Analysis of Probabilistic Cache Related Pre-emption Delays

Robert I. Davis[1], Luca Santinelli[2], Sebastian Altmeyer[3], Claire Maiza[4] and Liliana Cucu-Grosjean[5]
[1]University of York, [2]ONERA Toulouse, [3]University of Amsterdam, [4] Grenoble INP Verimag, [5] INRIA
rob.davis@york.ac.uk, luca.santinelli@onera.fr, altmeyer@uva.nl, claire.maiza@imag.fr, liliana.cucu@inria.fr

***Abstract*—This paper integrates analysis of probabilistic cache related pre-emption delays (pCRPD) and static probabilistic timing analysis (SPTA) for multipath programs running on a hardware platform that uses an evict-on-miss random cache replacement policy. The SPTA computes an upper bound on the probabilistic worst-case execution time (pWCET) of the program, which is an exceedance function giving the probability that the execution time of the program will exceed any given value on any particular run. The pCRPD analysis determines the maximum effect of a pre-emption on the pWCET. The integration between SPTA and pCRPD updates the pWCET to account for the effects of one or more pre-emptions at any arbitrary points in the program. This integration is a necessary step enabling effective schedulability analysis for probabilistic hard real-time systems that use pre-emptive or co-operative scheduling. The analysis is illustrated via a number of benchmark programs.**

I. Introduction

Critical real-time systems such as those deployed in space, aerospace, transport, and medical applications require guarantees that the probability of the system failing to meet its timing constraints is below an acceptable threshold (e.g. 10^{-9} per hour). Advances in hardware technology and the large gap between processor and memory speeds, bridged by the use of cache, make it difficult to provide such guarantees without significant over-provision of hardware resources. The use of deterministic cache replacement policies means that pathological worst-case behaviours need to be accounted for, even when in practice they may have a vanishingly small probability of actually occurring. Further, the quality of deterministic WCET estimates for such systems can be highly sensitive to missing information, making them overly pessimistic. Random cache replacement policies negate the effects of pathological worst-case behaviours while still achieving efficient average-case performance, hence they provide a means of increasing guaranteed performance in hard real-time systems [15]. Determining the timing behaviour of applications running on a processor with a random cache replacement policy requires probabilistic analysis of worst-case execution times and cache related pre-emption delays.

In this paper, we describe a Static Probabilistic Timing Analysis (SPTA) that can be used to compute an upper bound on the exceedance function (1 - CDF) for the probabilistic Worst-Case Execution Time (pWCET) of a program. An example exceedance function is given in Figure 1(b). From the exceedance function, it is possible to read off for a specified probability, an execution time that has that probability of being exceeded on any single run. SPTA computes the upper bound pWCET distribution for a program or task[1] assuming that it is executed non-pre-emptably. Pre-emption by another task results in execution of instructions belonging to the pre-empting task which have an impact on the probabilities of cache hits and misses for subsequent instructions executed by the pre-empted task. We refer to this effect as *probabilistic Cache Related Pre-emption Delay* (pCRPD). Analysis of pCRPD is essential in providing schedulability analysis for probabilistic hard real-time systems that use pre-emptive scheduling.

A. Related Work

Temporal analysis of *probabilistic real-time systems* where at least one parameter, e.g. execution time, is described by a random variable, was first investigated by Lehoczky in 1990 [12] who extended queuing theory under real-time hypotheses. This work was improved upon in 2002 by Zhu et al. [18]; however, the main limitation remained the use of the same probability law for the execution times of all tasks, which is not always realistic. Gardner et al. in 1999 [9] and Tia et al. in 1995 [16] also considered execution times as random variables with special assumptions made about the critical instant. Schedulability analysis for real-time systems with probabilistic execution times was given by Diaz et al. in 2002 [8] and refined by Lopez et al. in 2008 [13]; however, the analysis was difficult to use in practice for computational reasons. Improvements based on re-sampling of random variables were proposed by Maxim et al. in 2012 [14].

In 2009, Quinones et al. [15] investigated the use of random cache replacement policies as a means of obtaining real-time performance less dependent on execution history. In 2012, Cucu-Grosjean et al. [6] and Cazorla et al. [5] introduced SPTA for single-path programs, assuming an evict-on-access random cache replacement policy.

For deterministic systems, the integration of cache related pre-emption delays into schedulability analysis for fixed priority pre-emptive scheduling has been considered by (i) analysing the effect of the pre-empting task (Busquets-Mataix et al. in 1996 [4]), (ii) analysing the effect on the pre-empted task (Lee et al. in 1998 [11]), or (iii) a combination of both (Altmeyer et al. in 2011 [2] and 2012 [3]).

[1]In this paper, we use 'program' and 'task' interchangeably.

In this paper, we build on the idea of using random cache replacement policies in hard real-time systems proposed in [15], and the SPTA for the evict-on-access random cache replacement policy introduced in [6] and [5]; however, we assume an evict-on-miss policy because as we show, its performance dominates that of evict-on-access in terms of the pWCET distributions (exceedance functions) obtained. We extend previous work on SPTA for single path programs given in [6] and [5], both integrating analysis of pCRPD, and providing a method of analysing multipath programs.

Section II presents our system model, terminology and notation. In Section III we provide SPTA for single-path programs assuming an evict-on-miss random cache replacement policy. In Section IV we derive analysis of pCRPD, based on the effect on the pre-empted program. In section V we extend our SPTA and pCRPD analysis to multi-path programs. Section VI applies our analysis to a number of benchmarks, while Section VII concludes with a summary and discussion of future work.

II. System Model, Terminology and Notation

The system we consider is based on a processor with an instruction cache and no data cache. Programs running on this processor are composed of machine code instructions. Each instruction is associated with a *memory block* m in which it is stored. Each memory block may contain a number of instructions (typically 4 or 8) hence multiple instructions may be associated with the same memory block.

A. Random Cache Replacement Policy

We consider a fully associative cache with an evict-on-miss random replacement policy [15]. Here, if the requested instruction is not in the cache, then a cache line is randomly selected for eviction, and the memory block containing the instruction is fetched from main memory and loaded into the evicted location. Thus each cache line has the same probability of being evicted on a miss i.e. for an N-way associative cache, the probability of each cache line being evicted is $\frac{1}{N}$.

B. Instruction Modelling

We assume that the processor executes each instruction in a fixed number of clock cycles, and hence that the only source of instruction timing variation comes from the cache. Each instruction is characterised by two discrete latencies. For a cache hit, H is the time to load the instruction from cache and execute it, and for a cache miss, M is the time to check the cache, fetch the instruction from memory, load the instruction from cache and execute it. For convenience, we assume that the processor takes the same time to execute each instruction once it has been loaded, and hence H and M are the same for every instruction. In practice, a processor may take a different number of cycles to execute different instructions, in which case the analysis we present can be applied with simple modifications provided that the cache miss penalty $(M-H)$ is consistent for all instructions. In the remainder of the paper, we overload the term execution time to mean the overall latency.

We are interested in single-path and multi-path programs. A program path is a sequence of instructions which we represent by a sequence of symbols, one for each instruction, identifying the memory block in which the instruction is stored; for example $a, b, a, c, \ldots$.

Definition 1 (Re-use Distance): Given an arbitrary sequence of instructions, then the *re-use distance* k of a particular instruction is defined by the maximum possible number of evictions[2] since the last access to the memory block containing that instruction.

The re-use distance of an instruction dictates its overall probability of being a hit, with larger re-use distances indicative of a higher probability of a cache miss. We return to the calculation of these probabilities in Section III.

Example 1: For a single-path program described by the following sequence of symbols giving the memory block for *each* instruction $a, b, a, c, d, b, c, d, a, e, b, f, e, g, a, b, h$, its possible representation including re-use distances is $a, b, a^1, c, d, b^3, c^2, d^2, a^5, e, b^4, f, e^2, g, a^5, b^4, h$.

In the above example, the superscripts give the finite re-use distances. As we consider the cache to be initially empty, the re-use distance for the first access to any instruction is ∞. When *consecutive* instructions are in the same memory block, then the second instruction has a re-use distance of zero. This is because its memory block is definitely in the cache after the previous instruction. A re-use distance of zero corresponds to *always hit* and so with an evict-on-miss policy, such instructions do not contribute to the re-use distance of subsequent instructions as they do not result in evictions[3]. For example, with 4 instructions per memory block we may obtain the following sequence of memory block accesses and re-use distances: $a, a^0, b, b^0, b^0, b^0, a^1$.

Each instruction has a probability of being a cache hit $P\{hit\}$, and of being a cache miss $P\{miss\} = 1 - P\{hit\}$. Thus each instruction I is described by a discrete random variable[4] $\mathcal{I}$ representing the execution time of the instruction based on the history of previous accesses. Formally, the Probability Mass Function (PMF) of instruction I is

$$\mathcal{I} = \begin{pmatrix} H & M \\ P\{hit\} & P\{miss\} = 1 - P\{hit\} \end{pmatrix} \qquad (1)$$

C. Program Modelling

Probabilistic real-time analysis focusses on *random variables*, the notion of *independence* , and the "summation" of random variables via the *convolution* operator.

For two random variables $\mathcal{X}_1$ and $\mathcal{X}_2$ defined on the same probability space, the joint distribution defines the probability of events[5] defined in terms of the random variables,

[2]Actually, the number of evictions in the same cache set; however, as we assume a fully associative cache, there is only one cache set.

[3]Technically, such an access could result in a cache miss and an eviction, but only if it were immediately preceded by a pre-emption; however, in that case we consider the extra eviction as due to the pre-emption.

[4]We make use of calligraphic symbols to denote random variables.

[5]Here an event is defined by the fact that one or more instructions have a given value for the execution time.

$F(x_1, x_2) = P\{\mathcal{X}_1 \leq x_1, \mathcal{X}_2 \leq x_2\}$. The joint probability is different for dependent and independent events.

Definition 2 (Independence): Two random variables $\mathcal{X}$ and $\mathcal{Y}$ are *independent* if they describe two events such that the outcome of one event does not have any impact on the outcome of the other.

The sum $\mathcal{Z}$ of two independent random variables $\mathcal{X}_1$ and $\mathcal{X}_2$ is obtained via convolution: $\mathcal{Z} = \mathcal{X}_1 \otimes \mathcal{X}_2$. For discrete random variables $P\{\mathcal{Z} = z\} = \sum_{k=-\infty}^{+\infty} P\{\mathcal{X}_1 = k\} P\{\mathcal{X}_2 = z - k\}$. Convolution is commutative, i.e., $\mathcal{X}_1 \otimes \mathcal{X}_2 = \mathcal{X}_2 \otimes \mathcal{X}_1$.

Definition 3 (Greater than or equal to - Diaz et al. [13]): Let $\mathcal{X}$ and $\mathcal{Y}$ be two random variables. $\mathcal{Y}$ is greater than or equal to $\mathcal{X}$ (alternatively, $\mathcal{X}$ is less than or equal to $\mathcal{Y}$) denoted by $\mathcal{Y} \succeq \mathcal{X}$ ($\mathcal{Y} \preceq \mathcal{X}$) if $P\{\mathcal{Y} \leq v\} \leq P\{\mathcal{X} \leq v\}$ for any v ($P\{\mathcal{Y} \leq v\} \geq P\{\mathcal{X} \leq v\}$ for any v).

Since the execution time of a program can only take discrete values that are multiples of the processor clock cycle, the execution time of a program path i, assuming the worst-case (empty) initial cache state is given by a discrete random variable $\mathcal{C}_i$. Thus the execution time of path i has a PMF $f_{\mathcal{C}_i}(\cdot)$, with $f_{\mathcal{C}_i}(c) = P\{\mathcal{C}_i = c\}$ giving the probability that the path has an execution time equal to c. $\mathcal{C}_i$ can be represented as follows:

$$\mathcal{C}_i = \begin{pmatrix} C_i^0 = C_i^{min} & C_i^1 & \cdots & C_i^{n_i} = C_i^{max} \\ f_{\mathcal{C}_i}(C_i^{min}) & f_{\mathcal{C}_i}(C_i^1) & \cdots & f_{\mathcal{C}_i}(C_i^{max}) \end{pmatrix} \quad (2)$$

where $\sum_{j=0}^{n_i} f_{\mathcal{C}_i}(C_i^j) = 1$.

As an example, a path i might have an execution time

$$\mathcal{C}_i = \begin{pmatrix} 2 & 3 & 4 & 5 \\ 0.1 & 0.3 & 0.4 & 0.2 \end{pmatrix} \quad (3)$$

meaning that for any given run, there is probability of 0.1 that its execution time will be 2, a probability of 0.3 that its execution time will be 3, and so on.

The execution time of a path can also be described using its Cumulative Distribution Function (CDF) $F_{\mathcal{C}_i}(x) = \sum_{c=0}^{x} f_{\mathcal{C}_i}(c)$, or by the 1-CDF $F'_{\mathcal{C}_i}(x) = 1 - \sum_{c=0}^{x} f_{\mathcal{C}_i}(c) \equiv P\{\mathcal{C}_i \geq x\}$.

Definition 4 (probabilistic Worst-Case Execution Time): The probabilistic Worst-Case Execution Time (pWCET) distribution $\mathcal{Z}$ of a program is a tight upper bound on the execution time $\mathcal{C}_i$ of all possible paths. Hence, $\forall i,\ \mathcal{Z} \succeq \mathcal{C}_i$.

III. Static Probabilistic Timing Analysis

In this section, we derive a lower bound on the probability of a cache hit for each instruction in a single-path program. This lower bound is crucially independent of the previous sequences of cache hits and misses, and instead depends only upon the re-use distance of the instruction. Hence we show how SPTA can be used to determine the pWCET distribution for single-path programs assuming an evict-on-miss random cache replacement policy. Extensions to SPTA for the multi-path case are given in Section V.

