (i.e. the bits used by LRU to determine which line is to be evicted in the event of a miss). Interestingly, *an EoM random-replacement policy is stateless* and hits do not alter the cache state. With EoM, on the event of a hit, neither cache (data) contents nor any replacement information are changed. Only misses, which create evictions, alter the cache state. Hence, if $\{B_l\}$ accesses were generated by the co-runners of a given task τ_A's, we would observe that the larger the number (frequency) of $\{B_l\}$, assuming that each access has a non-null probability of miss, the lower the hit probability of τ_A's following accesses (A_j in this case).

Interleave: the particular instant in which co-runners of a task τ_A access the LLC determines which accesses of τ_A experience a decrease in their hit probability, hence affecting τ_A's execution time behaviour.

3.4 Probabilistically upper-bounding inter-task interference features in a TR LLC

In order to derive trustworthy and tight pWCET estimates in the context of PTA by upper-bounding the effect of inter-task interferences in the LLC while preserving time composability, we propose a LLC *eviction frequency limiting* mechanism (*EFL* for short). *EFL* limits how often a task can evict LLC cache lines. To that end *EFL* controls LLC miss frequency. Miss frequency bounds are applied in a different manner during analysis and deployment stages such that the timing behaviour observed for a program at analysis time upper-bounds its deployment-time behaviour.

Controlling eviction frequency at analysis time. The task under analysis (τ_A) is run in isolation in one core limiting how often it can evict lines from the LLC. *EFL* prevents τ_A from performing an eviction in the LLC until at least Minimum Inter-eviction Delay (*MID*) cycles have elapsed since τ_A last evicted a LLC line. However, LLC hits are allowed to proceed since they do not change LLC state in TR Evict-on-Miss caches. Later we provide details on the hardware implementation. The other cores, by means of our proposed hardware support, generate *artificial requests* that cause LLC evictions at the maximum allowed frequency, i.e. once very *MID* cycles [2]. In this way, τ_A execution times are obtained under the worst intertask interference scenario: maximum eviction rate from other cores.

Controlling eviction frequency at deployment time. Each task is allowed, at deployment time, to generate a new eviction in the LLC as long as at least *MID* cycles have elapsed since its last eviction. This is enforced by *EFL* hardware (described below). In the worst case, at deployment time all the co-runners of τ_A can systematically miss in cache. In order to preserve *time composability*, *EFL* makes no assumption on the miss rate of co-runners. Hence, during analysis time *EFL* forces all the artificial requests to produce evictions in the LLC.

Next we review how inter-task interference features are taken into account by *EFL* .

Data/Code memory mapping. Random placement caches remove the dependence between an addresses and the particular set in which that address is mapped. Hence, the memory ad-

[2]In reality, as explained later in this section, misses occur *on average* every *MID* cycles, but we consider *exactly MID* cycles at this point for the sake of clarity in the explanations.

dresses in which a program maps its data and code affects execution time in a way that is naturally captured by MBPTA [15].

Miss probability. In an Evict-on-Miss TR cache, the higher the probability of miss of the interfering accesses, the higher the probability they evict data of the program under study, τ_A, because on every miss each interfering access causes a LLC eviction. Thus, by enforcing all interfering accesses to cause a LLC eviction at analysis time, our approach upper-bounds the miss probability of the accesses of any co-runner at deployment time. This is illustrated in Figure 1, where we show how co-runners always perform evictions at analysis time, whereas they access normally at deployment time, hence not necessarily missing in every access.

Miss frequency. Our approach imposes a miss frequency at analysis time that cannot be exceeded at deployment time. This is so because misses occur *exactly* every *MID* cycles at analysis time, whereas they occur *at most* every *MID* cycles at deployment time.

