Static probabilistic timing analysis for multi-path programs

Benjamin Lesage, David Griffin, Sebastian Altmeyer, Robert I. Davis

Angaben zur Veröffentlichung / Publication details:

Lesage, Benjamin, David Griffin, Sebastian Altmeyer, and Robert I. Davis. 2015. "Static probabilistic timing analysis for multi-path programs." In *2015 IEEE Real-Time Systems Symposium, 1-4 December 2015, San Antonio, TX, USA*, edited by Chris Gill and Marco Caccamo, 361–72. Piscataway, NJ: IEEE. https://doi.org/10.1109/rtss.2015.41.

Static Probabilistic Timing Analysis for Multi-path Programs

Benjamin Lesage
University of York
York, UK
benjamin.lesage@york.ac.uk

David Griffin
University of York
York, UK
david.griffin@york.ac.uk

Sebastian Altmeyer
Université du Luxembourg
Luxembourg, Luxembourg
sebastian.altmeyer@uni.lu

Robert I. Davis
University of York, UK
Inria Paris, France
rob.davis@york.ac.uk

Abstract—**This paper introduces an effective Static Probabilistic Timing Analysis (SPTA) for multi-path programs. The analysis estimates the temporal contribution of an evict-on-miss, random replacement cache to the probabilistic Worst-Case Execution Time (pWCET) distribution of multi-path programs. The analysis uses a conservative join function that provides a proper over-approximation of the possible cache contents and the pWCET distribution on path convergence, irrespective of the actual path followed during execution. Simple program transformations are introduced that reduce the impact of path indeterminism while ensuring sound pWCET estimates. Evaluation shows that the proposed method is efficient at capturing locality in the cache, and substantially outperforms the only prior approach to SPTA for multi-path programs based on path merging. The evaluation results show incomparability with analysis for an equivalent deterministic system using an LRU cache.**

I. Introduction

Real-time systems such as those deployed in space, aerospace, automotive and railway applications require guarantees that the probability of the system failing to meet its timing constraints is below an acceptable threshold (e.g. a failure rate of less than 10^{-9} per hour for some aerospace and automotive applications). 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. One promising approach is to use a cache with a random replacement policy combined with probabilistic analysis. With this approach the probability of pathological worst-cases behaviours can be upper bounded at quantifiably extremely low levels, for example well below the maximum permissible failure rate (e.g. 10^{-9} per hour) for the system.

The timing behaviour of programs running on a processor with a cache using a random replacement policy can be determined using Static Probabilistic Timing Analysis (SPTA). SPTA computes an upper bound on the probabilistic Worst-Case Execution Time (pWCET) in terms of an exceedance function. This exceedance function gives the probability, as a function of all possible values for an execution time budget x, that the execution time of the program will exceed that budget on any single run. The reader is referred to [1] for examples of pWCET distributions, and to [2] for a detailed discussion of what is meant by a pWCET distribution.

This paper introduces an effective SPTA for *multi-path* programs running on hardware that uses an evict-on-miss, random replacement cache. Prior work on SPTA for multi-path programs by Davis et al. [1] in 2013 used a simple path merging approach to compute cache hit probabilities based on reuse distances. The analysis derived in this paper builds upon more sophisticated SPTA techniques for the analysis of single path programs given by Altmeyer and Davis [3] in 2014 (see also [4]). This new analysis provides substantially improved results compared to the path merging approach.

A. Related Work

Probabilistic timing analyses can be categorised into three different classes: (i) measurement-based [5], (ii) hybrid [6], and (iii) static. In this paper, we focus on static techniques.

Static probabilistic timing analyses derive the pWCET distribution for a program by analysing the structure of the program and modeling the behaviour of the hardware it runs on. Existing work on SPTA has primarily focussed on randomized architectures containing caches with random replacement policies. Initial results for the evict-on-miss [7] and evict-on-access [5], [8] policies were derived for single-path programs. These methods use the *reuse distance* of each access to determine its probability of being a cache hit. These results were superseded by later work by Davis et al. [1] who derived an optimal lower bound on the probability of a cache hit under the evict-on-miss policy, and showed that evict-on-miss dominates evict-on-access. Altmeyer and Davis [3] proved the correctness of the lower bound derived in [1], and its optimality with regards to the limited information that it uses (i.e. the reuse distance). They also showed that the probability functions previously given in [9] and [7] are unsound (optimistic) for use in SPTA.