With an evict-on-miss random cache replacement policy, the probability of evicting a given cache line is $1/N$ on each miss, where N is the number of cache lines in a set. (As we assume a fully associative cache, N equates to the total number of cache lines). In 2010, Zhou [17] gave the following formula for the *overall* probability of a hit on a particular access to such a cache

$$P^{hit} = \left(\frac{N-1}{N}\right)^{k} \quad (4)$$

where k is the re-use distance of the instruction.

Unfortunately, with an evict-on-miss policy, the probability that an instruction in memory block b is a hit is not independent of whether previous instructions since the last access to b were hits or misses, neither does a sequence of all misses necessarily provide the worst-case scenario. This lack of independence is reflected in the conditional probability. If we know that memory block a was *not* evicted because we observe a hit, then the probability that b was evicted instead may be higher than if we observed a miss for a; effectively there is a dependence via the finite size of the cache. We illustrate this via a simple example.

Consider the sequence of instructions represented by their memory blocks a, b, c, b, a, assuming a cache of size $N = 2$. If the second access to b is a hit, then both b and c must be in the cache at that point, and hence the conditional probability that the second access to a is also a hit is zero. Thus the joint probability that the second accesses to both a and b are hits is zero. This differs from the probability of 1/16 that would be obtained by assuming that all accesses were independent and could potentially be misses causing evictions.

Computing conditional probabilities is exponential in the re-use distance and so quickly becomes intractable. Instead, we derive a lower bound on the probability of a hit that is a function of the re-use distance but *independent* of the pattern of hits and misses for previous instructions. We achieve this by considering the maximum amount of information that could be known due to the behaviour of intervening instructions (e.g. by them being hits). An upper bound on this information is obtained by assuming that the intermediate instruction addresses are all unique and in different memory blocks, which remain in the cache for all of the re-use distance. This reduces the effective size of the cache available to the instruction of interest.

For an instruction with a re-use distance of k, then the probability of a hit can be lower bounded for each value of $h = 0 \cdots k$, where h is the number of potentially evicting accesses that are actually hits. Each such access reduces the effective cache size by 1, but also reduces the number of evictions by 1, hence a lower bound on the probability of a hit $P^{hit}(h)$ given h hits out of the k potentially evicting accesses is given by:

$$P^{hit}(h) = \left(\frac{N-h-1}{N-h}\right)^{k-h} \quad (5)$$

provided that $h < N$, otherwise the effective cache size is zero, as is the lower bound on the probability of a hit.

The function $P^{hit}(h)$ is a monotonically increasing function for $0 \leq h \leq k < N$, and hence $P^{hit}(0) =$

$\min_{0 \le h \le k < N} P^{hit}(h)$. (Proof is given in the appendix of the technical report [7] on which this paper is based). Thus a lower bound on the probability of a hit for arbitrary h is given by:

$$P^{hit} = \begin{cases} \left(\frac{N-1}{N}\right)^k & \text{if } k < N \\ 0 & \text{if } k \geq N, \end{cases} \tag{6}$$

P^{hit} provides a lower bound on the probability of a hit $P\{hit\}$ that is independent of the previous pattern of hits or misses. Hence substituting $P\{hit\} = P^{hit}$ in (1) delivers an upper bound on the 1-CDF of the instruction, and a PMF that can be convolved. P^{hit} is monotonically non-increasing with respect to k.

It is interesting to compare the formula for P^{hit} given by (6) for the evict-on-miss policy, with the equivalent formula given in [5] and [6] for evict-on-access[6]. This formula is reproduced below.

$$P^{hit}_{EoA} = \begin{cases} \left(\frac{N-(k-1)-1}{N-(k-1)}\right)^k & \text{if } k < N \\ 0 & \text{if } k \geq N, \end{cases} \tag{7}$$

We observe that evict-on-miss dominates evict-on-access in the sense that it provides, for every instruction, a probability of a hit that is larger $\forall k < N$. This is because evict-on-miss results in smaller re-use distances[7], and evict-on-access reduces the effective size of the cache by the re-use distance (see [5] and [6]). As an example, with $N = 256$, $k = 104$, $P^{hit}_{EoA} = 0.5$ for evict-on-access and $P^{hit} = 0.66$ for evict-on-miss.

For a single-path program, SPTA computes the pWCET distribution as the joint distribution of the composing instructions. As the lower bound probability of a hit for each instruction, given by (6), is valid irrespective of the previous sequence of hits and misses, we effectively have independence and hence the pWCET distribution $\mathcal{C}_j$ can be obtained via convolution:

$$\mathcal{C}_j = \mathcal{I}_1 \otimes \mathcal{I}_2 \otimes \ldots, \tag{8}$$

where I_i are the instructions, and $\mathcal{I}_i$ their distributions.

A. Program Representation

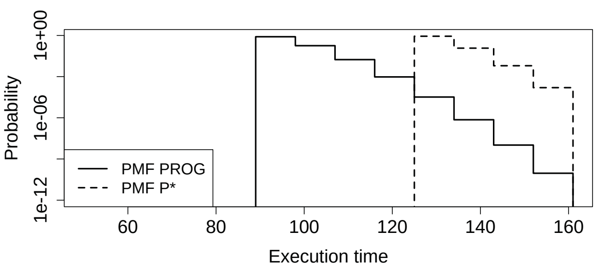
In our model, the only information needed to characterize a memory access is its re-use distance, hence a sequence of n instructions can be represented by the corresponding sequence of re-use distances

$$\mathbb{Q} = \{k_1, k_2, \ldots k_n\}. \tag{9}$$

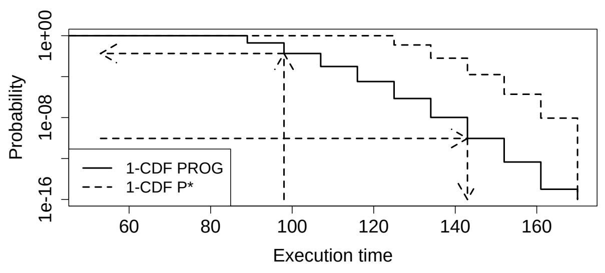
Further, as convolution is commutative, such a sequence $\mathbb{Q}$ can be reordered without changing the final result i.e. the computed pWCET distribution. For ease of use later, we consider $\mathbb{Q}$ ordered by increasing re-use distance.

[6]With evict-on-access, on each request for an instruction a cache line is selected at random and evicted. The cache is then checked to see if the instruction is present and if not its memory block is loaded into the evicted location in cache from main memory. This has the disadvantage that a request for an instruction may evict its own memory block.

[7]If the re-use distance for an instruction is k with evict-on-miss, then it is at least $k+1$ with evict-on-access, as the access for the instruction causes an additional eviction in that case.



(a) Distributions (PMFs)



(b) Exceedance functions (1-CDFs)

Fig. 1. pWCET of the program in Example 2. An exceedence probability of 10^{-9} corresponds to an execution time of 142, while an execution time of 98 has a probability of being exceeded of around 10^{-2}.

Example 2: The sequence of accesses $a, b, a^1, c, d, b^3, c^2, d^2, a^5, e, b^4, f, e^2, g, a^5, b^4, h$ from Example 1 results is the set of re-use distances $\mathbb{Q} = \{-, -, 1, -, -, 3, 2, 2, 5, -, 4, -, 2, -, 5, 4, -\}$ where $-$ stands for an infinite re-use distance. It can be reordered as $\mathbb{Q}^{PROG} = \{1, 2, 2, 2, 3, 4, 4, 5, 5, -, -, -, -, -, -, -, -\}$.

We use $\mathbb{Q}^{PROG}$ to denote the set of re-use distances for all of the instructions in a single path program. The pWCET distribution for a program comprising the sequence of instructions given in Example 2 is depicted in Figure 1. Here, the maximum execution time is obtained when all of the instructions are misses. Note, the latency of a hit and a miss (H, M) are 1 and 10 cycles respectively, and the cache size $N = 256$. An execution time of 142 cycles has a probability of slightly less than 10^{-9} of being exceeded on any given run.

B. Complexity of convolution

The complexity of convolving n instructions (8) might seem to be $O(2^n)$ (i.e. exponential), and indeed this would be the case if the PMF of each instruction contained two arbitrary values. However, the maximum value in the PMF for each instruction is a small constant (M in our model), hence after n convolutions, the largest value in the resulting PMF is nM, and hence a maximum of $2nM$ operations are required to convolve the PMF of the $(n+1)$-th instruction. Thus the complexity is in fact pseudo-polynomial $O(Mn^2)$ where M is a small constant. This makes the method tractable in practice even for quite large values of n. Further, re-sampling techniques can be used to significantly reduce the size of the resulting distributions, with little reduction in precision [14].

IV. Probabilistic CRPD

In this section we study the effect of pre-emption, referred to as the probabilistic cache related pre-emption delay (pCRPD), on the pWCET of single-path programs. (Extensions to the multi-path case are given in Section V). First, we model the effect of pre-emption on single instructions, from this we derive analysis of the pre-emption effect on multiple instructions due to pre-emption at a specific point in the program. We then derive an upper bound on the pre-emption effect at any arbitrary point in the program, and finally the effect of multiple pre-emptions at arbitrary points.

Assuming a sequence of instructions for a program, we use P_p to refer to a pre-emption point after the p-th instruction in the sequence, hence P_1 refers to pre-emption after the first instruction, and so on. Pre-emption at point P_p changes the sequence of instructions executed by effectively inserting a sub-sequence of new instructions. These new instructions are executed prior to the program being resumed and its remaining instructions being executed. Instructions belonging to the program that are contained in memory blocks that are accessed both prior to and after point P_p have the re-use distance of their first occurrence after point P_p increased as a result of the pre-emption. We use the notation $\mathbb{Q}_p$ to represent the set of re-use distances of instructions affected by pre-emption at point P_p. Instructions that are in memory blocks not accessed prior to P_p or not accessed after P_p do not suffer any change in their re-use distances. (We note that the sets of instructions whose re-use distances are affected by pre-emption have some similarities with the sets of Useful Cache Blocks used in the analysis of deterministic cache replacement policies [11]). The increase in re-use distances provides a way of bounding the effect of pre-emption on a per instruction basis, and hence the effect on the overall execution time of the program.

For a single affected instruction I, pre-emption has the effect of changing its distribution from $\mathcal{I}$ to $\mathcal{I}'$. We note that the latencies do not change but the probabilities do, and they change according to (6) such that if k is the re-use distance without pre-emption, then the re-use distance with pre-emption becomes $k' = k + d$, where d is the maximum number of evictions that could be caused due to the pre-emption. Hence pre-emption decreases the probability of a cache hit and increases the probability of a cache miss. Modelling the increased re-use distance in this way gives a safe upper bound on the pre-emption effect, but requires precise information about the increase in the re-use distance caused by the pre-emption (i.e. knowledge of the potentially nested pre-empting tasks).

A simpler upper bound which we consider in this paper is obtained by assuming pessimistically that pre-emption flushes the cache, i.e. evicts all of the cache contents. This can be modelled via the random variable $\mathcal{B}_I$ representing the bounding instruction with an infinite re-use distance, and hence $P^{hit} = 0$ and $P^{miss} = 1$. At the instruction level, intuitively the pre-emption effect is the difference between $\mathcal{I}$ and $\mathcal{B}_I$. The bigger this difference, the larger the pre-emption effect on instruction I.

Recall that $\mathbb{Q}^{PROG}$ is a representation of the program without pre-emption. We can obtain a representation $\mathbb{Q}_{P_p}^{PROG}$ of the program including the effect of pre-emption at some point P_p by removing the values in $\mathbb{Q}_p$ from $\mathbb{Q}^{PROG}$ and replacing them with $|\mathbb{Q}_p|$ infinite re-use distances. We introduce the binary operator pre which does this.

$$\mathbb{Q}_{P_p}^{PROG} = pre(\mathbb{Q}^{PROG}, \mathbb{Q}_p). \tag{10}$$

The instruction distributions corresponding to $\mathbb{Q}_{P_p}^{PROG}$ can then be convolved to obtain an upper bound pWCET distribution $\mathcal{C}^{P_p}$ for the program with pre-emption at point P_p.

Example 3: Returning to our running example $a, b, a^1, c, d, b^3, c^2, d^2, a^5, e, b^4, f, e^2, g, a^5, b^4, h,$ pre-emption after the first a affects just a^1, while pre-emption after the first d modifies a^5, b^3, c^2, and d^2. Hence $\mathbb{Q}_1 = \{1\}$ and $\mathbb{Q}_5 = \{2, 2, 3, 5\}$ represent the sets of instructions affected by pre-emption at points P_1 and P_5, respectively. Without pre-emption, the sequence can be represented by $\mathbb{Q}^{PROG} = \{1, 2, 2, 2, 3, 4, 4, 5, 5, -, -, -, -, -, -, -, -\}$, accounting for pre-emption at point P_5, gives $\mathbb{Q}_{P_5}^{PROG} = \{1, 2, 4, 4, 5, -, -, -, -, -, -, -, -, -, -, -, -\}$.

Our pCRPD analysis makes use of the concept of *dominance* between the effects of pre-emption at different points in the program.

Definition 5 (Dominance among Pre-emption Points): The pre-emption effect due to preemption at a point P_x is said to *dominate* that due to preemption at a point P_y if $\mathcal{C}^{P_x} \succeq \mathcal{C}^{P_y}$ where $\mathcal{C}^{P_x}$ ($\mathcal{C}^{P_y}$) is the upper bound pWCET distribution of the program assuming pre-emption at point P_x (P_y). (See Definition 3 and Diaz et al. [13] for the definition of $\succeq$).