Interleave. How accesses of different tasks interleave may impact tasks' timing behaviour, as explained before. Hence, imposing fixed intervals between evictions could cause systematic effects depending on how those evictions interleave with τ_A accesses. Our approach to avoid such effect consists in randomising how accesses interleave, so that the effect of interleaving can be bounded probabilistically. Under MBPTA, by capturing in end-to-end execution time observations enough outcomes of this randomised event, pWCET estimates capture the effect of such event. Hence, we proceed by enforcing *MID* not to be a deterministic value but instead a random value. If, for instance, the desired *MID* is 1,000 cycles, on every access we set up a new *MID* randomly picked between 0 and $2 \times MID = 2,000$ cycles. By doing so (1) actual *MID* values match, on average, the desired *MID* value. (2) At analysis time the probability of experiencing an interfering access in each cycle is homogeneous and random ($\frac{1}{2\times MID+1}$). Therefore, interfering accesses interleave randomly. And (3) at deployment time the number of interfering accesses between two particular accesses of τ_A is still probabilistically lower than those experienced at analysis time.

3.5 Hardware support

EFL deploys a simple access control unit, see Figure 2, that serves as a bridge among each core and the LLC. Through a *rMID* register the system software (i.e. the Operating System) can establish the *MID* value for each core. By means of a *rmode* register, the system software establishes the operation mode, which is either analysis-time or deployment-time mode. The Access Control Unit also has a *cache request generator* (CRG) per core, that sends eviction requests to the LLC as required. To that end cache requests (accesses) are flagged: eviction requests created at analysis time by the CRG are flagged with a *force-miss bit*. Such bit is reset at deployment time, when the CRG is off [3].

In addition to the *rMID*, *rmode* and the CRG, *EFL* needs to limit LLC miss frequency from each core, both at analysis and deployment time. To that end *EFL* deploys a count-down counter (*cdc*) and a pseudo-random number generator (PRNG) per core, see Figure 2. The particular PRNG we have used in this paper is the Multiply-With-Carry (MWC) [21] PRNG, since we have tested that (i) it generates numbers with a sufficiently high level or randomness, (ii) its period is huge, and (iii) it can be efficiently implemented in hardware [4].

On a LLC miss, the PRNG produces a random value in the range $[0, 2 \cdot MID_{desired}]$, which is used to initialise the counter. Such counter is decremented by 1 in each cycle until it reaches zero. *Note that other intervals and probability functions for the latencies are allowed as long as they are kept the same at analysis and deployment phases. Keeping the probability distribution function ensures representativeness of the collected execution times during analysis time w.r.t those that may be exercised at deployment time.*

[3]Note that many current ISAs contain in the opcode for load operations 'hint bits' that could be used for EFL purposes.

[4]The access control unit can use the PRNG used to implement random replacement in first level caches as it provides up to 32 bits per cycle, largely above the bandwidth needed.

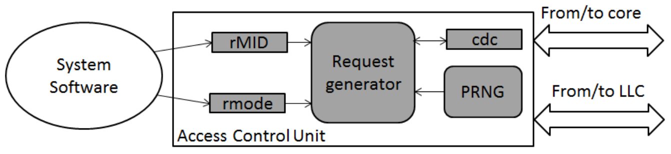

Figure 2: Block Diagram of the Access Control Unit

The port of the core through which the LLC is reached — either directly or through a bus — is extended with a *eviction allowed (EAB) bit* per core. The EAB is set to 1 when the *cdc* reaches zero. If the EAB is 0, it means that such core is not allowed to perform any eviction. The LLC needs to be enhanced such that on a miss of a request with the EAB set to 0 the eviction is delayed until the EAB is set to 1 and the port for such core is set to busy to block any further access. This allows LLC hits to proceed regardless of the count-down counter contents, but stalls misses, which would cause an early eviction otherwise. At analysis time, the *rmode* register is set such that CRGs in all cores but the core where the task under analysis (τ_A) runs, issue uninterruptedly eviction requests to the LLC. All requests, those generated by τ_A and the artificial ones, are controlled by *EFL*, which is also used in all cores at deployment time.