In 2013, a simple SPTA for multipath programs was introduced by Davis et al. [1], based on path merging. With this method, accesses are represented by their reuse distances. The program is then virtually reduced to a single sequence which upper-bounds all possible paths with regards to the reuse distance of their accesses.

In 2014, more sophisticated SPTA methods for single path programs were derived by Altmeyer and Davis [3]. They introduced the notion of cache contention, which combined with reuse distance enables the computation of a more precise bound on the probability that a given access is a cache hit. Altmeyer and Davis [3] also introduced a significantly more effective method based on combining exhaustive evaluation of the cache behaviour with cache contention. Exhaustive state enumeration is only performed on a limited number of pre-selected *focused* accesses. This method provides an effective trade-off between analysis precision and tractability. In this paper, we build upon this state-of-the-art approach, extending it to multi-path programs.

Recently, Reineke [10] observed that SPTA based on reuse distances [1] results, by construction, in less precise bounds than existing analyses based on stack distance for an equivalent system with an LRU cache [11]. However, this does not necessarily hold for the more sophisticated SPTA based on cache contention and collecting semantics given by Altmeyer and Davis [3] in 2014. Deterministic analyses for LRU are incomparable with these analyses for random replacement caches. This is illustrated by our evaluation results. It can also be seen by considering simple examples such as a repeated sequence of accesses to five memory blocks $\langle a, b, c, d, e, a, b, c, d, e\rangle$ with a four-way associative cache. With LRU, no hits can be predicted. By contrast, with a random replacement cache, and SPTA based on cache contention, four out of the last five accesses can be assumed to have a non-zero probability of being a cache hit (as shown in table 1 of [3]), hence SPTA for a random replacement cache outperforms analysis of LRU in this case. We note that in spite of recent efforts [12] the stateless random replacement policies have lower silicon costs than LRU, and so can potentially provide improved real-time performance at lower hardware cost.

Early work [13], [14] in the domain of SPTA for *deterministic* architectures relied for its correctness on knowledge of the probability that a specific path would be taken or that specific input data would be encountered; however, in general such information may not be available. As an example, a car is very unlikely to spend much of its operating life at both maximum rpm and maximum speed, yet it can potentially do so for protracted periods of time, during which failure of the engine management system is unacceptable. The analysis given in this paper does not require any assumption about the probability distribution of different paths or inputs. It relies only on the *random* selection of cache lines for replacement.

B. Organisation

In this paper, we introduce a set of methods that are required for the application of SPTA to multi-path programs. Section II recaps the assumptions and methods which we build upon. These were used in previous work [3] to upper-bound the pWCET distribution of a trace of a single path program. We address the issue of multi-path programs in the context of SPTA in Section III. This includes the definition of conservative (over-approximate) join functions to collect information regarding cache contention, possible cache contents, and the pWCET distribution at each program point, irrespective of the path followed during execution. Section IV introduces simple program transformations which improve the precision of the analysis while ensuing that the pWCET distribution of the transformed program remains sound (i.e. upper-bounds that of the original). Multi-path SPTA is applied to a selection of benchmarks in Section V and the precision and run-time of the different approaches compared. Section VI concludes with a summary of the main contributions of the paper and a discussion of future work.

II. Static probabilistic timing analysis

In this section, we recap on state-of-the-art SPTA techniques for single path programs [3]. We first give an overview of the system model assumed throughout the paper. The pertinence of the model is discussed at the end of this section.

We assume a time-composable architecture, without timing anomalies [15], i.e. that local worst-cases (a miss in the context of the cache) add up to the global worst-case. This enables analysis of the impact of the cache in isolation from other architectural features.

A. Cache model

We assume a single level, private, N-way fully associative cache with an evict-on-miss random replacement policy. On an access, should the requested memory block be absent from the cache then the contents of a randomly selected cache line are evicted. The requested memory block is then loaded into the selected location. Given that there are N ways, the probability of any given cache line being selected by the replacement policy is $\frac{1}{N}$. We assume a fixed upper-bound on the hit and miss latencies, denoted by $\mathcal{H}$ and $\mathcal{M}$ respectively. (We note that the restriction to a fully associative cache can be easily lifted for a set-associative cache through the analysis of each cache set as an independent fully-associative cache).

