Probabilistic Timing Analysis on Conventional Cache Designs

Leonidas Kosmidis[†,*], Charlie Curtsinger[‡], Eduardo Quiñones[†], Jaume Abella[†], Emery Berger[‡], Francisco J. Cazorla[⋆,†]
[†] Barcelona Supercomputing Center (BSC). Barcelona, Spain
[‡] University of Massachusetts (UMass). Amherst, MA, USA
[*] Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
[⋆] Spanish National Research Council (IIIA-CSIC). Barcelona, Spain

Abstract—**Probabilistic timing analysis (PTA), a promising alternative to traditional worst-case execution time (WCET) analyses, enables pairing time bounds (named probabilistic WCET or pWCET) with an exceedance probability (e.g., 10^{-16}), resulting in far tighter bounds than conventional analyses. However, the applicability of PTA has been limited because of its dependence on relatively exotic hardware: fully-associative caches using random replacement. This paper extends the applicability of PTA to conventional cache designs via a software-only approach. We show that, by using a combination of compiler techniques and runtime system support to randomise the memory layout of both code and data, conventional caches behave as fully-associative ones with random replacement.**

I. Introduction

Hard real-time systems depend on computing worst-case execution times (WCET) that upper-bound the amount of time a given computation may take. Static and measurement-based timing analysis techniques [10], [16], [15], [19] have been shown to have significant limitations when used in processors equipped with caches because WCET estimates may be largely overestimated or guarantees on the trustworthiness of the WCET bound cannot be provided [11].

Recently, probabilistic timing analysis (PTA) has emerged as an alternative family of solutions [8], [7]. Unlike standard WCET estimates, PTA yields *probabilistic WCET* or pWCET estimates: time bounds together with an associated exceedance probability (e.g., 10^{-16}). PTA can be applied in either a static context [7] (SPTA) or in measurement-based context [8] (MBPTA). This paper considers MBPTA techniques since they pose fewer requirements to the user.

PTA depends on appropriate hardware designs that allow obtaining probabilities by limiting the dependence of execution time on previous execution history. For example, the state of a least recently used (LRU) cache depends on the address of every object that has been recently accessed. This dependence impedes to predict the likelihood that any given access is a hit or a miss without almost full knowledge of previous accesses, so it is incompatible with PTA. Instead fully-associative cache designs with random replacement (FA-RR) are compatible with PTA since each access has a known hit/miss probability[1] of occurrence [7]. This feature allows execution times to be modelled with random variables that are *independent and identically distributed* (i.i.d.) [12], a characteristic that PTA depends upon. Unfortunately, FA-RR caches are relatively rare because they are power hungry and costly, thus limiting the potential usefulness of probabilistic timing analyses.

The key contribution of this paper is to extend the applicability of PTA to conventional hardware, including direct-mapped and set-associative caches with deterministic placement and replacement policies such as modulo and LRU respectively. We show that the use of randomising compiler techniques that place object code and data in random locations in memory suffices to provide i.i.d. execution times as needed by PTA. We demonstrate empirically that this randomisation comes at an affordable cost, thus making MBPTA practical for the first time on conventional hardware.

II. Background

PTA provides a probabilistic WCET or pWCET distribution that upper-bounds the execution time of the program under analysis. The probability of exceeding a given pWCET is called *exceedance probability*. Among the two PTA approaches (SPTA and MBPTA) this paper focuses on MBPTA as it has been shown to have higher potential impact in real-time industry [8]. MBPTA generates the pWCET distribution by applying *extreme value theory* (EVT) [14], [8] for a collection of observed execution times of the program under analysis.

MBPTA requires that the execution time can be modelled as a i.i.d. random variable: observations are independent across different runs and a probability can be attached to each potential execution time. To that end, those events that may introduce variation in the execution time of a program (e.g. memory operations) must be random events. To do so, the processor architecture must guarantee the timing behaviour of each processor instruction to be represented with Execution Time Profiles (ETPs). An ETP defines the different latencies of an instruction and its associated probabilities of occurrence, represented 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 the program/instruction taking latency l_i [7].

MBPTA, therefore, requires that each memory access has an associated ETP. *The existence of the ETPs for each instruction ensures that the execution times are probabilistic and therefore MBPTA can be applied* [8]. Thus, MBPTA simply needs those ETPs to exist in order to satisfy its requirements (i.i.d. execution times), and unlike SPTA, there is no need to actually compute them.

