Probabilistic Timing Analysis on Conventional Cache Designs

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

***Abstract*—Probabilistic timing analysis (PTA) has emerged as a promising alternative to traditional worst-case execution time analyses, which can be excessively pessimistic on modern hardware. PTA enables probabilistic worst-case execution times, pairing time bounds with a probability that they will be violated (e.g., with $p < 10^{-16}$). Probabilistic worst-case execution times can potentially enable 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 a compiler and runtime system that perform fine-grained randomised layout of both code and data produces cache behaviour that closely approximates that achieved by randomised cache replacement on unmodified binaries. Our analysis shows that PTA can be employed on standard set-associative caches, significantly broadening its applicability.

I. Introduction

Hard real-time systems depend on computing worst-case execution times (WCET) that bound the amount of time a given computation may take. Unfortunately, these computations are often extremely conservative on modern systems that include hardware caches. These caches offer the potential to dramatically increase performance by reducing memory latency, but can make it difficult to predict the cost of memory accesses.

A number of static analyses provide WCET calculations tailored for systems with caches [9], [19], [18], [22]. In short, these analyses statically model the cache state in order to classify memory accesses as either misses or hits. Unfortunately, this approach requires complete information about access history. Any uncertainty quickly cascades and forces these analyses to conservatively assume that subsequent memory accesses are misses (in the absence of timing anomalies [23]); otherwise, they could *underestimate* the WCET. Since memory accesses that miss in the cache are typically two orders of magnitude slower than cache hits, the WCET computed by these analyses can easily be overly pessimistic.

Dynamic measurement-based analyses combine worst-case measurements computed by actual code execution. However, like all dynamic analyses, their efficacy depends on effective coverage by the inputs used for testing. An input that leads to a memory layout not observed during testing could result in a large number of cache misses, invalidating the WCET calculation.

Recently, probabilistic timing analysis (PTA) has emerged as an alternative family of solutions [12], [11]. Unlike standard WCET estimates, probabilistic timing analysis yields *probabilistic worst-case execution time* (pWCET): a time bound together with an associated probability that this bound will be violated (e.g., $p < 10^{-16}$). Probabilistic timing analysis can be applied in either a static context [11] (SPTA) or in measurement-based dynamic analysis [12] (MBPTA). This paper considers MBPTA techniques.

The probabilities generated by PTA depend on appropriate hardware designs that allow such calculations by limiting the dependence of execution time on execution history. For example, the state of an LRU cache depends on the address of every object that has been recently accessed—This dependence makes it impossible to predict the likelihood that any given access is a hit or a miss without full knowledge of the access history, so it is incompatible with PTA.

One hardware cache design that is compatible with PTA is a fully-associative cache with random replacement. In such a cache, each access has a known hit/miss probability.[1] This feature allows execution times to be modelled with random variables that are *independent and identically distributed* (i.i.d.) [14], a characteristic that PTA, including SPTA and MBPTA, depends upon. Unfortunately, fully-associative caches are relatively rare because they are power hungry and costly limiting the potential usefulness of probabilistic timing analyses.

The key contribution of this paper is to extend the applicability of probabilistic timing analyses to conventional hardware, including set-associative caches with approximate LRU. We show that the use of a randomising compiler—one that places object code and data in random locations in memory—is sufficient to provide the independent and identically distributed (i.i.d.) execution times that MBPTA requires. We demonstrate empirically that this randomisation comes at a modest cost, making MBPTA practical for the first time on conventional hardware.

II. Background

Unlike conventional timing analyses that simply produce a (possibly quite conservative) upper bound on execution time, probabilistic timing analysis provides a probabilistic WCET (pWCET) curve that bounds the execution time of the program under analysis. Each point on the curve represents the execution time of a program on the x-axis, and the probability that execution time will exceed this bound (e.g., 10^{-16}), along the y-axis. The probability of exceeding a bound is, appropriately, called the *exceedance probability*. The probabilistic timing behaviour of a program (or an instruction) can be represented with Execution Time Profiles (ETPs). An ETP defines the different execution times of a program (or latencies of an instruction) and its associated probabilities of ocurrence. That is, the timing behaviour of a program/instruction can be defined 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.