A. Pre-emption Effects on Single Instructions

We now consider dominance among pre-emption effects on single instructions.

Theorem 1 (Instruction Dominance): For a program containing instructions I_x and I_y where the re-use distance of I_x is less than or equal to the re-use distance of I_y and hence $\mathcal{I}_x \preceq \mathcal{I}_y$ (see (6)), then the effect of pre-emption at point P_v affecting only instruction I_x *dominates* the effect of pre-emption at point P_w affecting only instruction I_y, i.e. $\mathcal{C}^{P_v} \succeq \mathcal{C}^{P_w}$.

Proof: The pWCET distribution of the program without any pre-emption may be expressed as $\mathcal{C} = \mathcal{I}_x \otimes \mathcal{I}_y \otimes \mathcal{Z}$, where $\mathcal{Z}$ represents the convolution of the distributions for other instructions. Assuming a pre-emption at point P_v affecting only instruction I_x, then the pWCET distribution of the program is upper bounded by $\mathcal{C}^{P_v} = \mathcal{B}_I \otimes \mathcal{I}_y \otimes \mathcal{Z}$. (Obtained by replacing the distribution for instruction I_x with that given by $\mathcal{B}_I$). Similarly, for a pre-emption at point P_w affecting only instruction I_y, the pWCET distribution of the program is upper bounded by $\mathcal{C}^{P_w} = \mathcal{I}_x \otimes \mathcal{B}_I \otimes \mathcal{Z}$. By Lemma 1 given below, the fact that $\mathcal{I}_y \succeq \mathcal{I}_x$, and the commutativity of convolution, it follows that $\mathcal{C}^{P_v} \succeq \mathcal{C}^{P_w}$ ■

Lemma 1 (Convolution Monotonicity): Considering three discrete random variables $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$ with $\mathcal{Z} \succeq \mathcal{Y}$, then $\mathcal{X} \otimes \mathcal{Z} \succeq \mathcal{X} \otimes \mathcal{Y}$.

Proof: Given $\mathcal{Z} \succeq \mathcal{Y}$ (i.e. $P\{\mathcal{Z} \leq v\} \leq P\{\mathcal{Y} \leq v\}$ for any v), we have

$$P\{\mathcal{X} \otimes \mathcal{Z} \leq v\} = \sum_x \sum_{v' \leq v} P\{\mathcal{X} = x\} P\{\mathcal{Z} = v' - x\}$$

$$= \sum_x P\{\mathcal{X} = x\} \left(\sum_{v' \leq v} P\{\mathcal{Z} = v' - x\} \right)$$

$$= \sum_x P\{\mathcal{X} = x\} \left(\sum_{l' \leq l} P\{\mathcal{Z} = l'\} \right)$$

$$with \quad l = v - x \quad and \quad l' = v' - x.$$

$$= \sum_x P\{\mathcal{X} = x\} P\{\mathcal{Z} \leq l\}$$

$$\leq \sum_x P\{\mathcal{X} = x\} P\{\mathcal{Y} \leq l\}$$

$$since \quad \mathcal{Z} \succeq \mathcal{Y};$$

$$= \sum_x P\{\mathcal{X} = x\} \left(\sum_{l' \leq l} P\{\mathcal{Y} = l'\} \right)$$

$$= \sum_x P\{\mathcal{X} = x\} \left(\sum_{v' \leq v} P\{\mathcal{Y} = v' - x\} \right)$$

$$= \sum_x \sum_{v' \leq v} P\{\mathcal{X} = x\} P\{\mathcal{Y} = v' - x\} = P\{\mathcal{X} \otimes \mathcal{Y} \leq v\}$$

Then $P\{\mathcal{X} \otimes \mathcal{Z} \leq v\} \leq P\{\mathcal{X} \otimes \mathcal{Y} \leq v\}$, hence $\mathcal{X} \otimes \mathcal{Z} \succeq \mathcal{X} \otimes \mathcal{Y}$ ■

B. Pre-emption Effects on Multiple Instructions

We now consider the effect of pre-emption on multiple instructions. Our aim is to determine an upper bound on the effect of pre-emption at any arbitrary point in the program. For mathematical convenience and without loss of generality, we assume that the sets of re-use distances for the instructions affected by each pre-emption point are padded with infinite re-use distance values so that they are all of the same length. For example, $\mathbb{Q}_1 = \{1, -, -, -\}$ is equivalent to $\mathbb{Q}_1 = \{1\}$. We note that this does not change the pre-emption effect represented, as replacing the distribution for an infinite re-use distance instruction by $\mathcal{B}_I$ results in no change. In any case, such padded values will not appear in the final analysis.

We now introduce a binary operator min^+ which applies to our extended (padded) sets of re-use distances. Let $\mathbb{Q}_i = \{k_{i,1}, k_{i,2}, \ldots, k_{i,n}\}$ where $k_{i,r}$ are the re-use distances in order smallest first, and similarly for $\mathbb{Q}_j$. Note $|\mathbb{Q}_i| = |\mathbb{Q}_j|$.

$$min^+(\mathbb{Q}_i, \mathbb{Q}_j) = \{k_r = min(k_{i,r}, k_{j,r}) \; \forall \; r \leq |\mathbb{Q}_i|\} \quad (11)$$

Hence for the sets $\mathbb{Q}_1 = \{1\}$, $\mathbb{Q}_5 = \{2, 2, 3, 5\}$ referred to earlier, we have $min^+(\mathbb{Q}_i, \mathbb{Q}_j) = \{1, 2, 3, 5\}$. We note that $min^+()$ is associative; for brevity in the remainder of the paper, we assume that it may take multiple parameters.

Theorem 2 (Pre-emption Point Dominance): The effect of pre-emption at point P_x *dominates* the effect of pre-emption at point P_y (i.e. $\mathcal{C}^{P_v} \succeq \mathcal{C}^{P_w}$) if $\mathbb{Q}_x = min^+(\mathbb{Q}_x, \mathbb{Q}_y)$, where $\mathbb{Q}_x$ and $\mathbb{Q}_y$ are the extended representations of the re-use distances of the instructions affected by pre-emptions at points P_x and P_y respectively.

Proof: Follows by applying Theorem 1 to the pairs of instructions and re-use distances represented by corresponding elements of $\mathbb{Q}_x$ and $\mathbb{Q}_y$ (i.e. $k_{x,r}$ and $k_{y,r}$ $\forall r$), and the fact that convolution is commutative ■

Corollary 1 (Pre-emption Point Distributions): It follows from the proof of Theorem 2, that the effect of pre-emption at point P_x dominates that for pre-emption at point P_y if $\mathcal{X} \preceq \mathcal{Y}$ where $\mathcal{X}$ ($\mathcal{Y}$) is the convolution of the distributions of the extended (padded) set of instructions affected by pre-emption at point P_x (P_y).

Theorem 2 and the min^+ operator allow us to construct the pre-emption effect of a virtual pre-emption point P^* that dominates the effect of pre-emption at any point, and hence upper bounds the effect of pre-emption at any arbitrary point in the program.

$$\mathbb{Q}^* = min^+(\mathbb{Q}_1, \mathbb{Q}_2, \ldots, \mathbb{Q}_n) \quad (12)$$

We note that $\mathbb{Q}^*$ does not include any infinite re-use distances, but may include re-use distances obtained from a number of real pre-emption points. Hence the pre-emption effect captured by this virtual pre-emption point is a safe upper bound, but may be pessimistic.

Theorem 3 (Dominant Pre-emption Point): The upper bound $\mathcal{C}^{P^*}$ on the pWCET distribution of the program assuming the pre-emption effect represented by the virtual pre-emption point P^*, is *greater than or equal to* the upper bound on the pWCET $\mathcal{C}^{P_x}$ assuming pre-emption at any single arbitrary point P_x (i.e. $\mathcal{C}^{P^*} \succeq \mathcal{C}^{P_x}$).

Proof: Follows from the fact that each instruction affected by pre-emption at point P_x gives rise to a re-use distance that may be paired with a re-use distance in $\mathbb{Q}^*$ that is no larger. Application of the proof of Theorem 1 to each instruction affected by pre-emption at point P_x then suffices to prove the theorem ■

An upper bound on the pWCET of a program assuming a single pre-emption at any arbitrary point can therefore be obtained by applying the effect $\mathbb{Q}^*$ of the virtual pre-emption point to the sequence of instructions of the program and their re-use distances, as represented by $\mathbb{Q}^{PROG}$, via:

$$\mathbb{Q}^{PROG}_{P^*} = pre(\mathbb{Q}^{PROG}, \mathbb{Q}^*). \quad (13)$$

The set of instruction distributions represented by $\mathbb{Q}^{PROG}_{P^*}$ may then be convolved to produce an upper bound pWCET $\mathcal{C}^{P^*}$ for the program which is valid for a single pre-emption at any arbitrary point.

Example 4: Returning to our running example, $a, b, a^1, c, d, b^3, c^2, d^2, a^5, e, b^4, f, e^2, g, a^5, b^4, h$, there are 16 pre-emption points with $\mathbb{Q}_1 = \{1\}$, $\mathbb{Q}_2 = \{1, 3\}$, $\mathbb{Q}_3 = \{3, 5\}$, $\mathbb{Q}_4 = \{2, 3, 5\}$, $\mathbb{Q}_5 = \{2, 2, 3, 5\}$, $\mathbb{Q}_6 = \{2, 2, 4, 5\}$,

$\mathbb{Q}_7 = \{2,4,5\}$, $\mathbb{Q}_8 = \mathbb{Q}_9 = \{4,5\}$, $\mathbb{Q}_{10} = \mathbb{Q}_{11} = \mathbb{Q}_{12} = \{2,4,5\}$, $\mathbb{Q}_{13} = \mathbb{Q}_{14} = \{4,5\}$, $\mathbb{Q}_{15} = \{4\}$, $\mathbb{Q}_{16} = \{\}$. Hence, the virtual pre-emption point P^* results in $\mathbb{Q}^* = min^+_{r\in\{1,\ldots,16\}}\{\mathbb{Q}_r\} = \{1,2,3,5\}$.

Figure 1 illustrates the PMF and (1-CDF) of this program with no pre-emption ($PROG$) and accounting for one arbitrary pre-emption (P^*). In the non-pre-emptive case, an execution time of 142 cycles has a probability of 10^{-9} of being exceeded on any given run, whereas with a single arbitrary pre-emption modelled by P^* this increases to 161 cycles.

C. Modelling Multiple Pre-emptions

In the previous sub-section we characterized the effect of single pre-emptions. We now extend our approach to cater for multiple preemptions.

The effect of m pre-emptions can be obtained via a composition of the maximum effect of m single preemptions, and hence the processing of the values in $\mathbb{Q}^*$ m times. Given $\mathbb{Q}^* = \{k_1, k_2, \ldots, k_r\}$

$$m \times \mathbb{Q}^* = \{\underbrace{k_1, \ldots, k_1}_{m}, \underbrace{k_2, \ldots, k_2}_{m}, \ldots \underbrace{k_r, \ldots, k_r}_{m}\} \qquad (14)$$

We note that values in $m \times \mathbb{Q}^*$ may not all appear in $\mathbb{Q}^{PROG}$, indeed, $|m \times \mathbb{Q}^*|$ may be larger than $|\mathbb{Q}^{PROG}|$. To correctly account for m preemptions, we must use the following rules when processing $m \times \mathbb{Q}^*$ and $\mathbb{Q}^{PROG}$.

We process the values in $m \times \mathbb{Q}^*$ in order, smallest first. If the value is present in $\mathbb{Q}^{PROG}$, then we replace it with '$-$' meaning infinite re-use distance (i.e. the bounding instruction). If the value is not present in $\mathbb{Q}^{PROG}$, then we take the next larger value remaining in $\mathbb{Q}^{PROG}$ and replace it with the bounding instruction. We note that this is safe, as such a mismatch is a result of pessimism in the analysis of multiple pre-emptions. It indicates that m pre-emptions cannot affect instructions with this value of re-use distance m times; however, instructions with larger re-use distances could still be affected, and so these re-use distances must be replaced instead.

Example 5: Given a program described by $a, b, c, d, a^3, b^3, c^3, d^3, d^0, d^0, d^0, d^0, d^0, d^0$ then $\mathbb{Q}^* = \{0,3,3,3\}$. Assuming that we are interested in the effect of four pre-emptions, $4 \times \mathbb{Q}^* = \{0,0,0,0,3,3,3,3,3,3,3,3,3,3,3,3\}$. The resultant value of $\mathbb{Q}^{PROG}_{P^*}$ is then $\{0,0,-,-,-,-,-,-,-,-,-,-,-,-\}$ once the pre-emptions are accounted for. We note that in this case, $4 \times \mathbb{Q}^*$ has 16 elements whereas $\mathbb{Q}^{PROG}$ is only of size 14, nevertheless, 4 pre-emptions are insufficient to make all of the re-use distances infinite.

D. Complexity of pCRPD

The complexity of pCRPD analysis can be described in terms of the number of instructions n, the total number of memory blocks S, and the number of pre-emptions Z (where $S < n$ and $Z < n$). The time complexity of the steps in pCRPD analysis are as follows: (i) finding the re-use distance at each program point (or instruction) - $O(n)$ using an array of S values, (ii) computing the representation of the pre-emption cost at every pre-emption point - $O(nS)$, (iii) sorting the values in each of those pre-emption cost representations and combining them to build a representation of the virtual pre-emption point P^* - $O(nSlog(S))$, (iv) sorting the program representation $\mathbb{Q}^{PROG}$ - $O(nlog(n))$, and (v) combining it with the representation of the virtual pre-emption point Z times to bound the effect of Z pre-emptions - $O(Z(n+S)$. The overall complexity of pCRPD analysis is therefore upper bounded by $O(n^2log(n)$. Convolution of the resultant program representation (which accounts for Z pre-emptions) is then required, and has a complexity of $O(Mn^2)$, where M is a small constant, equating to the cost of a cache miss.