B. Traces and pWCET

We now recap on the existing SPTA method [3] for evaluating the pWCET of a trace using the notion of cache contention. A trace T is defined as an ordered sequence of n memory blocks $[e_1, ..., e_n]$, such that $e_i = e_j$ if access i and j target the same memory block. Each element in a sequence has a probability of being a cache hit $P(e_i^{hit})$, and of being a cache miss $P(e_i^{miss}) = 1 - P(e_i^{hit})$.

The reuse distance $rd(e, T)$ of element e in trace T is the maximum number of possible evictions since the last access to the same block. Should there be no such prior access to the same block, the reuse distance is defined as ∞. Given the set of all traces $\mathbb{T}$ and of all elements $\mathbb{E}$, the reuse distance is defined as:

$$rd(e_i, [e_1, ..., e_{i-1}]) = \begin{cases} i - j - 1 & if\ \exists e_j : e_i = e_j, \\ & \forall k : j < k < i, e_i \neq e_k \\ \infty & otherwise \end{cases} \tag{1}$$

The probability of e_i being a hit is set to 0 if there are more blocks since the last access to the same block that contend for cache space than the N available lines. This is captured by the cache contention $con(e_i, T)$ [3] of element e_i in trace T. The definition of $\hat{P}(e_i^{hit})$ which denotes a lower bound on the actual probability $P(e_i^{hit})$ of a cache hit is as follows:

$$\hat{P}(e_i^{hit}) = \begin{cases} 0 & con(e_i, T) \geq N \\ \left(\frac{N-1}{N}\right)^{rd(e_i,T)} & otherwise \end{cases} \tag{2}$$

The cache contention $con(e_i, T)$ includes all potential hits since the last access to the same block as e_i, since we assume each access that is a hit may occupy a separate location in the cache. Contention depends on and contributes to the potential hits captured by $\hat{P}(e_j^{hit}), j < i$, and is computed from the first accesses, where $rd(e_i, T) = \infty$, to the last. The contention also accounts for the first access e_r which follows the previous access to the same memory block as e_i and hence contends with e_i. The replacement policy means that e_r always contends for space. The cache contention is formally defined as:

$$con(e_i, T) = \begin{cases} \infty & if\ rd(e_i, T) = \infty \\ |conS(e_i, T)| & otherwise \end{cases} \tag{3}$$

$$\begin{aligned} conS(e_i, T) = \quad & \{j \mid e_j \in T \wedge i - rd(e_i, T) < j < i \wedge \hat{P}(e_j^{hit}) \neq 0\} \\ & \cup \{r \mid r = i - rd(e_i, T)\} \end{aligned}$$

The execution time of an element e_i can be approximated by the discrete random variable $\hat{\xi}_i$ which has a probability mass function (PMF) defined as:

$$\hat{\xi}_i(x) = \begin{cases} \hat{P}(e_i^{hit}) & \text{if } x = \mathcal{H} \\ 1 - \hat{P}(e_i^{hit}) & \text{if } x = \mathcal{M} \\ 0 & otherwise \end{cases} \tag{4}$$

The probabilistic worst-case execution time (pWCET) [2] distribution $\hat{\mathcal{D}}$ of a trace, is an upper-bound on the execution time distribution $\mathcal{D}$, that would be obtained by executing the trace an infinite number of times, such that $\forall v, P(\hat{\mathcal{D}} \geq v) \geq P(\mathcal{D} \geq v)$. In other words, the distribution $\hat{\mathcal{D}}$ is greater than $\mathcal{D}$ [16], denoted $\hat{\mathcal{D}} \geq \mathcal{D}$.