Previously, MBPTA has only been shown to work with FA-RR caches [8]. In a FA-RR cache, on an access resulting in a miss, each cache line can be selected as victim for replacement with probability $\frac{1}{N}$, where N is the number of cache lines (ways). As a result, the timing behaviour of each cache access can be represented with an ETP, a probability of hit and miss. Other conventional cache designs, e.g. modulo placement and LRU replacement caches, are deterministic by nature, making impossible to attach a hit/miss probability to each memory

[1] Note that this probability is different from frequency

*978-3-9815370-0-0/DATE13/©2013 EDAA

operation, precluding the use of PTA techniques. This paper extends the applicability of PTA to conventional caches by means of compiler and runtime techniques that make execution time to be i.i.d. We verify this using standard statistical tests of independence and identical distribution, and via an informal argument that randomly placing objects in memory guarantees the existence of an ETP for every memory operation.

III. Compiler and Runtime Support for MBPTA

A *memory object* refers to a memory entity, normally stored in consecutive memory addresses (e.g., functions, basic blocks, data structures), which is manipulated by a software component. These objects can be created off-line by the compiler and the linker, or on-line by the program loader and runtime memory-related libraries.

We define a *cache layout* as the result of mapping all memory objects that form a program into the N cache sets of the cache. Under each cache layout of a program, memory objects conflict in a different manner in cache, which, in combination with the replacement policy, may potentially result in different execution times for the program.

Given a set of memory objects and a fixed sequence of memory accesses, deterministic cache designs generate a single cache layout due to deterministic placement, mapping objects into the exact same cache sets on every execution, and the same sequence of accesses in each cache set due to deterministic replacement. As a result, the execution time does not vary across program invocations [2] as long as *(i) objects are always placed in the same memory location and (ii) the same input data set is used, under which a single path in the program is exercised*.

Therefore, forcing randomised timing behaviour on conventional caches, requires the assistance from a specialised compiler and runtime system that randomises the location of objects in memory, and so the cache layout, before execution begins. For the sake of clarity, we first assume that caches are direct-mapped with modulo placement, so there is no replacement policy. We next generalise our approach by considering set-associative caches implementing a replacement policy.

A. Random Location of Memory Objects

The location of memory objects in random memory positions has the effect of leading deterministic direct-mapped caches to behave as random ones. The reason is that randomised layouts lead the cache set to be randomly selected at every new memory allocation, mimicking the behaviour of a random placement policy and so generating random cache layouts across program invocations.

Consider a program formed by a loop in which two leaf functions are called: f_a and f_b, each composed of sequential code. Assume that we execute this program on a processor with a direct-mapped cache implementing a modulo placement policy, and that the total size of the two functions is smaller than the cache size.

Figure 1 shows three different possible cache layouts. In Figure 1(a) the two functions are placed in consecutive memory positions that do not collide with each other, thereby having no cache conflicts among objects (*inter-object conflict*). However, if they are placed in memory positions such that the modulo function makes two pairs of addresses from the two functions collide into the same cache set, the effectiveness of the cache will be decreased because of inter-object conflicts, as shown in Figures 1(b) and 1(c). Randomly mapping memory objects results in random cache layouts, each leading to potentially different execution times.

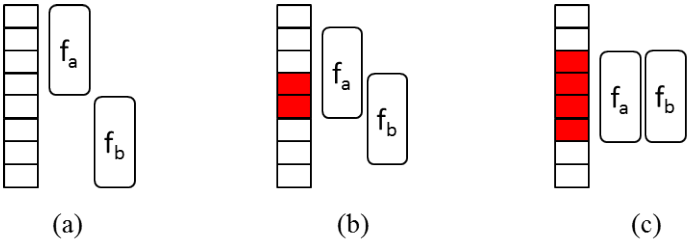


Fig. 1. Different cache locations of functions f_a and f_b in a direct-mapped cache implementing a modulo placement policy. Red (shaded) locations correspond to cache conflicts among the two functions.

Note, however, that cache conflicts within memory objects (*intra-object conflicts*) are deterministic. For instance, if the size of f_a size exceeds the size of the cache, some of its cache lines would be mapped into the same cache set and would conflict. MBPTA requires execution times collected to capture the behaviour of the program under analysis. Such behaviour can manifest in only two ways: (i) constant or (ii) probabilistic, because deterministic non-constant behaviour cannot be modeled with probabilities. If memory objects are defined at a granularity (e.g., function code, stack data) so that their internal layout cannot change, *all* runs of the program will have identical intra-object conflicts and so variation on execution time will be only produced due to random placement of objects in memory.