V. Multi-path Analysis

In practice, programs may have multiple execution paths rather than the simple sequential execution considered so far. In this case, the static probabilistic timing analysis required to derive the pWCET distribution without pre-emption is more complex because it has to take into account all of the possible paths that the program may execute. Further, analysis of the pCRPD is also more complex as pre-emption may take place at any point on any path. In this section, we present an approach to SPTA applicable to multi-path programs. This approach collapses a multi-path program into a synthetic, single path representation and hence provides an upper bound pWCET distribution for non-pre-emptive execution. We also present a method of deriving an upper bound on the pre-emption effect at any point in a multipath program, compatible with our analysis of the upper bound pWCET distribution for the non-pre-emptive case. The set of pre-emption effects for all pre-emption points can then be reduced to a single dominant pre-emption effect $\mathbb{Q}^*$ as in the single path case, and applied to the synthetic single path representation to compute an upper bound on the pWCET distribution for a multipath program subject to one or more pre-emptions. We use Lambda Calculus to express our multipath analysis.

A. Synthetic Path

We derive one single synthetic path for each program which over-approximates all concrete paths. We first upper-bound at each program point (instruction) and for each memory block the re-use distance using a simple program analysis. We then combine all sub-paths into the single synthetic path. We require the following two assumptions to hold: (i) each loop in the program is bounded and (ii) the code is well-structured. Both assumption hold for most hard-real time systems, which were designed with timing analysability in mind.

1) Program Analysis: We assign each program point a function rd that maps a memory block m to an upper bound on its re-use distance. This means $rd(m)$ at program point P gives the maximal number of evictions since a previous access to m up to program point P. Hence, the domain of the analysis is defined as $rd : \mathbb{M} \rightarrow \mathbb{N}^\infty$, with an initial valuation that assigns ∞ to each memory block (since we cannot assume any prior use of any memory block):

$$\forall m \in \mathbb{M} : rd_{\text{init}}(m) = \infty \tag{15}$$

The transfer function updates the domain of the analysis at each access to a memory block m. If the re-use distance of memory block m was previously zero, then the block is in the cache and the subsequent access is always a hit, and so does not increase the re-use distances of other instructions. Hence, in this case, the transfer function copies the previous values. Otherwise, memory block m is assigned a re-use distance of 0, and the re-use distances of all other blocks are increased by one.

$$tf : (\mathbb{M} \to \mathbb{N}^{\infty}) \times \mathbb{M} \to (\mathbb{M} \to \mathbb{N}^{\infty})$$
$$tf(rd, m) = rd' \tag{16}$$

with

$$rd'(m') = \begin{cases} 0 & m = m' \\ rd(m') & rd(m) = 0 \\ rd(m') + 1 & \text{otherwise} \end{cases} \tag{17}$$

As we are interested in upper bounds on the re-use distances, we compute the maximum of the re-use distances at program joins.

$$\bigsqcup : (\mathbb{M} \to \mathbb{N}^{\infty}) \times (\mathbb{M} \to \mathbb{N}^{\infty}) \to (\mathbb{M} \to \mathbb{N}^{\infty}) \tag{18}$$

$$\left(rd_1 \bigsqcup rd_2\right)(m) = \max(rd_1(m), rd_2(m)) \tag{19}$$

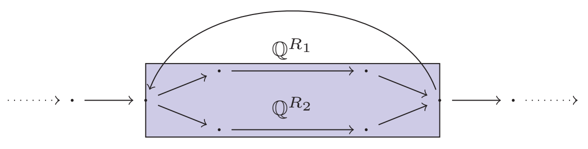
Using these definitions, we can compute a fixed-point of rd on the control-flow graph, which delivers valid upper bounds on the re-use distances for each memory block at each program point. We then replace the nodes within the control-flow graph with the corresponding re-use distance of the memory access.

For a given program point (instruction) accessing m, the re-use distance is given by the maximum value of $rd(m)$ for any immediately preceding program point (instruction).

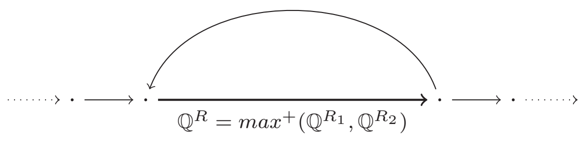
2) Path Combination: We are interested in a synthetic path representing all concrete paths of a program. Explicit enumeration of all paths within a program is however computationally infeasible. The number of paths grows exponentially with the number of loop-iterations and control-flow splits. For instance, the control-flow graph depicted in Figure 2 has up to 2^l different paths, where l is the loop bound. So, we aim for a recursive computation of the synthetic path. To this end, we split the control-flow graph into single-entry, single-exit (sese) regions and compute a synthetic path for each region. Later on, we combine these regions into a single synthetic path for the whole program.

We start with an inner-most sese region R, which does not contain any loop (see Figure 2(a)). We derive all sub-paths $\mathbb{Q}^{R_1}, \ldots, \mathbb{Q}^{R_n}$ from the entry to the exit of the region and replace the complete region with a synthetic path (see Figure 2(b)):

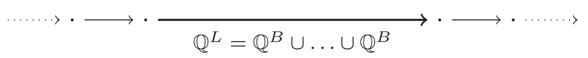
$$\mathbb{Q}^R = max^+(\mathbb{Q}^{R_1}, \mathbb{Q}^{R_2}, \ldots, \mathbb{Q}^{R_n}) \tag{20}$$



(a) Basic Control Flow Graph with an Inner Region R



(b) Region Collapse, Inner Synthetic Path



(c) Loop replacement

Fig. 2. Steps of the Path Combination.

where max^+ is defined in a similar way to min^+, i.e.

$$max^+(\mathbb{Q}^{R_i}, \mathbb{Q}^{R_j}) = \{k_r = max(k_{i,r}, k_{j,r}) \; \forall \; r \leq |\mathbb{Q}^{R_i}|\} \tag{21}$$

however, in this case the sets $\mathbb{Q}^{R_i}$ representing the sub-paths are padded with zeros until they are all of the same length, before being sorted, smallest value first. This ensures that the pWCET distribution for $\mathbb{Q}^R$ upper bounds those for $\mathbb{Q}^{R_i}$ with a minimum amount of pessimism.

For each loop, we first compute the synthetic path of the loop body region $\mathbb{Q}^R$ and create a new path for the loop, in which we duplicate $\mathbb{Q}^B$ (representing the loop body) l-times, where l denotes the upper loop bound (see Figure 2(c)):

$$\mathbb{Q}^L = \underbrace{\mathbb{Q}^B \cup \ldots \cup \mathbb{Q}^B}_{l} \tag{22}$$

We recursively repeat these steps until we end up with one single synthetic path.

B. Pre-emption Effects

We now derive a representation of the pre-emption effect at each point in a multi-path program. As enumerating all possible paths is typically intractable, we instead consider program points on the control flow graph. We assume that each node (program point) on the control flow graph has an associated upper bound re-use distance for its memory access that has been computed as described previously.

1) Program Analysis: We assume that at each program point P, we have computed (via the fixed point of rd), the maximum re-use distance rd_k for the memory access m at that point. We assign each program point a function pe (pre-emption effect) that maps each memory block m to the minimum re-use distance for that block that could be affected by pre-emption immediately prior to the program point. (Note these minimum values are computed with respect to the maximum re-use distances rd_k assumed in the computation of the pWCET distribution for the non-pre-emptive case. This ensures that the treatment of pre-emption effects provides a safe upper bound when applied to the pWCET of the non-pre-emptive case. Larger pre-emption effects could be observed in practice, but only on an instruction where the actual re-use distance is smaller than that assumed by the pWCET analysis. In this case part of the pre-emption effect is already captured as pessimism in the pWCET analysis. This is similar to the deterministic case where CRPD analysis only needs to consider accesses that are not already considered as cache misses by the timing analysis [1]).

The function $pe(m)$ is computed by backwards analysis, starting at the end of the program and working towards the start. The initial valuation assigns ∞ to each memory block, since there is no further use of any memory block after the end of the program.

$$\forall m \in \mathbb{M} : pe_{\text{init}}(m) = \infty \tag{23}$$

The transfer function updates the domain of the analysis at each access to a memory block:

$$tf(pe, m) = pe' \tag{24}$$

with

$$pe'(m') = \begin{cases} rd_k & m = m' \\ pe(m') & \text{otherwise} \end{cases} \tag{25}$$

As we are interested in the largest pre-emption effects, then on program joins the minimum values for re-use distances are taken:

$$\left(pe_1 \bigsqcup pe_2\right)(m) = \min(pe_1(m), pe_2(m)) \tag{26}$$

Using this definition of $pe(m)$, we can compute a fixed point of pe on the control flow graph. The fixed point of pe at each program point gives the set of re-use distances describing the maximum effect of pre-emption immediately prior to that program point. These values may be described in the $\mathbb{Q}$ notation. The $\mathbb{Q}$ values for all possible pre-emption points can then be combined to form a representation $\mathbb{Q}^*$ of the maximum effect of pre-emption at any arbitrary point in the program, via (12) as per the single path case. The upper bound pWCET distribution for the program, assuming one or more pre-emptions at arbitrary points, may then be obtained by applying $\mathbb{Q}^*$ to the synthetic, single path representation using the techniques derived for the single path case. (We note that due to pessimism in the analysis for the multi-path case, not all of the values in $\mathbb{Q}^*$ may appear in $\mathbb{Q}^{PROG}$. This is similar to the multiple pre-emption case for single path programs and the same rules for processing $\mathbb{Q}^*$ apply).

VI. Experimental Evaluation

We applied our integrated probabilistic analysis approach to the FAC, FIBCALL, FDCT, JFDCTINT (single-path with loops) and BS, INSERTSORT, FIR (multi-path) benchmarks from the Mälardalen benchmark suite [10]. We investigated the effect of multiple pre-emptions on the pWCET distribution (1-CDF) of each program, for both the evict-on-miss and the evict-on-access [6] random cache replacement policies for memory block sizes of 1 instruction (4 bytes), and 4 instructions (16 bytes) and a cache of size $N = 128$ blocks (i.e. 512 bytes and 2048 bytes respectively). We also examined the effect on the pWCET of each program obtained by increasing the memory block size from 1, 2, 4, to 8 instructions. Our experiments assumed cache hit and miss latencies of 1 and 10 respectively. The results of all of these experiments can be found in the technical report [7]. There, we provide comparisons against the evict-on-access policy and its basic analysis given in [6] which we have extended to account for the effect of pre-emptions and multiple paths; for reasons of space and clarity, here we only present results for the simple FAC benchmark, assuming the superior evict-on-miss policy.

Figure 3 compares the pre-emption effect for every pre-emption point in the FAC benchmark, together with the resulting dominant virtual pre-emption point P^*, shown by the bold lower line (downwards staircase). This graph shows the 1-CDF of the (padded) set of instructions affected by the pre-emption that are replaced by a set containing an equal number of bounding instructions. Thus an intuitive interpretation is that the pre-emption effect corresponds to the area between the line running horizontally across the top of the graph and then vertically down (the bounding instructions) and the staircase corresponding to each pre-emption point. Thus the largest effect is for the virtual pre-emption point P^*. In this case, at most 8 instructions can be affected by a single pre-emption.

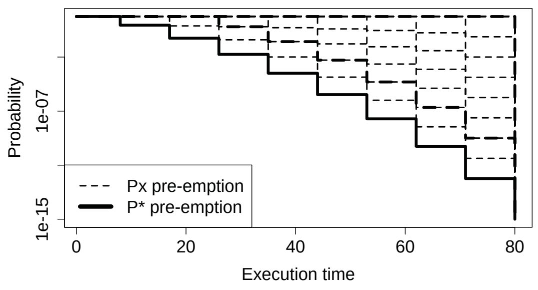


Fig. 3. FAC: 1-CDF representation of the pre-emption effect at every pre-emption point P_x, and also for the virtual pre-emption point P^*, assuming a memory block size of 1, cache size of $N = 128$.

Figures 4 and 5 give the pWCET distribution (1-CDF) for the FAC benchmark, assuming an evict-on-miss policy, and memory blocks sizes of 1 and 4 instructions respectively. Also shown on these graphs are the upper bound pWCET distributions accounting for $1, 2, 3, \cdots$ pre-emptions. The horizontal dotted line on each of the graphs represents a probability of 10^{-9}, thus the effect of pre-emption can be interpreted as increasing the execution time that has a probability of 10^{-9} of being exceeded on each run. Observe that with a memory block size of 1 instruction, after 5 pre-emptions, all instructions are reduced to being misses, whereas with a larger memory block size of 4 instructions, over 25 pre-emptions are required. Note that the final few lines on the graph are vertical, representing a single additional instruction that is altered from always hit to always miss. (Note FAC is a very small program that nevertheless includes loops and conditional statements). For larger programs, a very large number of pre-emptions are required before the pWCET is reduced to the equivalent of all misses, for example, in the case of INSERTSORT, over 500 pre-emptions are required to do this.

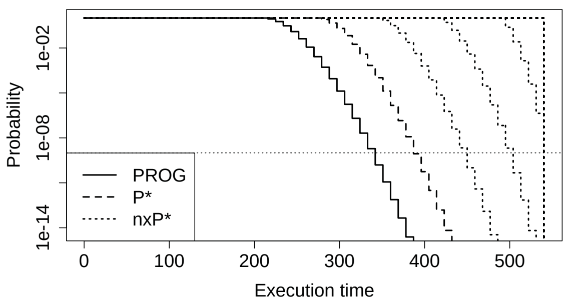


Fig. 4. FAC: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 1**, Cache size $N = 128$, **EVICT-ON-MISS**.