The probability mass functions $\hat{\mathcal{E}}_i$ are independent upper-bounds on the behaviour of corresponding accesses e_i. An estimate for trace T can be derived by combining the probability mass function $\hat{\mathcal{E}}_i$ for each of its composing memory accesses e_i: $\hat{\mathcal{D}}^{con}(T) = \bigotimes_{e_i \in T} \hat{\mathcal{E}}_i$ where $\otimes$ represents the convolution [2] of PMFs:

$$(\hat{\xi}_i \otimes \hat{\xi}_j)(x) = \sum_{k=-\infty}^{+\infty} \hat{\xi}_i(k) \cdot \hat{\xi}_j(x - k) \tag{5}$$

C. Collecting semantics

We now recap on the collecting semantics introduced in [3] as a more precise but more complex alternative to the contention-based method of computing pWCET estimates. This approach performs exhaustive cache state enumeration for a selection of *focused* accesses, hence providing tight analysis results for those accesses. To prevent state explosion, at each point in the program no more than R memory blocks are in *focus* at the same time. The *focused* accesses are ones heuristically identified as benefiting the most from a precise analysis. If access e_i is focused, the block it accesses will be considered focused until the next non-focused access, if any, to the same block. Cache state enumeration is only applied to focused accesses while the contention-based method is used for the others, identified as $\perp$ in the trace. Thus the set of elements becomes $\mathbb{E}^{\perp} = \mathbb{E} \cup \{\perp\}$.

The abstract domain of the analysis is a set of cache states. A cache state is a triplet $CS = (E, P, \mathcal{D})$ with cache contents E, a corresponding probability $P \in \mathbb{R}, 0 < P \leq 1$, and a miss distribution $\mathcal{D} \colon \mathbb{N} \rightarrow \mathbb{R}$ when the cache is in state E. E is a multiset of N elements picked from $\mathbb{E}^{\perp}$. Only $\perp$ may occur multiple times to represent, without further discrimination, partial knowledge about the cache contents or the presence of non-focused blocks. The set of all cache states is denoted by $\mathbb{CS}$. The analysis starts from the empty cache state $\{(\{\perp, ..., \perp\}, 1, (\lambda x.1\ if\ x = 0, 0\ otherwise))\}$.

The update function u describes the update for a single cache state upon access to element $e \in \mathbb{E}^{\perp}$. Upon accessing a focused element $e \neq \perp$, if e is present in the cache, its contents are left unchanged. Otherwise new cache states need to be generated considering that each element may be evicted with the same probability $\frac{1}{N}$ (in the $evict$ function). A miss is accounted for in the resulting distributions $\mathcal{D}'$ only upon misses on a focused access; cache contention is used to predict misses on other accesses. Formally:

$$u((E, P, \mathcal{D}), e) = \begin{cases} \{(E, P, \mathcal{D})\} & if\ e \in E \wedge e \neq \perp \\ evict((E, P, \mathcal{D}), e) & otherwise \end{cases} \tag{6}$$

$$evict((E, P, \mathcal{D}), e) = \left\{ (E \setminus \{e'\} \cup \{e\}, P \cdot \frac{1}{N}, \mathcal{D}' \mid e' \in E \right\}$$

$$\mathcal{D}'(x) = \begin{cases} \mathcal{D}(x) & if\ e = \perp \\ 0 & if\ x = 0 \wedge e \neq \perp \\ \mathcal{D}(x - 1) & otherwise \end{cases}$$

The $evict(s, e)$ function creates N different cache states, one per possible evicted element, some of which might represent the same cache contents. To reduce the state space, a merge operation $\uplus$ combines two cache states if they contain exactly the same memory blocks. If merging occurs, each distribution is weighted by its probability:

$$\begin{aligned} & (E_1, P_1, \mathcal{D}_1) \uplus (E_2, P_2, \mathcal{D}_2) = \\ & \begin{cases} \left\{ (E_1, P_1 + P_2, \left(\frac{P_1}{P_1 + P_2} \cdot \mathcal{D}_1 \right) \oplus \left(\frac{P_2}{P_1 + P_2} \cdot \mathcal{D}_2 \right) \right\} & if E_1 = E_2 \\ \{(E_1, P_1, \mathcal{D}_1), (E_2, P_2, \mathcal{D}_2)\} & otherwise \end{cases} \end{aligned} \tag{7}$$

where $p \cdot \mathcal{D}$ is the multiplication of the elements of distribution $\mathcal{D}$, $(p \cdot \mathcal{D})(x) = p \cdot \mathcal{D}(x)$, and $\mathcal{D}_1 \oplus \mathcal{D}_2$ is the summation of distributions, $(\mathcal{D}_1 \oplus \mathcal{D}_2)(x) = \mathcal{D}_1(x) + \mathcal{D}_2(x)$.