There are two variants of probabilistic timing analysis (PTA): static PTA, or SPTA, and measurement-based PTA, or MBPTA. This paper focuses on measurement-based PTA as it has been shown to have higher potential impact in real-time industry [12]. MBPTA constructs the pWCET curve from a collection of observed execution times of the application under analysis by applying *extreme value theory* (EVT) [17],

[1] Note that this probability is different from frequency: a memory instruction may have a 50% hit probability if at every cache access we flip a coin and hit if we get heads. Conversely, if the instruction hits and misses alternately, it has a hit frequency of 50%.

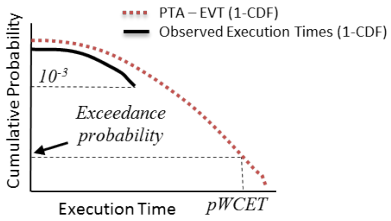


Fig. 1. CDF of observed execution times and tail projection

[12]. EVT allows the projection of the upper-bound of the tail of the cumulative distribution function (CDF) given by the observed execution times. By doing so, MBPTA techniques can provide pWCET estimates for any exceedance probability. Figure 1 shows a hypothetical result of applying EVT to a collection of 1,000 observed execution times. The continuous line represents the inverse CDF derived from the observed execution times; the dotted line represents the projection obtained with EVT, i.e. the actual pWCET curve [15], [8], [12].

MBPTA requires that events under analysis, here execution time observations, to be modelled by *independent and identically distributed* (i.i.d.) random variables. This is the case when observations are independent across different runs and a probability can be attached to each potential execution time. To that end, it is enough if we *make the events that may introduce variation in the execution time of a program (e.g. memory operations) random events*. Hence, taking measurements from a program is equivalent to rolling a dice, with each face having a probability of appearance. Making enough rolls is enough to apply EVT and hence MBPTA.

Independence for a given instruction exists, if the timing probability distribution captured by its ETP is fully independent of the execution history, that is, its ETP holds constant across all executions of the instruction. However, deriving a processor architecture that provides such property is unaffordable [11]. Instead, PTA-imperfect approaches are also feasible. In those approaches the timing vector of the ETP is insensitive to execution history but the probability vector is not, so *means for probabilistically bound that dependence are needed*. The fact that dependences are probabilistic makes that the measurements (execution times) obtained by running the program probabilistically capture the effect of such dependence.

Hence MBPTA requires that: 1) *each memory access has a hit-miss probability*, and 2) *In case memory instructions are dependent, that dependence must be probabilistically modellable*. As noted in [12] *the existence of the ETPs for each instruction ensures that the execution times are probabilistic and therefore MBPTA can be applied*. Hence, MBPTA simply needs those ETPs to exist in order to satisfy its requirements (i.i.d. execution times), but unlike SPTA (the second variant of PTA), there is no need to actually compute them.

Previously, MBPTA has only been shown to work with fully-associative random-replacement (FA-RR) caches. In a FA-RR cache, on a access resulting in a miss, each cache line can be selected as victim for replacement with probability $1/N$, where N is the number of cache lines (ways). For FA-RR caches, the timing behaviour of each cache access can be represented as: $(\overrightarrow{l}, \overrightarrow{p}) = \{l_{hit}, l_{miss}\}\{p_{hit}, p_{miss}\}$, where l_{hit} and l_{miss} are the latency of hit and miss respectively and p_{hit} and p_{miss} the associated probability in each case. Other cache designs, such as conventional modulo placement and least recently used (LRU) replacement caches, are deterministic by nature, making impossible to attach a hit/miss probability to each memory operation, precluding the use of PTA techniques. This paper extends the applicabilty of PTA to conventional architectures by employing compiler and runtime techniques that enforce i.i.d. properties for execution times. We verify this using standard statistical tests of independence and identical distribution, and via an informal argument that random layout 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, and 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 run-time memory-related libraries.