B. Formal Justification for Applicability of MBPTA

MBPTA requires the existence of an ETP for each instruction [8]. We now argue why randomised layouts guarantee the existence of an ETP for each instruction i accessing to a given cache line, i.e. $ETP(i) = \{l_{hit}, l_{miss}\}\{1 - P_{miss_i}, P_{miss_i}\}$.

A memory operation i accessing cache line c_a belonging to object a will conflict in the cache if there exists another cache line c_b belonging to another object b that is mapped into the same cache set. A modulo placement policy uses some index bits of the memory address to identify the cache set. This approach logically divides the address space into $\frac{M}{N}$ different chunks, where M is the total memory size divided by the cache line size. Within each chunk, memory addresses are mapped to the N cache sets in the same manner. Therefore, the memory chunk in which a memory object is placed is not relevant, but the offset within the chunk. Thus, if we randomly place those objects with respect to memory chunk boundaries (either in a new chunk or in an already in-use chunk if objects do not overlap), inter-object conflicts will occur randomly, and each object will have exactly $\frac{1}{N}$ different placements with respect to the cache (memory objects must be aligned to cache line boundaries, which is usually the case).

Hence, assuming an arbitrary sequence of memory accesses to cache lines $c_a, c_{b_1}, c_{b_2}, \cdots c_{b_m}, c_a$ belonging to objects $a, b_1, b_2, \cdots b_m, a$ respectively, the probability that the second access to c_a is a miss is $P_{miss}(i) = 1 - \left(\frac{N-1}{N}\right)^m$, where $\frac{N-1}{N}$ is the probability that a particular cache line is not placed into a particular cache set and m is the number of unique cache lines accessed in between the two accesses to the cache line c_a.

[2] We consider that other activities, e.g. OS noise, is not considered at WCET analysis but at system integration.

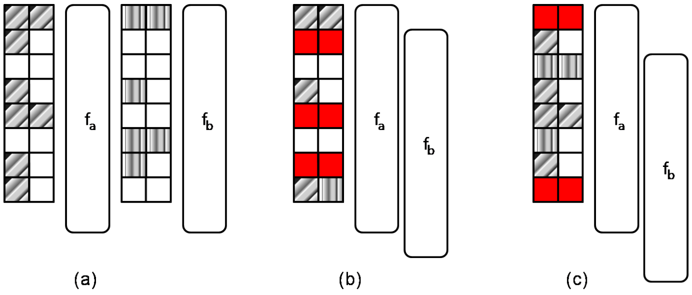


Fig. 2. Cache locations and layouts of functions f_a and f_b in a deterministic two-way set-associative cache. Red regions denote the cache way conflicts between the two functions.

C. *Effect of Replacement Policy*

A cache with a deterministic replacement policy can be made to behave as if it was using random replacement by randomising the order of memory accesses to each particular cache set. Random layout changes the mapping of objects to sets on each execution, thus randomising the order of accesses to each cache set in a random (and thus probabilistic) way.

This effect is illustrated in the following example. Figure 2(a) shows the cache layout of placing f_a (left) and f_b (right) into a two-way set-associative cache. None of the functions has a sequential structure and so they allocate two lines in some cache sets, and only one or zero lines in other sets. This example reflects the cache utilisation of the dynamic invocation of functions when some parts of the code can be skipped due to jump instructions.

When the two functions are co-located in the same cache (Figures 2(b) and (c)), cache lines belonging to f_a and f_b may conflict in some cache sets. Such conflicts will depend on where functions have been randomly placed. Thus, if functions are located as shown in (b), there will be conflicts in 3 cache sets (marked in red), as 3 or 4 different cache lines are candidates for only two ways. This is not the case for the last cache set, in which cache lines belonging to f_a and f_b fit. Instead, if functions are located as shown in (c), there will be conflicts in only 2 cache sets (marked in red), different from the ones that occur in (b).

As shown, random layout of memory objects randomises the cache lines from each object colliding into each set, so the accesses to each cache set (those determining the behaviour of deterministic replacement policies such as LRU) will be determined by random events (the particular random layout). This ensures that inter-object conflicts do not occur deterministically, and their effects can be captured by ETPs.

Note that using a hybrid solution, combining randomised layout with hardware random replacement, would also cause both inter- and intra-object conflicts to occur probabilistically but would increase the degree of randomisation [2], [13].