The update function can be defined for a set of cache states using the update function u for a single cache state and the $\uplus$ merge operator as follows: $U(S, e) = \biguplus\{u(CS, e) \mid CS \in S\}$.

Given S_{res} the set of cache states at the end of the execution of a trace T, the miss distribution $\hat{\mathcal{D}}_{miss}$ of T is the sum of the individual distributions of each cache state weighted by their probability of occurrence:

$$\hat{\mathcal{D}}_{miss} = \bigoplus\{\mathcal{D} \cdot P \mid (E, P, \mathcal{D}) \in S_{res}\} \tag{8}$$

The definition of the cache contention given by (3) needs to be updated. The R focused blocks need to always be taken into account in the contention as they may be cached. Conversely, they do not need to be counted in $conS(e_i, T)$ since their contribution is included in R:

$$con(e_i, T) = \begin{cases} \infty & if\ rd(e_i, T) = \infty \\ |conS(e_i, T) \setminus \{i \mid focused(e_i)\}| + R & otherwise \end{cases} \tag{9}$$

The pWCET of a trace can then be derived by convolving the execution time distributions produced by the contention, $\hat{\mathcal{D}}^{con}$, and collecting approaches, $\hat{\mathcal{D}}^{coll}$. The latter is derived from the final miss distribution $\hat{\mathcal{D}}_{miss}$, for a trace with f focused accesses, as follows:

$$\hat{\mathcal{D}}_{miss}(x) = \hat{\mathcal{D}}^{coll}\left(x \times \mathcal{M} + (f - x) \times \mathcal{H}\right) \tag{10}$$

We note that this description of the collecting semantics is necessarily concise. For a more detailed explanation, with examples, see [3].

D. Discussion: Relevance of the model

The SPTA techniques described apply irrespective of whether the contents of the memory block are instruction(s), data or both. While address computation [17] may not be able to pinpoint the exact target of an access, e.g. for data-dependent requests, relational analysis [18], introduced in the context of deterministic systems, can be used to identify accesses which map to the same or different sets, and access the same or different block. Two accesses which obey the same block relation can then be replaced by accesses to the same unique element, hence improving the precision of the analysis.

The methods further assume that there are no inter-task cache conflicts due to preemption, i.e. a run-to-completion semantics with non-preemptable program execution. Analyses of probabilistic cache-related preemption delays have been described in [1]. Concurrent cache accesses are also precluded, i.e. we assume a private cache or appropriate isolation [19].

In practice, detailed analysis could potentially distinguish between different latencies for each access, beyond $\mathcal{M}$ and $\mathcal{H}$, but such precise estimation of the miss latency requires additional analysis steps, e.g. analysis of the main memory [20]. Further, to reduce the pessimism inherent in using a simple bound, events such as memory refresh can be accounted for as part of higher level schedulability analysis [21].

III. Application of SPTA to multi-path

In this section, we improve upon the state-of-the-art SPTA techniques for traces [3] recapitulated in Section II and present methods for multi-path programs, that is complete control-flow graphs. A naive approach would be to compute all possible traces $\mathcal{T}$ of a task, analyse each independently and combine their distributions. However, there are two significant problems with such an approach.

Firstly, while the merge operation (7) could be used to provide a weighted combination given the probability of each path being taken at runtime, such assumptions about path probability do not hold in general. This issue can, however, be resolved by taking the maximum distribution of the resulting execution-time distributions for each trace:

$$\bigodot_{t \in \mathcal{T}} \mathcal{D}(t) \tag{11}$$

$$\mathcal{D}_a \odot \mathcal{D}_b := \mathcal{D}^H \tag{12}$$

$$\mathcal{D}^H(x) = \max\left(\sum_{y \geq x} \mathcal{D}_a(y) - \sum_{y > x} \mathcal{D}^H(y), \sum_{y \geq x} \mathcal{D}_b(y) - \sum_{y > x} \mathcal{D}^H(y), 0\right)$$

where the $\odot$ operator computes a convex hull of the complementary cumulative distribution (1-CDF) of all its operands (similar to the bound in Fig. 1), a maximum of distributions which is valid irrespective of the path executed at runtime.