The location at which a memory object is placed into memory determines its location in a cache that implements deterministic placement policy, such as modulo placement. In particular placement policies determine the cache set in which the memory object is allocated based on its memory address. Analogously, if deterministic replacement policies such as LRU are used, the access pattern dictated by the program under analysis determines the cache way in which data are stored based on its access history with respect to the other cache accesses that are located in the same cache set.

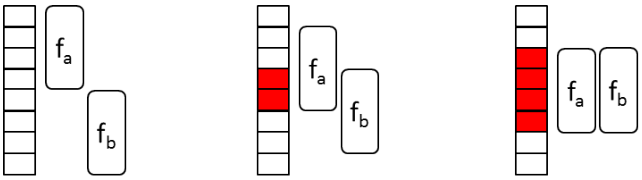
We define a *cache layout* as the result of assigning all memory objects that form a program into the N cache sets of the cache. Under each cache layout of a program, memory objects conflict in a different manner in cache, which, in combination with the replacement policy, may potentially result in different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

Given a set of memory objects and a fixed sequence of memory accesses, deterministic cache designs generate the same cache layouts 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.*

MBPTA has been shown deal with multi-path programs [12] successfully. However, in order to work with object placement, MBPTA assumes FA-RR caches. FA-RR caches [12] have a single cache set and thus, placement is not an issue because there is a single cache layout (all objects are mapped into the only cache set). Random replacement introduces the randomisation needed by MBPTA. Hence, FA-RR caches cause execution time to change across executions. These differences have the disadvantage of adding variance but the significant advantage of producing i.i.d. execution times. The reason is that the location in which data (at cache line granularity) are placed into cache is randomly selected, independent of the location of objects in memory.

Enabling MBPTA to use conventional cache architectures (that is, deterministic set- associative caches) would greatly enhance the applicability of probabilistic timing analysis because such caches are present in nearly all processors. We now explain how it is possible to achieve the *effect* of randomised hardware on conventional caches with assistance

[2]The effect of other activities affecting the execution time of the program, such as OS noise, is not taken into account by the WCET analysis tool but rather at the system integration level.



(a) (b) (c)

Fig. 2. 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.

from a specialised compiler and runtime system that randomise the location of objects in memory 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 that the cache size. Since caches are typically much larger than functions, this is a reasonable assumption.

Figure 2 shows three different possible cache layouts. In Figure 2(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 2(b) and 2(c). Random layout of memory objects results in random cache layouts, each leading to potentially different execution times.

Note, however, that the 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. Therefore, we require that the compiler, linker and runtime have the ability to align memory objects to cache line boundaries so that *intra-object conflicts* are constant. In this way, *all* runs of the program will have identical intra-object conflicts. By doing this, memory objects can only introduce execution time variations due to random placement of objects in memory.

B. Effect of Placement Policy

This section shows how random layout causes the modulo placement policy to exhibit the same i.i.d. properties as random placement;

A modulo placement policy uses the index bits of the memory address to identify the cache set. This approach

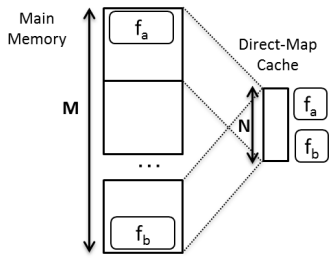


Fig. 3. Location of functions f_a and f_b in main memory.

logically divides the address space into M/N different chunks, where M is the total number of main memory entries. Within each chunk, memory addresses are mapped to the N cache sets in the same manner.

Figure 3 shows a logical address space division produced by the modulo placement policy, and the location in main memory and in cache of functions f_a and f_b. As shown, the memory chunk in which a memory object is placed is not relevant. Instead, what really determines how memory objects will conflict in cache is the offset of the objects within chunks. 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. Further, each object will have exactly $\frac{1}{N}$ different placements with respect to the cache given that, as stated before, memory objects are aligned to cache line boundaries.