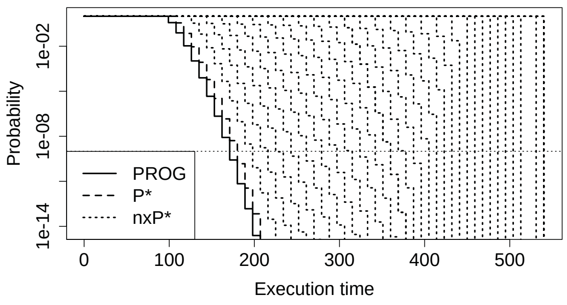


Fig. 5. FAC: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 4**, Cache size $N = 128$, **EVICT-ON-MISS**.

VII. Conclusions and Future Work

Random cache replacement policies have the potential to provide an increase in the level of performance of hard real-time systems that can be guaranteed with respect to an acceptable threshold for the timing failure rate [5]. This is achieved by making the probability of any pathological cases vanishingly small.

The main contribution of this paper is the introduction of integrated probabilistic cache related pre-emption delay (pCRPD) analysis and static probabilistic timing analysis (SPTA) for multi-path programs running on hardware that uses an evict-on-miss random cache replacement policy. The SPTA provides an upper bound on the exceedance function (1-CDF) for the probabilistic worst-case execution time (pWCET) of a program, using only information about the structure of the program, the re-use distances of its instructions, and the size of the cache. The pCRPD analysis determines the maximum effect that pre-emption of the program has on its pWCET. The integration between SPTA and pCRPD updates the pWCET to account for the effects of one or more pre-emptions at arbitrary points. Our analysis is based on a lower bound on the probability of a cache hit for each instruction that is crucially *independent* of the previous history of hits and misses, depending instead only upon the re-use distance. We showed that this lower bound for the evict-on-miss policy dominates that for evict-on-access given in [5] and [6]. We demonstrated the viability of our approach on a number of programs from the Mälardalen benchmark suite [10].

Finally, we note that a number of extensions are possible to our research. In future, we intend to make improvements to the pWCET and pCRPD analysis using techniques such as loop unrolling. Further, we will make comparisons with state-of-the-art deterministic analysis for systems with traditional cache replacement policies. We also intend to fully integrate our approach with schedulability analysis.

References

[1] S. Altmeyer and C. Burguiere. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *Proceedings of the Euromicro Conference on Real-Time Systems, (ECRTS)*, 2009.

[2] S. Altmeyer, R. I. Davis, and C. Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *Proceedings of the Real-Time Systems Symposium, (RTSS)*, 2011.

[3] S. Altmeyer, R. I. Davis, and C. Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5), 2012.

[4] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the Real-Time Technology and Applications Symposium (RTAS)*, 1996.

[5] F. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim. PROARTIS: Probabilistically analysable real-time systems. *ACM Transactions on Embedded Computing Systems (to appear)*, 2013.

[6] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Quinones, and F. J. Cazorla. Measurement-Based Probabilistic Timing Analysis for Multi-path Programs. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[7] R. I. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean. Analysis of probabilistic cache related pre-emption delays. In *University of York, Department of Computer Science, Technical Report YCS-2012-477*, 2012. Available from http://www-users.cs.york.ac.uk/ robdavis/.

[8] J. Díaz, D. Garcia, K. Kim, C. Lee, L. Bello, L. J.M., and O. Mirabella. Stochastic analysis of periodic real-time systems. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, 2002.

[9] M. Gardner and J. Lui. Analyzing stochastic fixed-priority real-time systems. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 44–58, 1999.

[10] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. pages 137–147, Brussels, Belgium, July 2010. OCG.

[11] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6):700–713, June 1998.

[12] J. Lehoczky. Real-time queueing theory. In *Proceedings of the 10th IEEE Real-Time Systems Symposium (RTSS96)*, pages 186–195, 1996.

[13] J. Lopez, J. L. Daz, J. E., and D. Garca. Stochastic analysis of real-time systems under preemptive priority-driven scheduling. *Real-time Systems*, 40(2), 2008.

[14] D. Maxim, M. Houston, L. Santinelli, L. Cucu-Grosjean, and R. Davis. Re-Sampling for Statistical Timing Analysis of Real-Time Systems. In *Proceedings of the International Conference on Real-Time and Network Systems (RTNS)*, 2012.

[15] E. Quinones, E. Berger, G. Bernat, and F. Cazorla. Using Randomized Caches in Probabilistic Real-Time Systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pages 129–138, 2009.

[16] T. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L. Wu, and J. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 1995.

[17] S. Zhou. An efficient simulation algorithm for cache of random replacement policy. In *Proceedings of the International Conference on Network and Parallel Computing (NPC2010)*, pages 144–154, 2010.

[18] H. Zhu, J. Hansen, J. Lehoczky, and R. Rajkumar. Optimal partitioning for quantized EDF scheduling. In *Proceedings of the Real-time Systems Symposium (RTSS02)*, pages 202 – 213, 2002.

VIII. Appendix

A. Monotonicity of $P^{hit}(h)$

Theorem 4 (Monotonicity of $P^{hit}(h)$): The formula for the probability of a hit in the case of the evict-on-miss random cache replacement policy given by

$$P(\text{hit}) = \begin{cases} \min\left\{ \left(\frac{N-h-1}{N-h}\right)^{k-h} \right\} & N > k \\ 0 & \text{otherwise} \end{cases} \quad (27)$$

can be simplified to

$$P(\text{hit}) = \begin{cases} \left(\frac{N-1}{N}\right)^{k} & N > k \\ 0 & \text{otherwise} \end{cases} \quad (28)$$

Proof: We prove the correctness of the simplification by showing that the inner function

$$g(h) = \left(\frac{N-h-1}{N-h}\right)^{k-h} \quad (29)$$

is monotonically increasing in $h \in \{0, \dots N\}$ if $N > k$ and hence, is minimal if $h = 0$. First, we reformulate $g(h)$ by replacing $h = N - x$:

$$f(x) = \left(\frac{x-1}{x}\right)^{x-(N-k)}$$

and show that $f(x)$ is monotonically decreasing in $x \in \{\min(1, N-h), \dots, N\}$ if $N > k$, i.e.,

$$\forall x : f(x) \geq f(x+1)$$

$$\begin{aligned}
\forall x :& f(x) \geq f(x+1) \\
\Leftrightarrow& \left(\frac{x-1}{x}\right)^{x-(N-k)} \geq \left(\frac{x}{x+1}\right)^{x+1-(N-k)} \\
\Leftrightarrow& \left(\frac{x-1}{x}\right)^{x-(N-k)} \left(\frac{x+1}{x}\right)^{x-(N-k)} \geq \frac{x}{x+1}, \forall x \geq 1 \\
\Leftrightarrow& \left(\frac{(x-1)(x+1)}{x^2}\right)^{x-(N-k)} \geq \frac{x}{x+1}, \forall x \geq 1 \\
\Leftrightarrow& \left(\frac{x^2-1}{x^2}\right)^{x-(N-k)} \geq \frac{x}{x+1}, \forall x \geq 1 \text{ and } N > k \\
\Leftarrow& \left(\frac{x^2-1}{x^2}\right)^{x-1} \geq \frac{x}{x+1}, \forall x \geq 1 \\
\Leftrightarrow& \left(\frac{x^2-1}{x^2}\right)^{x} \geq \frac{x(x^2-1)}{(x+1)x^2}, \forall x \geq 1 \\
\Leftrightarrow& \left(\frac{x^2-1}{x^2}\right)^{x} \geq \frac{x-1}{x}, \forall x \geq 1 \\
\Leftrightarrow& (x^2-1)^x \geq x^{2x} - x^{2x-1}, \forall x \geq 1 \\
\Leftrightarrow& \sum_{i=0}^{x} \binom{x}{i} x^{2(x-i)} (-1)^i \geq x^{2x} - x^{2x-1}, \forall x \geq 1 \\
\Leftrightarrow& x^{2x} - x \cdot x^{2(x-1)} + \sum_{i=2}^{x} \binom{x}{i} x^{2(x-i)} (-1)^i \\
& \geq x^{2x} - x^{2x-1}, \forall x \geq 1 \\
\Leftrightarrow& \sum_{i=2}^{x} \binom{x}{i} x^{2(x-i)} (-1)^i \geq 0, \forall x \geq 1
\end{aligned}$$

To prove the last inequality, we show that the sum of two succeeding terms starting with an even i is positive $\forall x \geq 1$:

$$\begin{aligned}
& \binom{x}{i} x^{2(x-i)} (-1)^i + \binom{x}{i+1} x^{2(x-(i+1))} (-1)^{(i+1)} \geq 0 \\
\Leftrightarrow& \binom{x}{i} x^{2(x-i)} - \binom{x}{i+1} x^{2(x-i-1)} \geq 0 \\
\Leftrightarrow& \binom{x}{i} x^{2x-2i} \geq \binom{x}{i+1} x^{2x-2i-2} \\
\Leftrightarrow& \binom{x}{i} \geq \binom{x}{i+1} x^{-2} \\
\Leftrightarrow& \binom{x}{i} \geq \frac{x!(i+1)}{(i)!(x-i-1)!x^2} \\
\Leftrightarrow& \binom{x}{i} \geq \frac{x!i}{(i)!(x-i)!x^2} \\
\Leftrightarrow& \binom{x}{i} \geq \binom{x}{i} \frac{i(x-i)}{x^2} \\
\Leftrightarrow& 1 \geq \frac{i(x-i)}{x^2} \\
\Leftrightarrow& x^2 \geq i(x-i)
\end{aligned}$$

which holds since $x \geq i$.

Note that there is one additional positive term if x is even. ■

B. Additional Experimental Evaluation

In this section, we give the results of an experimental evaluation, applying our multi-path SPTA and pCRPD analysis to the BS, FAC, FDCT, FIBCALL, FIR, JFDCTINT, and INSERTSORT benchmarks from the Mälardalen benchmark suite [10]. For each benchmark, we conducted a series of experiments determining an upper bound pWCET distribution (1-CDF) for the program, for the evict-on-miss and the evict-on-access random cache replacement policies. We carried out 4 experiments on each benchmark:

1) Varying the number of pre-emptions: 0, 1, 2, 3, and so on, for a memory block size of 1 and a cache size of $N = 128$ blocks (512 bytes).
2) Varying the number of pre-emptions: 0, 1, 2, 3, and so on, for a memory block size of 4 and a cache size of $N = 128$ blocks (2048 bytes).
3) Varying the memory block size, corresponding to 1, 2, 4, and 8 instructions (i.e. 4, 8, 16, 32 bytes, given that instructions are 4 bytes), with the cache size corresponding to $N = 128$ blocks (so 512, 1024, 2048, and 4096 bytes respectively).
4) Varying the memory block size, corresponding to 1, 2, 4, and 8 instructions (i.e. 4, 8, 16, 32 bytes, given that instructions are 4 bytes), with the cache size fixed at 1024 bytes (so $N = 256$, $N = 128$, $N = 64$, and $N = 32$ respectively).

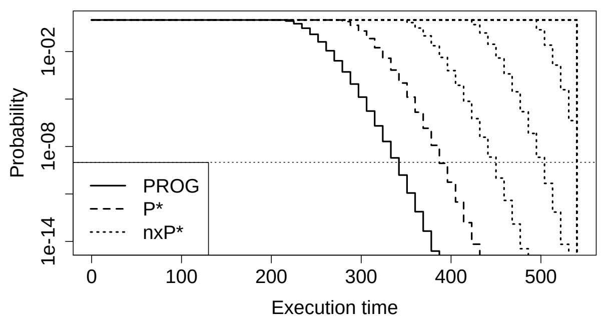


Fig. 6. FAC: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 1**, Cache size $N = 128$, **EVICT-ON-MISS**.

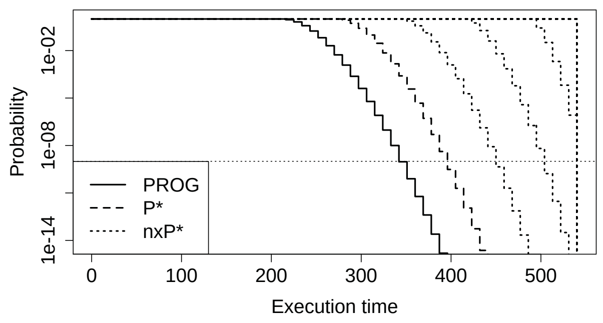


Fig. 7. FAC: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 1**, Cache size $N = 128$, **EVICT-ON-ACCESS**.

In our first experiment, we compared the upper bound pWCET distribution of the program for the evict-on-miss and evict-on-access random cache replacement policies assuming 0, 1, 2, 3, and so on arbitrary pre-emptions. Figures 6 and 7 illustrate these results for the FAC benchmark, assuming a memory block size of 4 bytes (1 instruction) and a cache size of 512 bytes ($N = 128$). Comparing the two graphs, we observe that evict-on-miss has marginally better performance than evict-on-access in each case. Note that after 5 pre-emptions, the analysis indicates that all of the instructions could become misses, hence the final vertical line at the right hand side of both graphs. (We note that FAC is a very small program that nevertheless contains loops and conditional statements).