Secondly, the number of distinct traces is exponential in the number of control flow divergences, branches and loop iterations, which means that this naive approach is computationally infeasible. A standard data-flow analysis is also problematic, since it is not possible to assign to each instruction a corresponding contribution to the execution time distribution.

Our analysis on control-flow graphs resolves these problems. It relies on the collecting and the contention approaches for focused and non-focused blocks respectively, as per the cache collecting approach on traces given in [3]. First, the loops in the control-flow graph are unrolled. This allows the implementation of the following steps, the computation of cache contention, the identification of focused blocks and the collecting semantics, to be performed as simple forward traversals of the control flow graph. Approximation of the possible incoming states on path convergence keeps the analysis tractable. Finally, the contention and collecting distributions are combined using convolution.

A. Program representation

We represent the possible paths in a program using a control-flow graph (CFG), that is a directed graph $G = (V, L, v_s, v_e)$ with a finite set V of nodes, a set $L \subseteq V \times V$ of edges, a start node $v_s \in V$ and an end node $v_e \in V$. Each node v corresponds to an element in $\mathbb{E}$ accessed at v. A path π from node v_1 to node v_k is a sequence of nodes $\pi = [v_1, v_2, \ldots, v_{k-1}, v_k]$ where $\forall i \colon (v_i, v_{i+1}) \in L$. By extension, $[\pi, \pi']$ denotes the path composed of path π followed by path π'. Given a set of nodes V', the symbol $\Pi(V')$ denotes the set of all paths with nodes that are included in V', and $\Pi(G) \subseteq \Pi(V)$ the set of all paths of CFG G from v_s to v_e. Similarly to traces, the pWCET $\hat{\mathcal{D}}(G)$ of a program is a tight upper-bound on the execution time distribution of all possible paths. Hence, $\forall \pi \in \Pi(G), \hat{\mathcal{D}}(G) \geq \mathcal{D}(\pi)$. Fig. 1 illustrates this relation using the 1-CDF $(F(x) = P(\mathcal{D} \geq x))$ of different execution time distributions and a valid pWCET.

We say that a node v_d dominates v_n in the control-flow graph G if every path from the start node v_s to v_n goes through v_d, $v_s \rightarrow^* v_n = v_s \rightarrow^* v_d \rightarrow^* v_n$, where $v_s \rightarrow^* v_d \rightarrow^* v_n$ is the set of paths from v_s to v_n through v_d. Similarly, a node v_p post-dominates v_n if every path from v_n to the end node v_e goes through v_p, $v_n \rightarrow^* v_e = v_n \rightarrow^* v_p \rightarrow^* v_e$. We refer to the set of dominators of node v_n as $dom(v_n)$.

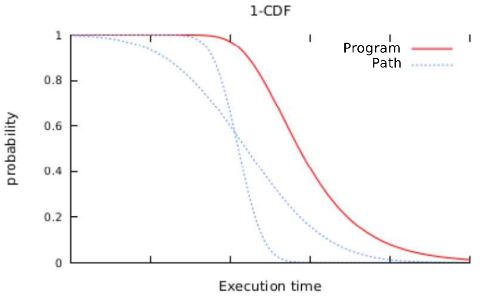

Fig. 1. Relation between paths and program pWCET.

We assume that there is no dead code, and that the program always terminates. These are reasonable assumptions for the software in critical real-time systems. Bounded recursion and loop iterations are requirements to ensure this termination property of the analysed application. The additional restrictions described below are for the most part tied to the WCET analysis framework [11] and not exclusive to the new method.

Any cycle in the CFG must be part of a natural loop. We define a natural loop $l = (v_h, V_l)$ in G with a header $v_h \in V$ and a finite set of nodes $V_l \subseteq V$. The header is the single entry-point of the loop, $\forall v_n \in V_l, v_h \in dom(v_n)$. Conversely, a natural loop may exhibit multiple exits as a result of *break* constructs. Loop l contains at least one back edge to v_h, an edge whose end is a dominator of its source $\exists v_b \in V_l, (v_b, v_h) \in L$. All nodes in the loop can reach one of its back edges without going through the header v_h. The transition from the header v_h of loop l to one of its nodes $v_n \in V_l$ begins an iteration of the loop. The maximum number of consecutive iterations of each loop, i.e. not separated by the traversal of a node outside V_l, is assumed to be upper-bounded by $max\text{-}iter(l, ctx)$. The value of $max\text{-}iter(l, ctx)$ might change depending on the context ctx, call stack and loop iteration, of loop l, e.g. to capture triangular loops. This guarantees a finite number of paths in the program.