C. Formal Justification for Applicability of MBPTA

MBPTA requires the existence of an ETP for each instruction [12]. We now argue why randomised layout guarantees the existence of a ETP for each instruction i, 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. 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 $a, b_1, b_2, \cdots b_m$ respectively, the probability that the second access to c_a is a miss can be given as $P_{miss}(i) = \left(\frac{1}{N}\right)^m$, where $\frac{1}{N}$ is the probability that a particular cache line is placed into a given cache set and m is the number of unique cache lines accessed in between the two accesses to the cache line c_a.

As in [11],[12], accesses to different pieces of data belonging to a particular cache line are considered as accesses to the same address since all of them access the same cache line.

Recall that the value of the ETP does not need to be known for MBPTA (although it is needed for SPTA [11]). For MBPTA, it is enough that these ETPs exist. As shown, miss (and thus hit) probability can be computed, as needed to generate an ETP for memory operations, so i.i.d. properties are fulfilled and we can derive probabilistic WCETs using MBPTA techniques.

D. Implications of Memory Object Size

a) Big objects: Intra-object conflicts: While random layout forces inter-object conflicts to occur with a given probability (as shown in Section III-C), this is not necessarily the case for intra-object conflicts that are not affected by the memory location.

For example, consider the (contrived) case of an object spanning across 9 cache lines and an 8-set direct-mapped modulo-placement cache. In such an arrangement, the first and

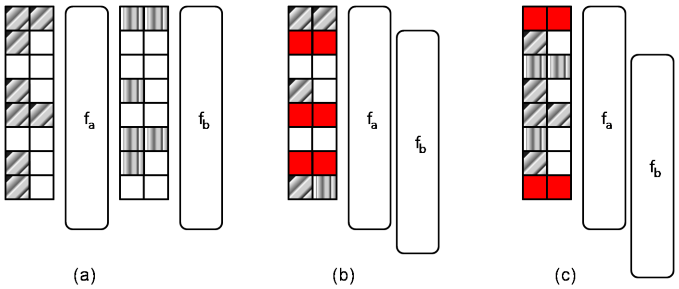


Fig. 4. Cache locations and layouts of functions f_a and f_b in a deterministic two-way set-associative cache. Red regions denote the cache way conflicts between the two functions.

the last cache lines of the object will always collide (map to the same cache line), regardless of where the object starts.

However, this deterministic behaviour does not affect our analysis because intra-object conflicts occur independently of the location of memory objects. In general, as long as the access pattern inside objects is fixed, then the effect of intra-object conflicts can be seen as a constant that does not affect the i.i.d. property of memory accesses. On the other hand, different access patterns within a memory object can be only generated across different input sets, which are properly characterised by performing a sufficiently large number of runs (typically in the order of few hundreds) for each path as described in [12].

b) Small objects: Typical cache line sizes are in the range of 16 to 256 bytes. Some memory objects such as the code or stack of functions may be rather small so that more than one object (e.g., two functions) fit inside the same cache line. If two such objects can be placed into the same cache line, violating MBPTA assumptions because execution time variations could arise from a non-probabilistic source. Therefore, the compiler, linker and runtime software must impose that *two memory objects are either always or never placed in the same cache line*. In this way, the execution time of any run (observation) of the program used to feed MBPTA always reflects the actual conflicts, removing this conflicts as a source of execution time variation among different runs. the real behaviour of the program. Otherwise, any unobserved behaviour could occur systematically once the system is deployed, thus invalidating the analysis.

E. Effect of Replacement Policy

Deterministic replacement policies (e.g. LRU) select the cache way using the order of accesses to the contents of the selected cache set. 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 4(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 invokation 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 4(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 coexist in it. 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.

A hybrid solution, combining randomised layout with random replacement, would cause both inter- and intra-object conficts to occur probabilistically and would increase the degree of randomisation. This cache configuration is becoming increasingly popular and is implemented in a number of current processors [2], [16]. Its recent popularity is due to the fact that randomised replacement decreases the probability of pathological conflicts in cache sets induced by deterministic replacement policies, thus decreasing the probability of degenerate intra-object cache layouts that lead to large numbers of misses (thrashing).