While different cores can update the RII of each of their local caches independently, this is not the case for the LLC. Updating the RII of the LLC must occur coordinately at program execution boundaries. This is doable in many domains such as avionics and automotive which rely on Integrated Modular Avionics (IMA) [1] and AUTOSAR [3] respectively. Both pursue temporal and spatial partitioning. Temporal partitioning is achieved by splitting execution time into fixed-size time frames, which determine the time budget available for task scheduling. For instance, the incarnation of those time frames in IMA are MInor Frames (MIF) and MAjor Frames (MAF). MIF duration is in the order of few milliseconds. Therefore, the OS can easily change the RII of the LLC at MIF boundaries, which occur coordinately across all cores.

Overall, the hardware overhead is negligible since each core needs a small counter (e.g., a 12-bit counter if $MID_{desired}$ is 2,048) and all cores but one need little logic to generate eviction requests. Regarding the PRNG, since first level caches for data (DL1) and instructions (IL1) already implement random replacement in each core, those PRNG can be easily reused. For instance, the PRNG used in [15] is able to provide 32 random bits per cycle, which are far more bits than needed.

4. EVALUATION

4.1 Experimental Setup

We use a cycle-accurate simulator to model a 4-core processor. Each core is 4-stage pipelined with in-order execution. The latency of the fetch stage is 1 cycle on a instruction cache hit. Instruction misses have a variable latency to access memory. After the decode stage, memory operations access the data cache so they can last 1 cycle (hit) or a variable latency to access memory in case of a miss (as described below). Finally, instructions go to the write-back stage.

The memory hierarchy is composed of per-core first level separated instruction (IL1) and data (DL1) caches, a shared cache (LLC) across all cores (for both data and instructions) and main memory. IL1 and DL1 are as follows: 4KB, 4-way and 16B-line size implementing random placement and random replacement (EoM) policies [15]. The LLC is identical to IL1 and

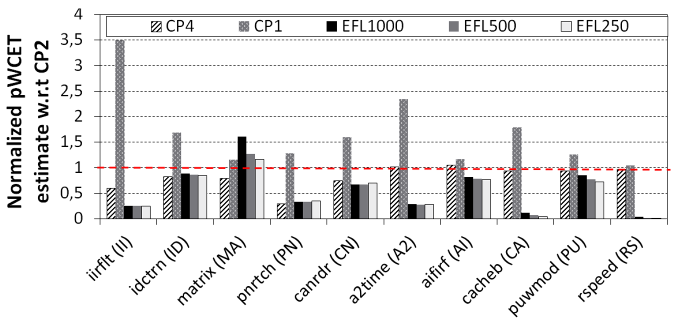

Figure 3: pWCET of each setup normalised to CP2

DL1 except because its size is 64KB and its associativity is 8. The LLC is non-inclusive. All caches are write-back[5]. Instruction/data accesses have 1-cycle IL1/DL1 hit latency, 10-cycle LLC hit latency and 100-cycle memory latency. The remaining operations have a fixed execution latency in the execution stage (e.g. integer additions take 1 cycle). A similar two-level cache hierarchy has been shown to be MBPTA compliant [16] though it was deployed in a single-core architecture with no inter-task interferences. For the bus, whose access latency is 2 cycles, we use random arbitration policies [13]. For the memory controller we use the solution proposed in [25], which upper-bounds the effect of inter-task interferences on the requests of a core to the memory controller.

We use 10 benchmarks with different cache/memory requirements from the EEMBC Autobench suite [26][6], which reflects current real-world demand of some automotive critical real-time embedded systems. WCET estimation is performed in isolation under the analysis operation mode. For the purpose of measuring average and guaranteed performance we have run 1,024 4-benchmark workloads composed of randomly selected Autobench benchmarks. We collected at most 1,000 measurements (i.e. runs) for each experiment.