Calls are also subject to a minimal set of restrictions to guarantee the termination of the program. Recursion is assumed to be bounded, that is cycles or repetitions in the call graph of the analysed application must have a maximum number of iterations, similarly for loops in the control flow. Function pointers can be represented as multiple targets attached to a single call. Here, the set of targets must be exact or an over-estimate of the actual ones, so as to avoid unsound estimates which omit to take some valid paths into account.

B. Complete loop unrolling and call inlining

In the first analysis step, we conceptually transform the control-flow graph into a directed acyclic graph by loop unrolling and function inlining [22]. Loop unrolling and function inlining are well-known techniques to improve the performance of data-flow analyses.

A complete physical unrolling that removes all back-edges significantly increases the size of the control-flow graph and thus of the size of the analysis problem. A virtual unrolling and inlining is instead performed during analysis and distinguishes the different call and iteration contexts of a vertex. In either case, the size of the graph explored during analysis scales with the length of the program under consideration, similarly to fixpoint-based LRU analyses. Unrolling simplifies the analysis and significantly improves the precision. Successive iterations increase the probability of blocks in the loop working set being in the cache. In contrast to the naive approach of enumerating all possible traces, complete loop unrolling has linear rather than exponential complexity.

C. Reuse Distance/Cache Contention on CFG

To extend the concept of reuse distance (contention) to control-flow graphs, we lift the definition from a single trace to all traces and take the maximal reuse distance (contention) of all possible traces ending in the node v:

$$rd^G(v) = \max_{\pi=[v_s,\dots,v]} (rd(v, \pi)) \tag{13}$$

$$con^G(v) = \max_{\pi=[v_s,\dots,v]} (con(v, \pi)) \tag{14}$$

An upper-bound of both metrics for each access can be computed through a simple forward data flow analysis, using the maximum of the possible values on path convergence.

We then traverse the unrolled control-flow graph in reverse post-order, compute the distributions with the contention-based approach, and use the maximum distribution on path convergence, with the convex hull $\odot$ as the join operator.

D. Selection of focused blocks

The selection of focused blocks is also modified to accommodate for a control-flow graph. Cache state enumeration is only performed for focused accesses, ensuring more precise analysis results for the selected accesses. Earlier work [3] relied on an absolute set of R focused blocks, for the whole trace. Instead, we only restrict ourselves to at most R focused blocks at any point in the program. Given a position in the control-flow, the heuristic tracks the R blocks with the smallest lifespan, i.e. the smallest distance between their last and next access. Such accesses are among the most likely to be kept in the cache and benefit from a precise estimate of their hit probability through state enumeration. Note that this heuristic relies on a lower bound on the lifespan of blocks instead of an upper bound.

E. Ordering of cache state sets

We assume no information about the probability of taking one path or another, hence the join operator must combine cache states in such a way that the resulting state is an over-approximation of both, i.e. it contains the same or degraded information. To capture this property, we introduce the partial ordering $\sqsubseteq$ between sets of cache states such that $S_a \sqsubseteq S_b$ implies that S_a holds more precise information than S_b, resulting in less pessimistic estimates.

Consider a simple cache state $s = (\{a, b\}, 0.5, \mathcal{D})$. (We further omit unknown value $\perp$ in cache contents.) The information represented by $s_a = (\{a\}, 0.3, \mathcal{D})$ is less precise than that captured by s, $s \sqsubseteq s_a$. Indeed, there is no information on the presence of b in s_a. Conversely, $s_c = (\{a, c\}, 0.1, \mathcal{D})$ holds more precise information regarding c, so $s \not\sqsubseteq s_c$. The set $S = \{(\{a\}, 0.25, \mathcal{D}), (\{b\}, 0.25, \mathcal{D})\}$ also approximates s, $s \sqsubseteq S$; the knowledge that a and b are both present in the cache (in s) is reduced to guarantees only about the presence of either a or b (in S). As a consequence, the sequence of accesses $abab$ will trigger more misses starting from states in S, than from state s. Assuming $\mathcal{D} < \mathcal{D}'$, then $s' = (\{a, b\}, 0.5, \mathcal{D}')$ holds more pessimistic information than s, $s \sqsubseteq s'$.