F. Randomising Compiler and Runtime System

We evaluate the effectiveness of software randomisation using *Stabilizer*. Stabilizer comprises both a compiler transformation (using LLVM [1]) and a runtime system that randomises the layout of functions and stack frames [7]. Stabilizer uses the DieHard memory allocator as the basis of its runtime system to perform efficient ($O(1)$) dynamic layout randomisation [3]. overruns and use-after-free errors, DieHard's sparse random heap structure is ideal for ensuring that allocated objects have an equal probability of mapping to each cache set.

1) Function Randomisation: Function randomisation works by relocating a function at startup time by copying its body to a new random memory location. A *Relocation Table* (RT) is placed at the end of each new relocated function to identify the addresses of all globals and functions pointed by the relocated function.

2) Stack Randomization: Stabilizer randomises the stack by making it non-contiguous: each function call has a stack frame in a random location. These randomly-placed frames are also allocated via DieHard, but to reduce overhead, Stabilizer reuses them for some time before they are freed. This bounded reuse improves cache reuse and reduces the number of calls to the allocator while still enabling re-randomisation.

Both function and stack randomisation enable randomising the location of memory objects in memory so that execution time variations depend solely on random events by removing the structural dependence on the actual memory location of objects. Hence, i.i.d. properties needed for MBPTA are fulfilled.

IV. Experimental Setup

All measurements presented here are conducted on a PowerPC-compatible cycle-accurate execution-driven simulator based on the SoCLib simulation framework [24]. The simulator models a 4-stage pipelined processor with a memory hierarchy composed of separate instruction and data caches and main memory. Both caches model a 4KB set-associative

TABLE I
I.I.D. TESTS FOR CACHES USING MODULO + LRU*mod+lru* AND MODULO + RANDOM REPLACEMENT (*mod+rr*).

Cache	Benchmark	id. distr.	ind.	*Passed?*
mod+rr	a2time	0.73	0.38	ok
	cacheb	0.52	0.13	ok
	puwmod	0.55	0.51	ok
mod+lru	a2time	0.25	0.38	ok
	cacheb	0.45	1.45	ok
	puwmod	0.29	0.13	ok

cache with 8 ways, 32 sets and 16-byte line size, implementing a modulo placement policy with an LRU or random replacement policy. The data cache implements a write-through, no- allocate write policy. The only source of execution time variation in the processor is the cache.

We use a subset of the EEMBC Autobench benchmark suite [21] for evaluation. These benchmarks implement representative algorithms that are used in hard real-time systems, and have been specially designed for embedded environments. Since benchmarks are much smaller than real applications, the number of functions and function calls inside does not suffice in general to illustrate the potential of our approach. Therefore, we have used 3 benchmarks: *a2time01*, *cacheb01* and *puwmod01*. *Cacheb01* has been left unchanged. *A2time01* and *puwmod01* have a single main loop. We have transformed the code inside loops into a sequence of function calls so that function and stack randomisation has some effect.

To compute pWCET estimates, we use the extreme value theory (EVT) methodology presented in [12]. This method starts from a sample of execution times of the program under analysis (in our case, 1000 runs). A technique called *block maxima* is then used to extract those execution times used to project the tail of the execution time distribution. We use groups of 50 elements as it has been shown to provide good results [12]. Finally, the resulting execution times are then used by EVT to determine the parameters of a Gumbel distribution that approximates the tail by applying a linear regression to a QQ- plot of our data [10].

V. RESULTS

A. Independence and Identical Distribution Tests

Table I shows the results of the i.i.d. test 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*).

In order to test independence we use the Wald-Wolfowitz independence test [6]. We use a 5% significance level (a typical value for this type of tests), which means that absolute values obtained after running this test is lower than 1.96 if there is independence, and higher otherwise. For identical distribution, we use the two-sample Kolmogorov-Smirnov identical distribution test [5] as described in [12]. For 5% significance, the outcome provided by the test should be above the threshold (0.05) to indicate identical distribution, and non-identical distribution otherwise.