D. *Randomising Compiler and Runtime System*

We evaluate the effectiveness of software randomisation using *Stabilizer* [9]. Stabilizer comprises both a compiler transformation (using LLVM [1]) and a runtime system that randomises the layout of functions and stack frames. Stabilizer uses the DieHard memory allocator [3] as the basis of its runtime system to perform efficient ($O(1)$) dynamic layout randomisation.

IV. Results

A. *Experimental Setup*

All measurements presented here are conducted on a PowerPC-compatible cycle-accurate execution-driven simulator based on the SoCLib simulation framework [20], modelling a memory hierarchy composed of separate instruction and data caches and main memory. Both caches model a 4KB set-associative cache with 8 ways, 32 sets and 16-byte line size, implementing a modulo placement policy with LRU or random replacement policy. We use a subset of the EEMBC Autobench benchmark suite [18] for evaluation: *a2time01*, *cacheb01* and *puwmod01*. To compute pWCET estimates, we use the method in [8].

B. *Independence and Identical Distribution Tests*

In order to test independence and identical distribution, we use the Wald-Wolfowitz independence test [6] and the two-sample Kolmogorov-Smirnov identical distribution test [5] as described in [8]. We have evaluated i.i.d. properties for the three benchmarks under analysis considering two cache configurations implementing modulo placement and LRU replacement policies (labelled as *mod+lru*) and modulo placement and random replacement policies (labelled as *mod+rr*). For all cases, the p-values obtained (not shown due to space constrains) pass the tests ($p - value > 0.05$ for identical distribution and $p - value < 1.96$ for independence), indicating that both cache configurations provide i.i.d. execution times when we randomise function and stack layout.

C. *pWCET Estimates*

Figure 3 shows the pWCET estimates obtained with MBPTA [8] for *a2time* (a), *cacheb* (b) and *puwmod* (c), considering our two cache configurations. In all cases, we require less than 1,000 runs to project the tail.

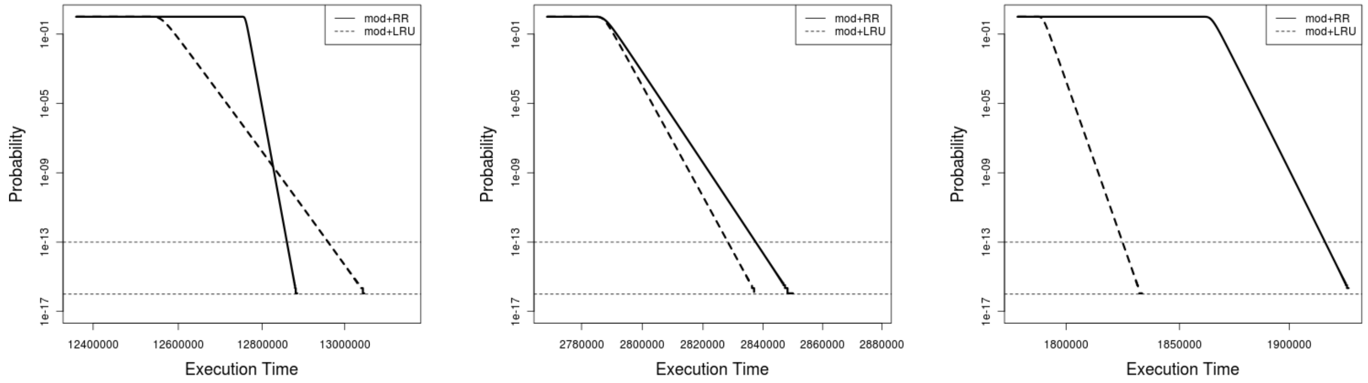
The effect of using a random replacement policy instead of LRU replacement policy depends on the program. If we consider the pWCET estimates at an exceedance probability of 10^{-16}, random replacement increases the pWCET estimate of *puwmod* by 5% over LRU. However, for *a2time*, random replacement reduces the pWCET estimate by 2% over LRU. For *cacheb*, there is almost no variation in pWCET estimates between random and LRU replacement policies (less than 1%).

These results support the analysis of Section III: software randomisation makes it possible to apply MBPTA without requiring additional hardware support such as a random replacement policy. Nonetheless, the use of a random replacement policy remains desirable as it further randomises inter-object and intra-object conflicts.

D. *Overhead*

Our software randomisation approach introduces some overhead due to the relocation of functions and stacks. The former copies each function to a new location. The latter causes each function call to move the stack to a new location.