4.2 Experimental Results

MBPTA compliance MBPTA compliance of our EFL proposal holds by construction as explained in Section 3.4. However, we further contrast this result empirically by using proper statistical i.i.d. tests. Those tests are applied on the execution times coming from running EEMBC benchmarks on our multicore processor architecture deploying *EFL*. For independence we use the Wald-Wolfowitz (WW) test [7] and for identical distribution hypothesis we use the Kolmogorov-Smirnov (KS) goodness-of-fit test [7]. We use a 5% significance level (a typical value for this type of tests), which means that absolute values obtained with the WW test must be below 1.96 and for the KS test, the outcome provided by the test should be above the threshold (0.05). For each such benchmark on the example processor architecture less than 1,000 runs were needed, in line with previous experiences [7]. Our results confirm that for the WW test all results are below 1.96 and for the KS are above 0.05. Hence, for this level of significance ($\alpha = 0.05$), the i.i.d hypotheses cannot be rejected.

***EFL* vs. cache partitioning.** We compare our technique, *EFL*, w.r.t. hardware cache (way) partitioning [24]. For that purpose we consider different configurations of both techniques. For *EFL* we consider *MID* values 250, 500 and 1,000 cycles. We refer to those configurations as EFL*mid*. For cache partitioning we study cache setups with 1, 2 and 4 ways per program. We refer to those configurations as CP*ways*.

Figure 3 shows the pWCET estimates for all benchmarks normalised w.r.t. cache partitioning with 2 ways per program, which would be the solution where each of the 4 cores has exactly 2 out of the 8 LLC cache ways.

Some benchmarks (`ID`, `MA`, `CN`, `AI`, `CA`, `PU`, `RS`) are relatively insensitive to cache space as long as they are given at least 2 ways, i.e. $\frac{1}{4}$ of the cache space. Still *EFL* outperforms *CP* for those benchmarks, since *EFL* does not impose any constraint on the associativity available in each cache set as *CP* does. In particular *EFL* moderately improves *CP* for (`ID`, `CN`, `AI`, `PU`). It is also the case that all those benchmarks but `MA` show to be highly insensitive to the actual *EFL* . `MA` is a benchmark most of whose input set does not fit in LLC. As a result, it experiences a large number of LLC misses and hence, most LLC accesses get delayed due to *EFL*, hence increasing pWCET estimates. In this case it is clear that low *MID* values mitigate this effect.

Finally, `II`, `PN` and `A2`, which are more sensitive to cache space, are less affected by *EFL* than by *CP*.

Overall, *EFL* clearly outperforms *CP* in terms of pWCET estimates across benchmarks, especially for low *MID* values.

Guaranteed performance. The principle of both, *CP* and *EFL*, in order to provide guarantees in the performance of tasks is to reserve resources in the LLC, though *CP* does this in a static manner and *EFL* in a probabilistic manner. For a given benchmark, b, by dividing the instructions committed by b and its pWCET estimate, measured in processor cycles, obtained for *EFL* (or *CP*) for a given cutoff probability (e.g. 10^{-15} per run) we obtain b's guaranteed instructions per cycle (gIPC) for that cutoff probability $gIPC_{EFL}^{-15}(b) = \frac{Instructions(b)}{pWCET_{EFL}^{-15}(b)}$. The workload total guaranteed performance, $wgIPC^{prob}$, is obtained by adding the $gIPC^{prob}$ of the benchmarks in the workload.

wgIPC results can be indirectly obtained from Figure 3. For instance `A2` with CP4 has a normalized execution time close to 1, while for EFL250 it is 0.27. This translates into the fact that with *EFL* the gIPC of `A2` is almost 4 times bigger than for *CP* . In order to measure the *wgIPC* of *CP* and *EFL* in a systematic way, we randomly generated 1,024 workloads and measured the highest $wgIPC^{-15}$ that *CP* and *EFL* can provide under any setup. For *CP* this is equivalent to find the partition of the 8 ways of the LLC across the tasks such that $wgIPC^{-15}$ is maximised. For the case of *EFL* we look for the *MID* value (identical for all tasks) that maximises *wgIPC*.