For all cases, the p-values obtained pass the tests ($p - value > 0.05$ for identical distribution and $p - value < 1.96$ for independence), indicating that both cache configurations provide identically distributed and independent execution times when we randomise function and stack layout.

B. pWCET Estimates

Figure 5 shows the pWCET estimates obtained with MBPTA [12] for *a2time* (a) and *cacheb* (b) and *puwmod* (c), considering our two cache configurations. In all cases, we require less than 1,000 runs to project the tail.

TABLE II
OVERHEAD IN THE PWCET ESTIMATES AT AN EXCEEDANCE PROBABILITY OF 10^{-16}, UNDER DIFFERENT NUMBER OF ITERATIONS.

Iterations	Code	Code+Stack
10,000	1.13x	1.66x
1,000	1.6x	2.37x
100	6.22x	9.3x

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 estimation 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.

C. Overhead

The software randomisation approach introduces some overhead. This is due to the relocation of functions, in which the body of each function is copied to a new random memory location; and relocation of the stack, which causes each function call to move the stack to a new random location.

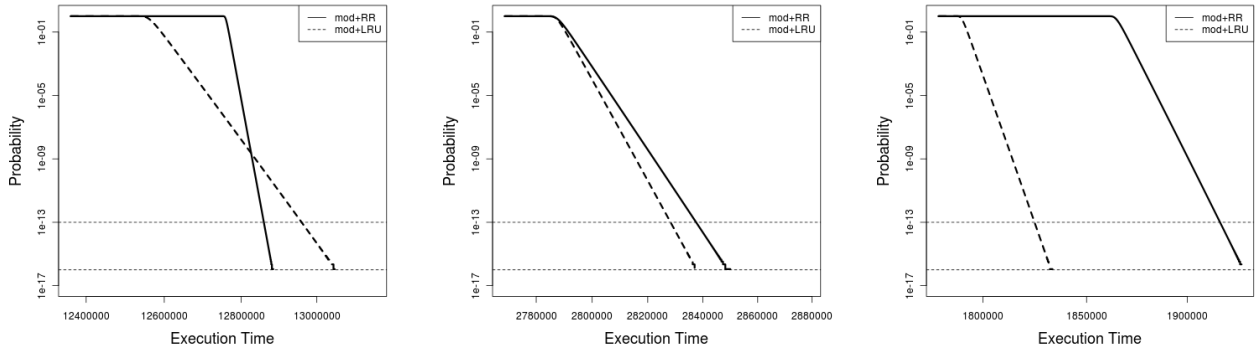
In order to understand the impact on pWCET estimates, we repeat the same experiment as in the previous section but on top of a FA-RR cache, where software randomisation has no effect on timing behaviour: FA-RR caches have only one cache set and the way selection is random. As a result, the pWCET estimate increment observed with respect to not applying software randomisation is only due to the overhead.

Moreover, it is also important to consider the trade-off between the execution time of the new code introduced by the software randomisation technique with respect to the overall execution time of the application under analysis. In order to illustrate this, we have designed a specific synthetic benchmark consisting of a loop which contains calls to four distinct functions. This structure is very similar to the one of the Automotive EEMBC benchmarks. Each of the functions receives two arguments and has a return value, while it contains local variables in order to exhibit stack usage. The code of the functions is linear, without any control flow which may increase the benefit from the instruction cache inside an iteration, and performs integer computations over its local data. No global variables or floating point operations were used.

Table II shows the pWCET estimate increment at an exceedance probability of 10^{-16} of the synthetic benchmark when applying only function random memory location (labelled as *Code*), and applying both function and stack random memory location (labelled as *Code+Stack*). We observe that, as we increase the number of iterations, the compiler overhead is reduced. Thus, when considering only 100 iterations, the software approach increases pWCET estimates by almost 10x. Such an increment is reduced to only 66% when considering 10,000 iterations.