In order to understand the impact on pWCET estimates, we repeat the same experiment as in the previous section but on top of a FA-RR cache, where software randomisation has no effect on timing behaviour. As a result, the pWCET estimate increment observed with respect to not applying software randomisation is only due to the overhead. We have designed a specific synthetic benchmark consisting of a loop which contains calls to four distinct functions. This structure is very similar to EEMBC.



(a) a2time01

(b) cacheb01

(c) puwmod01

Fig. 3. pWCET estimations of caches implementing modulo + LRU and modulo + random replacement (labelled as *mod+lru* and *mod+rr* respectively).

When considering the pWCET estimate increment at an exceedance probability of 10^{-16} of the synthetic benchmark when applying both function and stack random memory location, we observe that, as we increase the number of iterations, the compiler overhead is reduced, as the relative impact of the initialisation part is reduced. Executing only 100 iterations, the software approach increases pWCET estimates by almost 10x. Such an increment is reduced to 2x when executing 1,000 iterations and only 66% when executing 10,000 iterations.

V. Related Work

Bhatkar et al. [4] introduce stack randomisation as a method for thwarting stack-smashing based security exploits. Berger and Zorn's DieHard system [3] randomises the layout of objects on the heap to provide probabilistic memory safety, tolerating memory management errors. Mytkowicz et al. [17] show that the memory layout may degrade a program's performance by as much as 300%, and propose a random function layout in memory, varying the link and the size of environmental variables. Quiñones et al. [11] explored the effect of memory layout in the WCET of a program and showed that a random replacement policy can lead to less performance variation compared to other policies, while it has acceptable average case performance. Based on this observation, PTA techniques have been developed [8], [7] with the assumption that the underlying architecture uses fully associative caches with random replacement policy. In this paper we provide means to apply PTA on top of conventional cache designs.

VI. Conclusions

This paper presents an approach that extends the applicability of MBPTA to conventional cache designs, e.g. implementing modulo placement and both LRU and random replacement policies, via a software-only randomising compiler and runtime system. Placing functions and stack frames in random memory locations causes deterministic modulo placement policies to exhibit the same behaviour as a random placement policy, yielding observed execution times that satisfy the independent and identically distributed (i.i.d.) properties required by MBPTA. We provide a formal argument explaining how software randomisation enables the derivation of execution time profiles (ETPs) for each memory operation. Finally, we empirically show that software-only randomisation causes deterministic caches to behave as if they were random, making it possible to use MBPTA on top of conventional hardware.

Acknowledgments

This work has been supported by the PROARTIS FP7 Project (grant 249100) and by the Spanish Ministry of Science and Innovation (grant TIN2012-34557) and HiPEAC. Leonidas Kosmidis is also funded by the Spanish Ministry of Education (FPU grant AP2010-4208). Eduardo Quiñones is also funded by the Spanish Ministry of Science and Innovation (Juan de la Cierva grant JCI2009-05455).

References

[1] LLVM. http://dragonegg.llvm.org/.

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

[3] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 158–168, New York, NY, USA, 2006. ACM Press.

[4] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *the 14th USENIX Security Symposium - Volume 14*, 2005.

[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. Technical Report 7869(http://hal.inria.fr/hal-00663329), INRIA, to appear in ACM TECS, 2012.

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

[9] Charlie Curtsinger and Emery D. Berger. Stabilizer: Statistically sound performance evaluation. to appear in ASPLOS 2013, UMCS-TR-2012-012, Department of Computer Science, University of Massachusetts Amherst, 2012.

[10] Christian Ferdinand et al. Reliable and precise WCET determination for a real-life processor. *the 1st EMSOFT*, 2001.

[11] Quinones Eduardo et al. Using Randomized Caches in Probabilistic Real-Time Systems. In *22nd ECRTS*, pages 129–138, 2009.

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

[13] http://www.arm.com. *ARM Cortex-R4 processor manual*.

[14] Samuel Kotz and Saralees Nadarajah. *Extreme value distributions: theory and applications*. World Scientific, 2000.

[15] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Wcet analysis of multi-level set-associative data caches. *the 9th WCET Workshop*, 2009.

[16] Frank Mueller. Timing analysis for instruction caches. *Real-Time Systems - Special issue on WCET analysis archive*, 2000.

[17] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th ASPLOS*, pages 265–276, 2009.

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

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

[20] SoCLib. -, 2003-2012. http://www.soclib.fr/trac/dev.