Figure 4 shows the improvement that *EFL* obtains over *CP* in terms of *wgIPC*. Workloads are sorted from higher to lower EFL improvement. As exceedance probability we have chosen 10^{-15}, with similar results obtained for 10^{-17} and 10^{-19} .

EFL follows an S-curve and improves *CP* in 1,015 out of the 1,024 workloads. For more than 25% of the workloads improvements are higher than 70%, while for more than half it is higher than 47%. The average degradation for the 10 workloads in which *EFL* is worse than CP is smaller than 3.1% (with a maximum degradation smaller than 9.8%). Overall, *EFL* consistently improves *CP*, with a maximum *gIPC* improvement of up to 2.89x an average of 56%.

Average performance. Average performance is also an important metric in real-time systems. High average performance when running critical tasks enables, for instance, running non-critical tasks or saving power by power-gating cores or decreasing their operating frequency. For the same workloads used in Figure 4 we compute their average IPC (*waIPC*) observed at run time. The bottom function in Figure 4 shows *EFL* improvement over *CP* in terms of average performance. We observe that *EFL* improves *CP* in 910 out of the 1,024 workloads. For more than 25% of the workloads improvements are higher than 37%, while for more than half of the workloads is higher than 9%. The average degradation for the 10 workloads in which *EFL* is worse than CP is smaller than 6.4% (with a maximum degradation smaller than 16.8%). As for *wgIPC*, *EFL* consistently improves *CP*, with a maximum improvement of up to 64% and an average of 16%.

[5]If a write-through DL1 cache were used, LLC accesses would be much more frequent due to store instructions. In such case, either write operations are not allowed to allocate data in the LLC on a miss or stalls may be frequent with EFL, thus harming WCET estimates and average performance.

[6]We were not able to compile and execute the rest of the benchmarks in our simulation framework.

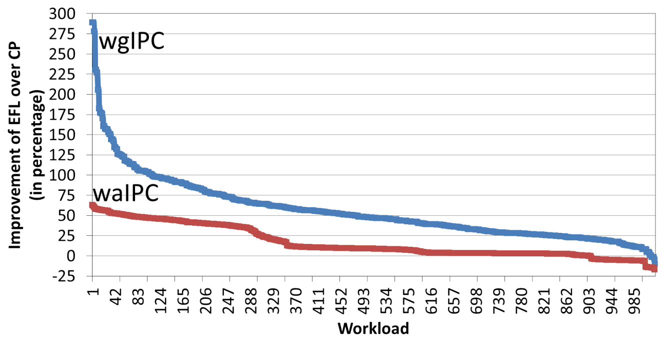

Figure 4: Workload guaranteed IPC (*wgIPC*) and average IPC (*waIPC*) improvement of *EFL* over *CP*

5. CONCLUSIONS

Conventional timing analysis techniques rely on cache partitioning, either hardware or software, to obtain time-composable WCET estimates in systems equipped with shared LLCs as they avoid inter-task interferences. However, cache partitioning challenges data sharing and task scheduling.

In this paper we propose *EFL*, a new technique to obtain time-composable WCET estimates on top of shared non-partitioned LLCs, thus removing partitioning constraints. EFL relies on the fact that time-randomised (Evict-on-Miss) caches used in conjunction with PTA remove the dependence of execution time and WCET on the actual addresses accessed by the program. Thus, by controlling when each core is allowed to evict lines from the LLC and by appropriately producing LLC evictions at analysis time, EFL effectively obtains trustworthy and tight WCET estimates. EFL increases the guaranteed performance w.r.t. cache partitioning by 56% and average performance by 16%.

Acknowledgments

This work has been mainly supported by the PROXIMA FP7 European Project under grant agreement 611085. It has been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557 and the HiPEAC Network of Excellence. Mladen Slijepcevic is funded by the *Obra Social Fundación la Caixa* under grant Doctorado "la Caixa" - Severo Ochoa. Leonidas Kosmidis is funded by the Spanish Ministry of Education under the FPU grant AP2010-4208.