Our second experiment was effectively a repeat of the first, but this time with a memory block size of 4 instructions (and a commensurately larger cache of 2048 bytes (i.e again $N = 128$ blocks). Figures 8 and 9 illustrate these results for the FAC benchmark for the evict-on-miss and evict-on-access policies respectively. We observe that the pWCETs are significantly reduced compared to the same experiments with a memory block size of 1 instruction. In this case, both the evict-on-miss and the evict-on-access policies benefit from the increased memory block size, as instructions that were previously in different memory blocks now share the same memory block

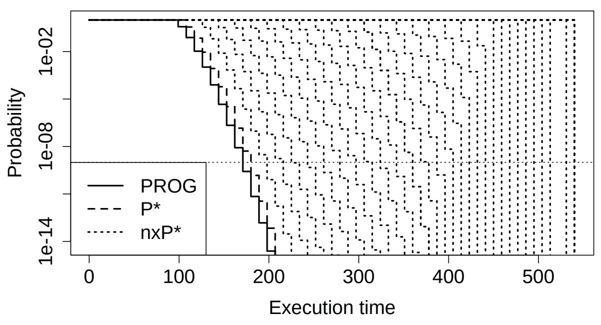


Fig. 8. FAC: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 4**, Cache size $N = 128$, **EVICT-ON-MISS**.

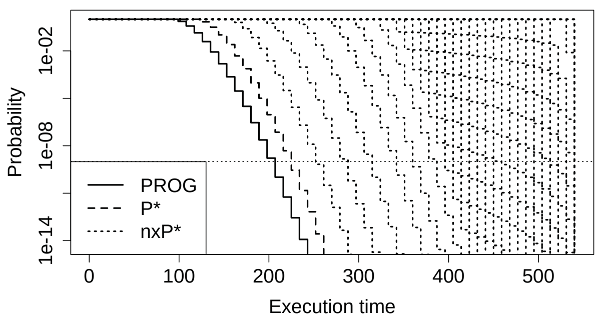


Fig. 9. FAC: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 4**, Cache size $N = 128$, **EVICT-ON-ACCESS**.

which reduces the re-use distances. Further, the performance of evict-on-miss is now significantly better than that of evict-on-access. This is due to the large number of instructions that have a re-use distance of zero with evict-on-miss, and so do not increase the re-use distances of subsequent instructions. Further, now substantially more pre-emptions are required to reduce all of the instructions to misses. This is because an individual pre-emption is needed to reduce each zero re-use distance instruction to a miss. (This can be seen in the spacing of the lines towards the right hand side of each graph which are 9 time units apart; the difference between a cache hit and a cache miss).

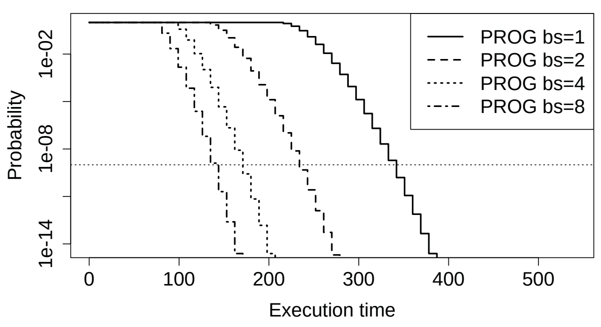


Fig. 10. FAC: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 128 blocks**, with a probability threshold at 10^{-9}. **EVICT-ON-MISS** random cache replacement policy.

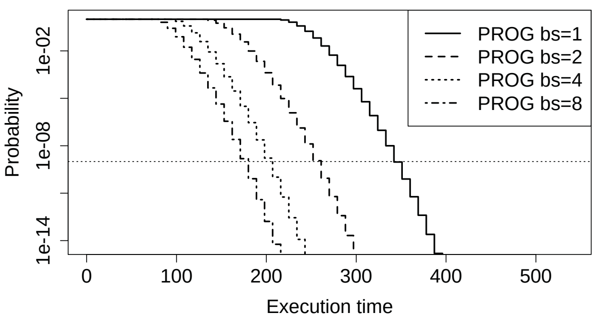


Fig. 11. FAC: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of N = 128 blocks**, with a probability threshold at 10^{-9}. **EVICT-ON-ACCESS** random cache replacement policy.

In our third experiment, we compared the upper bound pWCET distribution of the program without pre-emption, assuming memory block sizes of 1, 2, 4, and 8 instructions, with the cache size corresponding to $N = 128$ blocks (so 512, 1024, 2048, and 4096 bytes respectively, given that instructions are 4 bytes). Figures 10 and 11 illustrate these results for the FAC benchmark, for the evict-on-miss and evict-on-access random cache replacement policies respectively. We observe that as expected, increasing the memory block size while also increasing the cache size so that it is constant in terms of the number of memory blocks significantly improves the pWCET of the program, with the best performance obtained for memory blocks of size 8. We note that as the memory block size increases, so the advantage of evict-on-miss over evict-on-access also increases.

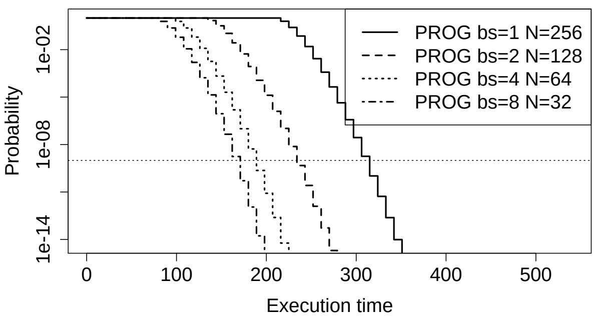


Fig. 12. FAC: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of N = 256, N = 128, N = 64, and N = 32 blocks** respectively (1024 bytes), with a probability threshold at 10^{-9}. **EVICT-ON-MISS** random cache replacement policy.

Our fourth experiment was similar to the third; however, this time we examined how memory block size (1,2,4, or 8 instructions) affects performance when the size of the cache is fixed, in this case at 1024 bytes (i.e. $N = 256$, $N = 128$, $N = 64$, and $N = 32$ blocks respectively). Figures 12 and 13 illustrate these results for the FAC benchmark, for the evict-on-miss and evict-on-access random cache replacement policies respectively. We observe that for evict-on-miss, increasing the block size still increases performance even though the total number of blocks in the cache is reduced to just 32 in the case of memory blocks of size 8 (this is partly because the FAC program is very small); however, in the case of evict-on-access, a memory block size of 4 instructions and a cache size of 64 blocks gives the best performance. Effectively a larger cache is needed to give better performance with the larger re-use distances of evict-on-access.

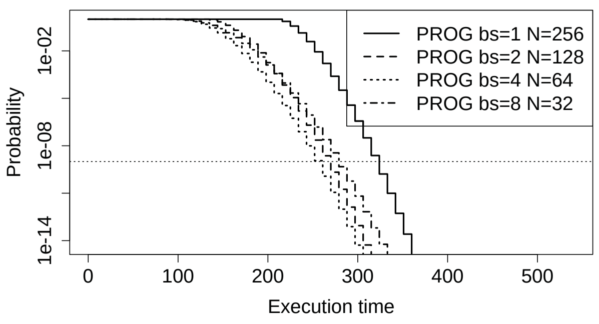


Fig. 13. FAC: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of N = 256, N = 128, N = 64, and N = 32 blocks** respectively (1024 bytes), with a probability threshold at 10^{-9}. **EVICT-ON-ACCESS** random cache replacement policy.

The remaining pages of this technical report show the results of our experiments for the BS, FDCT, FIBCALL, FIR, JFDCTINT, and INSERTSORT benchmarks. In these graphs, we present the results for up to a maximum of 10 pre-emptions. We note that the results for the evict-on-access policy and FDCT show sharp transitions for cache sizes $N \leq 128$. This is because FDCT contains loops with more than 128 instructions and once the re-use distance of an instruction exceeds the size of the cache, the probability of a hit is assumed to be zero.

Revision

This is a revised and updated version of this technical report with additional experimental data, published in February 2013. The original version was published in October 2012.

Acknowledgements

This work was partially funded by UK EPSRC Tempo project (EP/G055548/1), and the Transregional Collaborative Research Center AVACS of the German Research Council (DFG).

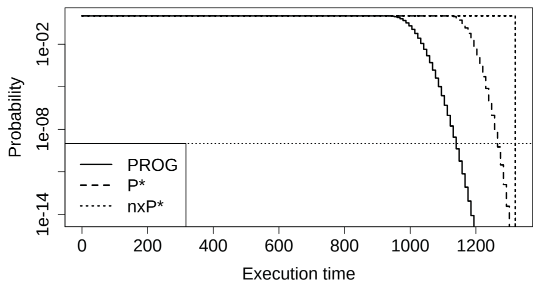


Fig. 14. BS: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 1**, Cache size $N = 128$, **EVICT-ON-MISS**.

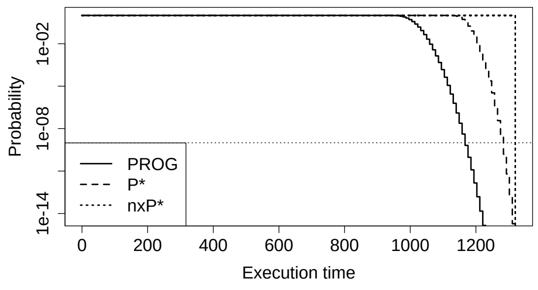


Fig. 15. BS: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 1**, Cache size $N = 128$, **EVICT-ON-ACCESS**.

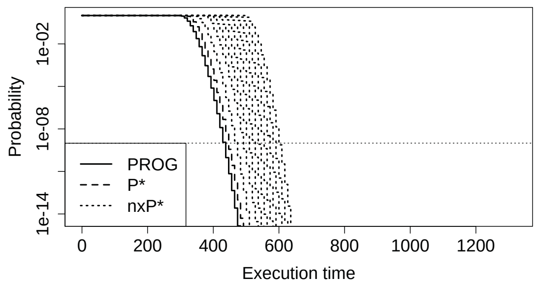


Fig. 16. BS: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 4**, Cache size $N = 128$, **EVICT-ON-MISS**.

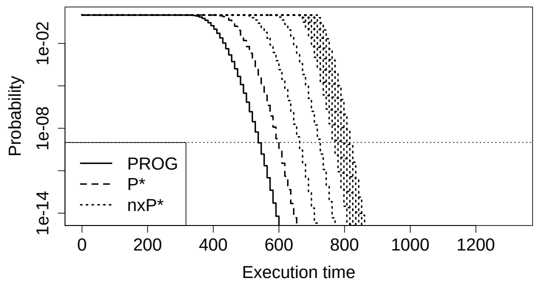


Fig. 17. BS: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 4**, Cache size $N = 128$, **EVICT-ON-ACCESS**.

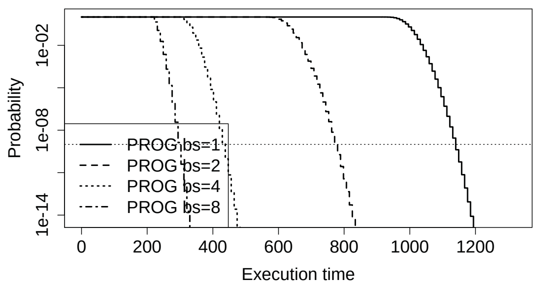


Fig. 18. BS: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 128 blocks**, with a probability threshold at 10^{-9}. **EVICT-ON-MISS**.

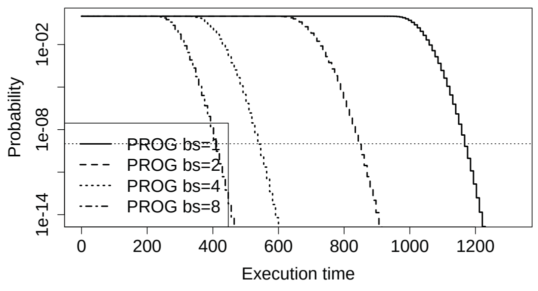


Fig. 19. BS: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 128 blocks**, with a probability threshold at 10^{-9}. **EVICT-ON-ACCESS**.

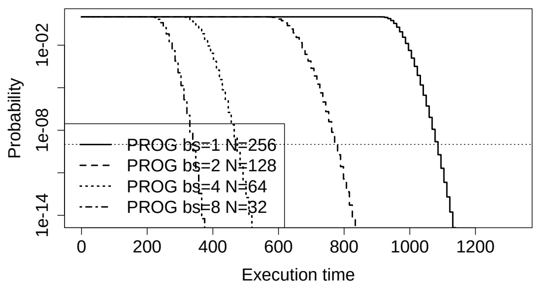


Fig. 20. BS: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 256, *N* = 128, *N* = 64, and *N* = 32 blocks** respectively (1024 bytes), with a probability threshold at 10^{-9}. **EVICT-ON-MISS**.

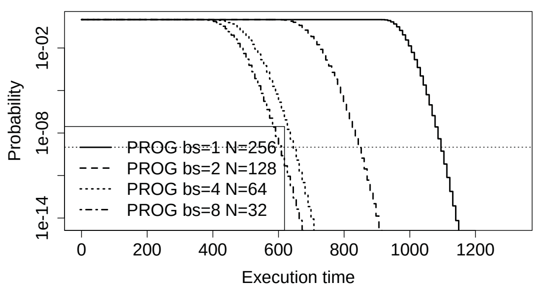


Fig. 21. BS: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 256, *N* = 128, *N* = 64, and *N* = 32 blocks** respectively (1024 bytes), with a probability threshold at 10^{-9}. **EVICT-ON-ACCESS**.

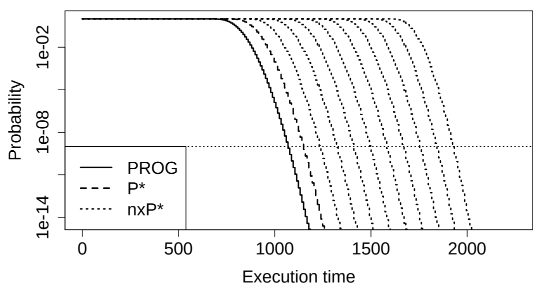


Fig. 22. FIBCALL: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 1**, Cache size $N = 128$, **EVICT-ON-MISS**.