VI. RELATED WORK

Software randomisation has been proposed in the past for enhancing security and performance improvement and evaluation. Bhatkar et al. [4] introduce stack randomisation as a



(a) a2time01 (b) cacheb01 (c) puwmod01

Fig. 5. pWCET estimations of caches implementing modulo + LRU and modulo + random replacement (labelled as *mod+lru* and *mod+rr* respectively).

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 in languages with explicit memory management like C and C++. Mytkowicz et al. [20] show that the memory layout may degrade a program's performance by as much as 300%, and they proposed varying the link order combined with different size of environmental variables, in order to achieve random function layout in the memory. However, this solution requires a costly space exploration and results in a static memory layout, which is the same for every execution of a given binary; in addition, the amount of resulting randomisation is small. By contrast, Stabilizer [7] periodically randomises the program's code, heap and stack during execution to minimize the effect that memory layout has on performance. This finer-grain, dynamic randomisation ensures (with high probability) that measured performance is not the result of performance outliers.

Despite the popularity of software randomisation in the fields of security and high performance, it has found very limited applicability in hard read-time systems, where WCET of the program must be derived and deterministic behaviour has traditionally been considered the ideal. Quiñones et al. [13] explored the effect of the 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, probabilistic timing analysis techniques have been developed [12], [11] with the assumption that the underlying architecture uses fully associative caches with random replacement policy.

VII. Conclusions

This paper presents an approach that extends the applicability of measurement-based probabilistic timing analysis (MBPTA) to systems with a wide range of cache designs, including conventional cache designs implementing modulo placement policy and both LRU and random replacement policies, via a software-only randomising compiler and runtime system. This is a first step towards the use of MBPTA in existing hardware systems.

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 that determine the probability that each memory operation is a hit or a miss. Finally, we empirically show that software-only randomisation causes deterministic caches to behave as if they are random, making it possible to use MBPTA with conventional hardware.

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] Charlie Curtsinger and Emery D. Berger. Stabilizer: Enabling statistically rigorous performance evaluation. Umass-cs-tr-2012-012, Department of Computer Science, University of Massachusetts Amherst, 2012.
[8] S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *the 22nd IEEE Real-Time Systems Symposium (RTSS01)*, 2001.
[9] Christian Ferdinand et al. Reliable and precise wcet determination for a real-life processor. *the 1st EMSOFT*, 2001.
[10] D. Faranda et al. Numerical convergence of the block-maxima approach to the GEV distribution. *Journal of Statistical Physics*, 2011.
[11] Francisco J. Cazorla et al. Proartis: Probabilistically analysable real-time systems. Technical Report 7869 (http://hal.inria.fr/hal-00663329), INRIA, 2012.
[12] L. Cucu-Grosjean et al. Measurement-based probabilistic timing analysis for multi-path programs. In *the 23rd ECRTS*, 2012.
[13] Quinones Eduardo et al. Using Randomized Caches in Probabilistic Real-Time Systems. In *22nd ECRTS*, pages 129–138, 2009.
[14] W. Feller. *An introduction to Probability Theory and Its Applications*. John Willer and Sons, 1996.
[15] J. Hansen, S. Hissam, and G. A. Moreno. Statistical-based wcet estimation and validation. In *the 9th WCET Workshop*, 2009.
[16] http://www.arm.com. *ARM Cortex-R4 processor manual*.
[17] Samuel Kotz and Saralees Nadarajah. *Extreme value distributions: theory and applications*. World Scientific, 2000.
[18] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Wcet analysis of multi-level set-associative data caches. *the 9th WCET Workshop*, 2009.
[19] Frank Mueller. Timing analysis for instruction caches. *Real-Time Systems - Special issue on WCET analysis archive*, 2000.
[20] 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.
[21] Jason Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.
[22] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37:99–122, 2007.
[23] J. Reineke et al. A definition and classification of timing anomalies. In *WCET*, 2006.
[24] SoCLib. -, 2003-2012. http://www.soclib.fr/trac/dev.