6. REFERENCES

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

[2] J. Abella, E. Quiñones, T. Vardanega, and F.J. Cazorla. Measurement-based probabilistic timing analysis and i.i.d property. White Paper. 2013. `http://www.proartis-project.eu/publications/MBPTA-white-paper`.

[3] AUTOSAR. *Technical Overview V2.0.1*, 2006.

[4] G. Bernat, A. Colin, and S.M. Petters. WCET analysis of probabilistic hard real-time systems. In *IEEE Real-Time Systems Symposium (RTSS 2002)*, Austin, Texas (USA), 2002.

[5] F.J. Cazorla et al. PROARTIS: Probabilistically analysable real-time systems. *ACM TECS*, 2013.

[6] S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *13th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2010)*, St. Goar, Germany, 2010.

[7] L. Cucu-Grosjean et al. Measurement-based probabilistic timing analysis for multi-path programs. In *24th Euromicro Conference on Real-Time Systems, (ECRTS 2012)*, 2012.

[8] R. I. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean. Analysis of probabilistic cache related pre-emption delays. In *25th Euromicro Conference on Real-Time Systems, (ECRTS 2013)*, 2013.

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

[10] C. Ferdinand and R. Wilhelm. Fast and Efficient Cache Behavior Prediction for Real-Time Systems. *Real-Time System*, XVII, 1999.

[11] J. Hansen, S Hissam, and G. A. Moreno. Statistical-based WCET estimation and validation. In *9th Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[12] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *IEEE Real-Time Systems Symposium (RTSS 2008)*, 2008.

[13] J. Jalle, L. Kosmidis, J. Abella, E. Quiñones, and F.J. Cazorla. Bus designs for time-probabilistic multicore processors. In *Design, Automation and Test in Europe (DATE 2014)*, 2014.

[14] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *25th Euromicro Conference on Real-Time Systems, (ECRTS 2013)*, Paris, France, 2013.

[15] L. Kosmidis, J. Abella, E. Quinones, and F.J. Cazorla. A cache design for probabilistically analysable real-time systems. In *Design, Automation and Test in Europe (DATE 2013)*, 2013.

[16] L. Kosmidis, J. Abella, E. Quiñones, and F.J. Cazorla. Multi-level unified caches for probabilistically time analysable real-time systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[17] L. Kosmidis, T. Vardanega, J. Abella, E. Quiñones, and F.J. Cazorla. Applying measurement-based probabilistic timing analysis to buffer resources. In *13th Workshop on Worst-Case Execution Time (WCET) Analysis*, 2013.

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

[19] B. Lesage, D. Hardy, and I. Puaut. WCET analysis of multi-level set-associative data caches. *9th Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[20] J. Liedtke, H. Hartig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Real-Time and Embedded Technology and Applications Symp. (RTAS)*, 1997.

[21] G. Marsaglia and A. Zaman. A new class of random number generators. *Annals of Applied Probability*, 1(3):462–480, 1991.

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

[23] F. Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 1995.

[24] M. Paolieri, E. Quiñones, F.J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *36th International Symposium on Computer Architecture (ISCA 2009)*, 2009.

[25] M. Paolieri, E. Quiñones, F.J. Cazorla, and M. Valero. *An Analyzable Memory Controller for Hard Real-Time CMPs .* Embedded System Letters (ESL), 2009.

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

[27] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37:99–122, November 2007.

[28] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache andpath analyses. *Real-Time Syst.*, 18(2/3), May 2000.

[29] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *25th Euromicro Conf. on Real-Time Systems, (ECRTS)*, 2013.

[30] F. Wartel et al. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, Porto, Portugal, 2013.

[31] 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.

[32] W. Zhang and J. Yan. Static timing analysis of shared caches for multicore processors. *Journal of Computing Science and Engineering*, 6(4), 2012.