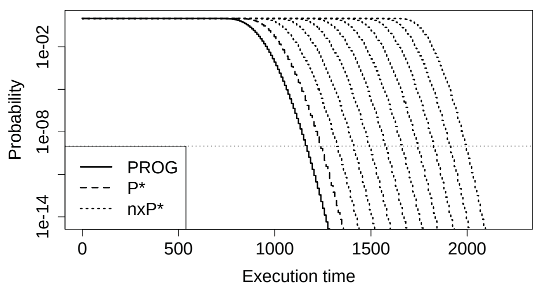


Fig. 23. FIBCALL: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 1**, Cache size $N = 128$, **EVICT-ON-ACCESS**.

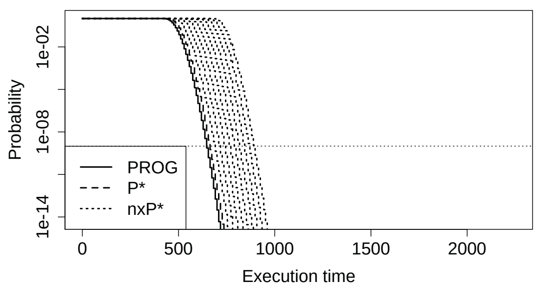


Fig. 24. FIBCALL: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 4**, Cache size $N = 128$, **EVICT-ON-MISS**.

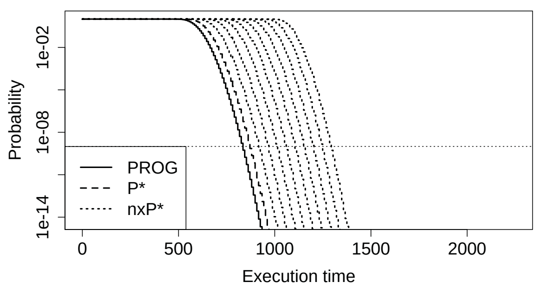


Fig. 25. FIBCALL: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 4**, Cache size $N = 128$, **EVICT-ON-ACCESS**.

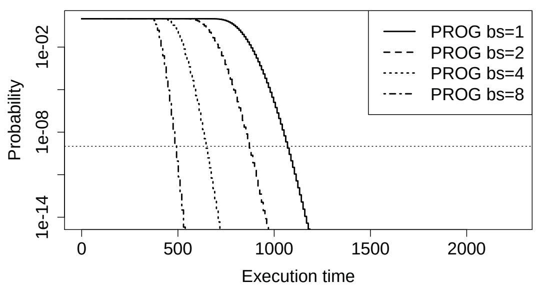


Fig. 26. FIBCALL: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 128 blocks**, with a probability threshold at 10^{-9}. **EVICT-ON-MISS**.

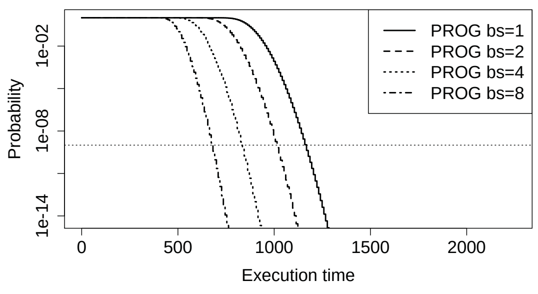


Fig. 27. FIBCALL: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 128 blocks**, with a probability threshold at 10^{-9}. **EVICT-ON-ACCESS**.

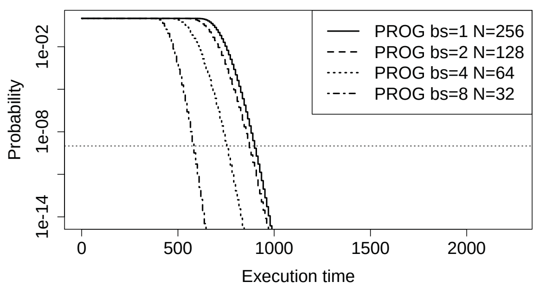


Fig. 28. FIBCALL: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 256, *N* = 128, *N* = 64, and *N* = 32 blocks** respectively (1024 bytes), with a probability threshold at 10^{-9}. **EVICT-ON-MISS**.

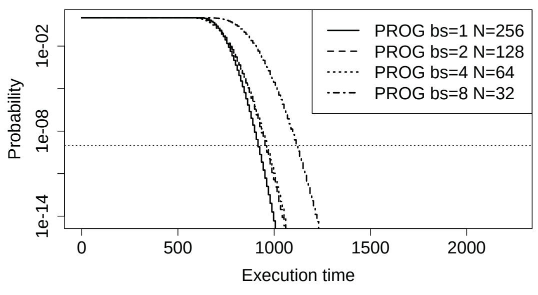


Fig. 29. FIBCALL: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 256, *N* = 128, *N* = 64, and *N* = 32 blocks** respectively (1024 bytes), with a probability threshold at 10^{-9}. **EVICT-ON-ACCESS**.

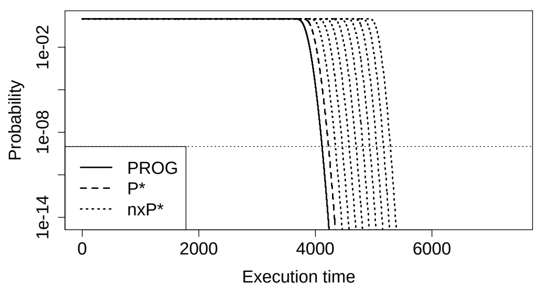


Fig. 30. INSERTSORT: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 1**, Cache size $N = 128$, **EVICT-ON-MISS**.

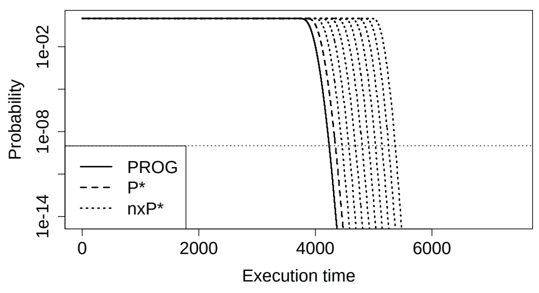


Fig. 31. INSERTSORT: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 1**, Cache size $N = 128$, **EVICT-ON-ACCESS**.

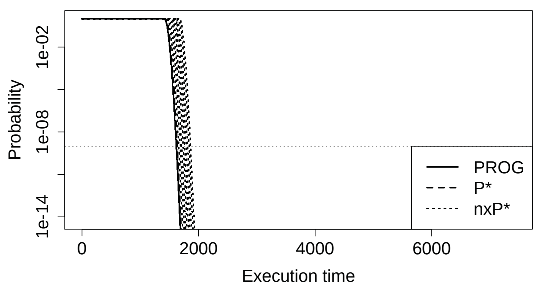


Fig. 32. INSERTSORT: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 4**, Cache size $N = 128$, **EVICT-ON-MISS**.

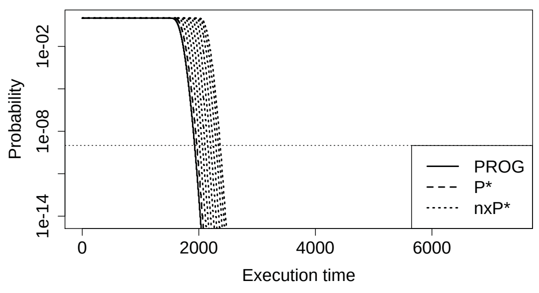


Fig. 33. INSERTSORT: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 4**, Cache size $N = 128$, **EVICT-ON-ACCESS**.

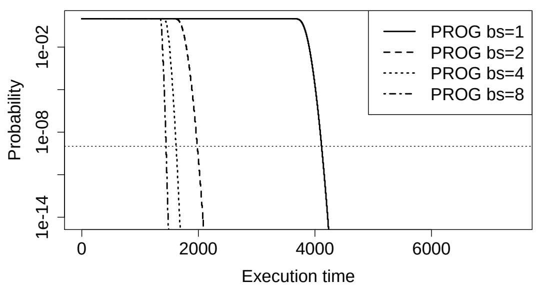


Fig. 34. INSERTSORT: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 128 blocks**, with a probability threshold at 10^{-9}. **EVICT-ON-MISS**.

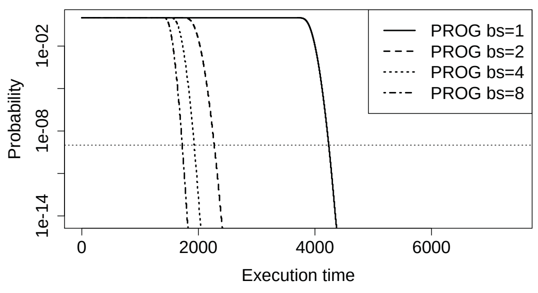


Fig. 35. INSERTSORT: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 128 blocks**, with a probability threshold at 10^{-9}. **EVICT-ON-ACCESS**.

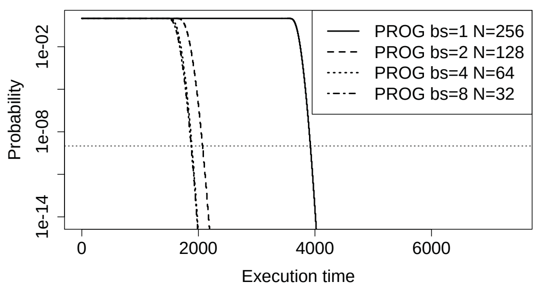


Fig. 36. INSERTSORT: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 256, *N* = 128, *N* = 64, and *N* = 32 blocks** respectively (1024 bytes), with a probability threshold at 10^{-9}. **EVICT-ON-MISS**.

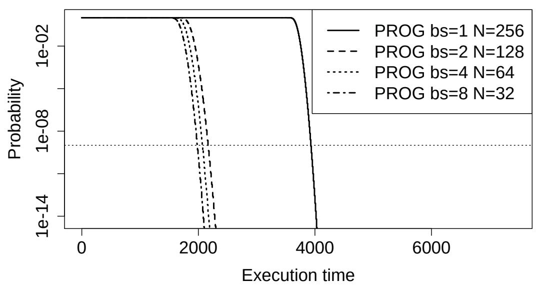


Fig. 37. INSERTSORT: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 256, *N* = 128, *N* = 64, and *N* = 32 blocks** respectively (1024 bytes), with a probability threshold at 10^{-9}. **EVICT-ON-ACCESS**.

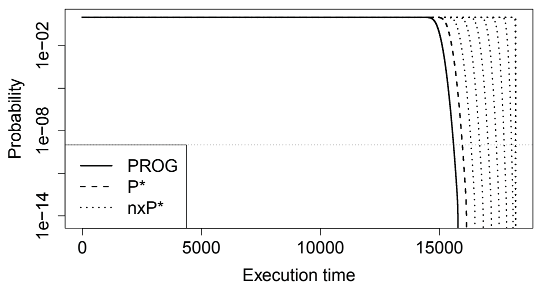


Fig. 38. FDCT: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 1**, Cache size $N = 128$, **EVICT-ON-MISS**.

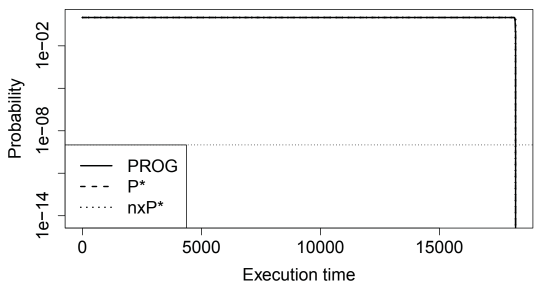


Fig. 39. FDCT: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 1**, Cache size $N = 128$, **EVICT-ON-ACCESS**.

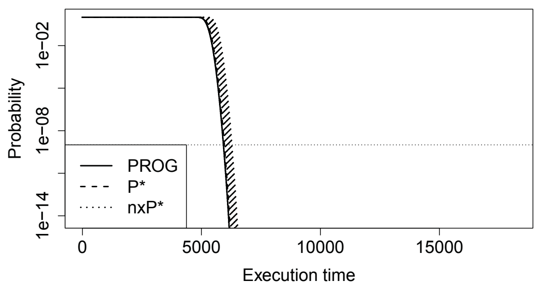


Fig. 40. FDCT: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 4**, Cache size $N = 128$, **EVICT-ON-MISS**.

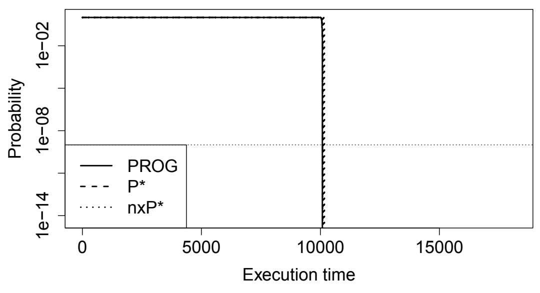


Fig. 41. FDCT: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 4**, Cache size $N = 128$, **EVICT-ON-ACCESS**.

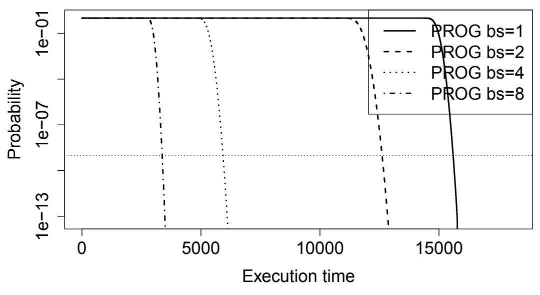


Fig. 42. FDCT: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 128 blocks**, with a probability threshold at 10^{-9}. **EVICT-ON-MISS**.

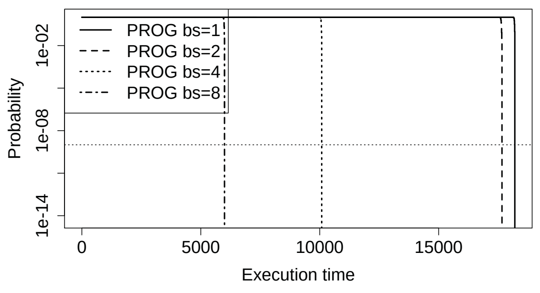


Fig. 43. FDCT: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 128 blocks**, with a probability threshold at 10^{-9}. **EVICT-ON-ACCESS**.

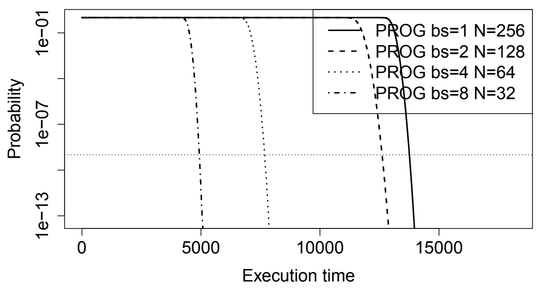


Fig. 44. FDCT: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 256, *N* = 128, *N* = 64, and *N* = 32 blocks** respectively (1024 bytes), with a probability threshold at 10^{-9}. **EVICT-ON-MISS**.

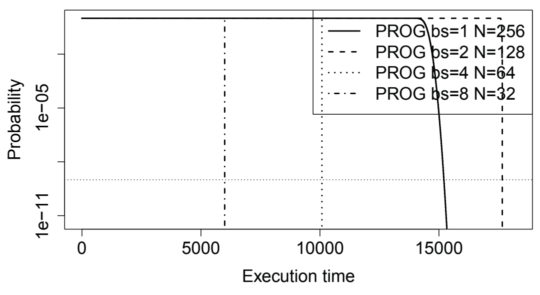


Fig. 45. FDCT: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 256, *N* = 128, *N* = 64, and *N* = 32 blocks** respectively (1024 bytes), with a probability threshold at 10^{-9}. **EVICT-ON-ACCESS**.

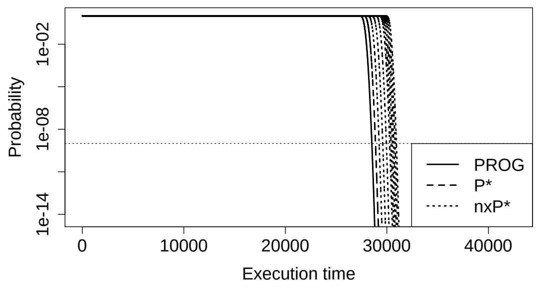


Fig. 46. FIR: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 1**, Cache size $N = 128$, **EVICT-ON-MISS**.

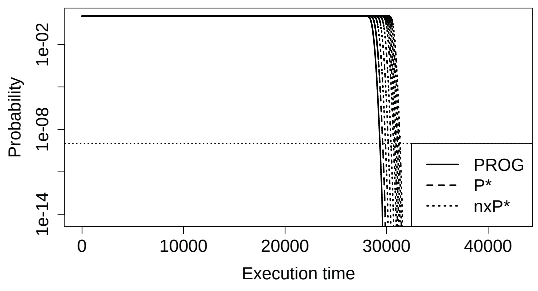


Fig. 47. FIR: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 1**, Cache size $N = 128$, **EVICT-ON-ACCESS**.

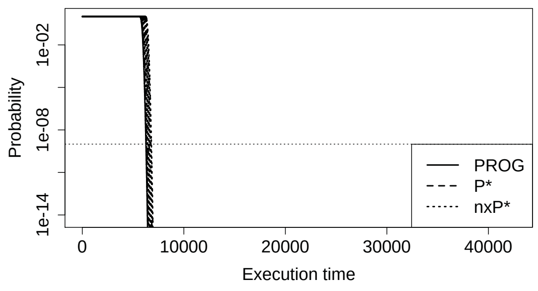


Fig. 48. FIR: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 4**, Cache size $N = 128$, **EVICT-ON-MISS**.

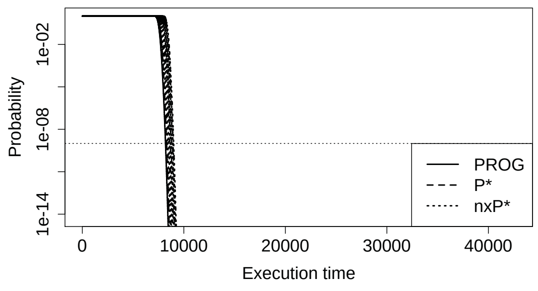


Fig. 49. FIR: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 4**, Cache size $N = 128$, **EVICT-ON-ACCESS**.

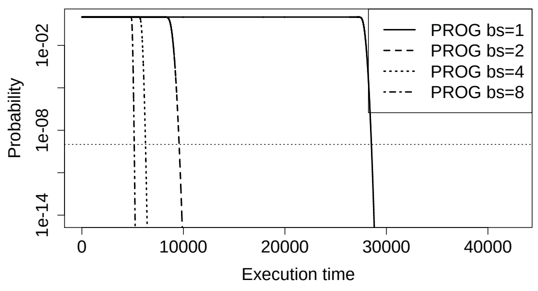


Fig. 50. FIR: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of N = 128 blocks**, with a probability threshold at 10^{-9}. **EVICT-ON-MISS**.

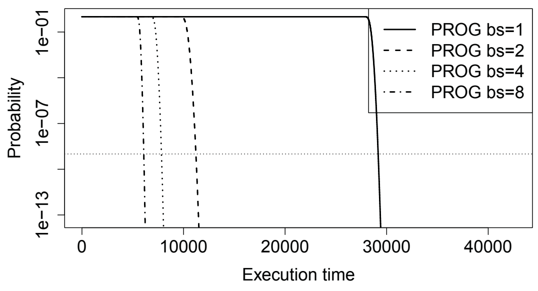


Fig. 51. FIR: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of N = 128 blocks**, with a probability threshold at 10^{-9}. **EVICT-ON-ACCESS**.

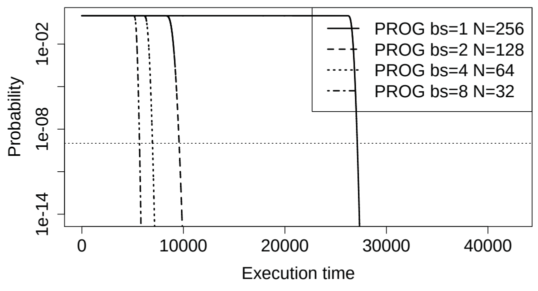


Fig. 52. FIR: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of N = 256, N = 128, N = 64, and N = 32 blocks** respectively (1024 bytes), with a probability threshold at 10^{-9}. **EVICT-ON-MISS**.

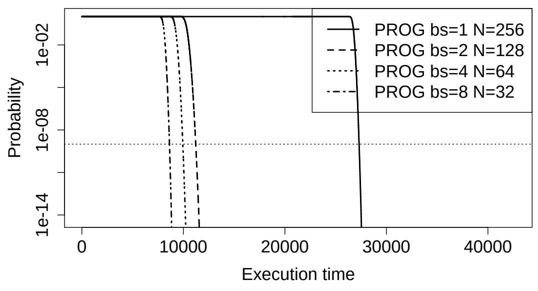


Fig. 53. FIR: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of N = 256, N = 128, N = 64, and N = 32 blocks** respectively (1024 bytes), with a probability threshold at 10^{-9}. **EVICT-ON-ACCESS**.

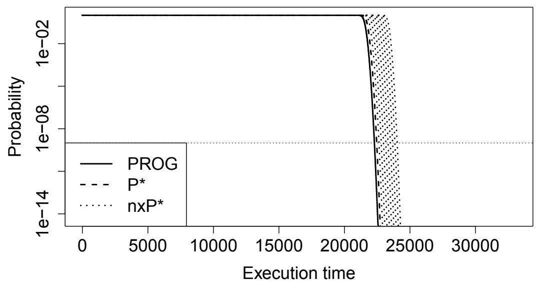


Fig. 54. JFDCTINT: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 1**, Cache size $N = 128$, **EVICT-ON-MISS**.

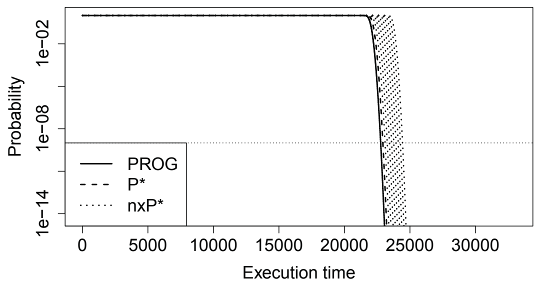


Fig. 55. JFDCTINT: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 1**, Cache size $N = 128$, **EVICT-ON-ACCESS**.

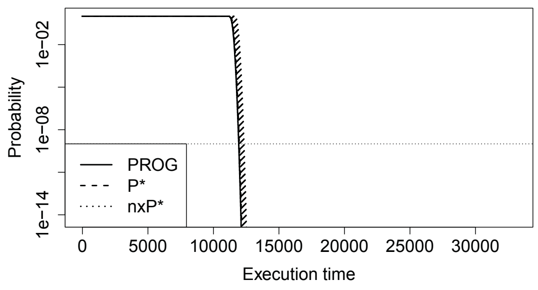


Fig. 56. JFDCTINT: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 4**, Cache size $N = 128$, **EVICT-ON-MISS**.

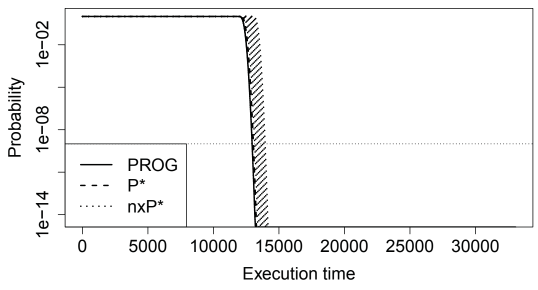


Fig. 57. JFDCTINT: 1-CDF of the program with no pre-emption (PROG), 1 pre-emption (P^*), and multiple pre-emptions ($n \times P^*$), with a probability threshold at 10^{-9}. **Memory block size = 4**, Cache size $N = 128$, **EVICT-ON-ACCESS**.

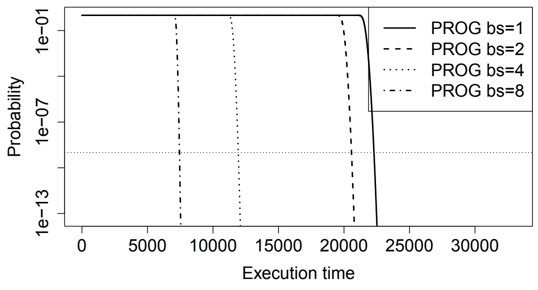


Fig. 58. JFDCTINT: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 128 blocks**, with a probability threshold at 10^{-9}. **EVICT-ON-MISS**.

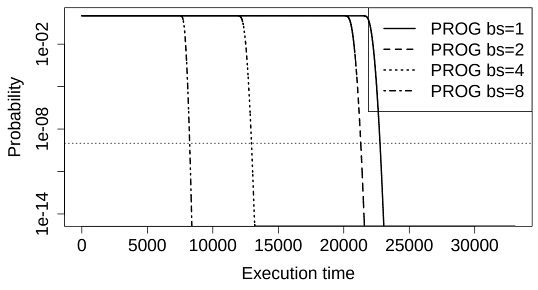


Fig. 59. JFDCTINT: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 128 blocks**, with a probability threshold at 10^{-9}. **EVICT-ON-ACCESS**.

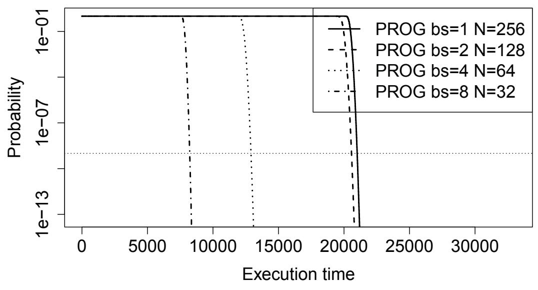


Fig. 60. JFDCTINT: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 256, *N* = 128, *N* = 64, and *N* = 32 blocks** respectively (1024 bytes), with a probability threshold at 10^{-9}. **EVICT-ON-MISS**.

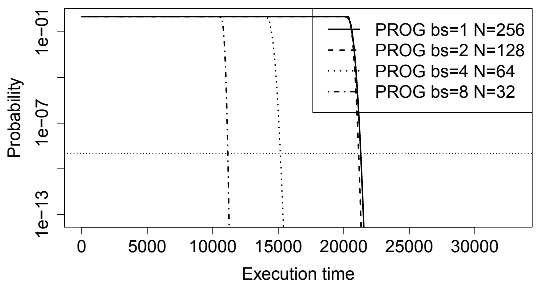


Fig. 61. JFDCTINT: 1-CDF of the program with memory blocks of size 1,2,4, and 8, and a **cache of *N* = 256, *N* = 128, *N* = 64, and *N* = 32 blocks** respectively (1024 bytes), with a probability threshold at 10^{-9}. **EVICT-ON-ACCESS**.