The intuition behind the approximation of a cache state is that the information it captures is further diluted into a single cache state or a set of cache states. By extension, the over-approximation of a set of cache states is the composition of approximations $F(s) \in 2^{\mathbb{CS}}$ of each element s in the set. We formally define the $\sqsubseteq$ partial ordering between sets of cache states $S_a \in 2^{\mathbb{CS}}$ and $S_b \in 2^{\mathbb{CS}}$ as follows:

$$\begin{aligned} S_a \sqsubseteq S_b \Rightarrow \quad & \exists F \colon \mathbb{CS} \rightarrow 2^{\mathbb{CS}}, \\ & (\forall s \in S_a, s \sqsubseteq F(s)) \wedge S_b = \biguplus_{s \in S_a} F(s) \end{aligned} \tag{15}$$

where the relation $s \sqsubseteq S$ holds if the set of cache states S approximates cache state $s = (E, P, \mathcal{D})$. In other words, 1) S is as likely to occur, 2) all blocks known to be in states of S are present in s, and 3) the contribution $\mathcal{D}'$ of S to the pWCET is greater than or equal to the contribution $\mathcal{D}$ of s. Formally:

$$\begin{aligned} (E, P, \mathcal{D}) \sqsubseteq S \quad \Rightarrow & \left(P \geq \left(\textstyle\sum_{(E', P', \mathcal{D}') \in S} P'\right)\right) \wedge \\ & \left(\begin{array}{ll} \forall (E', P', \mathcal{D}') \in S, & E \supseteq (E' \cap \mathbb{E}) \wedge \\ & \mathcal{D} \leq \mathcal{D}' \end{array} \right) \end{aligned} \tag{16}$$

Recall that unknown information in the cache contents E' is represented using $\perp$, hence $E' \cap \mathbb{E}$ only keeps the elements guaranteed to be present in the cache.

A join function $\sqcup$ is valid if given any set of cache states S_a and S_b, then $S_a \sqsubseteq (S_a \sqcup S_b)$ and $S_b \sqsubseteq (S_a \sqcup S_b)$.

F. Join operation for cache collecting

We traverse the (directed acyclic) graph in reverse post-order and compute the set of cache states at each program point. The join operator $\bigsqcup$ describes the combination of two data-flow states from two different sub paths.

Let S_a and S_b be the sets of cache states from the two merging paths. We first define the set of common blocks $\mathbb{M}^{S_a \cap S_b}$, and then restrict S_a and S_b to this set. For brevity, we focus on S'_a, the transposition to S'_b is straightforward:

$$\mathbb{M}^{S_a \cap S_b} = \left(\bigcup_{(E^a, P^a, \mathcal{D}^a) \in S_a} E^a \right) \cap \left(\bigcup_{(E^b, P^b, \mathcal{D}^b) \in S_b} E^b \right) \tag{17}$$

$$S'_a = \biguplus \{(E \cap \mathbb{M}^{S_a \cap S_b}, P, \mathcal{D}) | (E, P, \mathcal{D}) \in S_a\} \tag{18}$$

S'_a and S'_b are safe over-approximations of S_a and S_b respectively. They only contain memory blocks common to both sets of cache states, which can therefore be included in the joined set of cache states.

The set C contains all cache states common to both sets S'_a and S'_b, with the minimum probability of P_a and P_b, and a miss distribution given by the maximum of the individual distributions $\mathcal{D}_a$ and $\mathcal{D}_b$:

$$\begin{aligned} C = \quad & \{(E, \min(P_a, P_b), \mathcal{D}_a \odot \mathcal{D}_b) | (E, P_a, \mathcal{D}_a) \in S'_a \wedge \\ & (E, P_b, \mathcal{D}_b) \in S'_b \wedge E \neq \emptyset\} \end{aligned}$$

We need to collect the remaining cache states that are i) contained in S'_a but not in S'_b, or ii) are common to both sets, but have a higher probability in S'_a than in S'_b:

$$\hat{C}_a = \{(\emptyset, P, \mathcal{D}) | (\emptyset, P, \mathcal{D}) \in S'_a\} \tag{19}$$

$$\begin{aligned} & \uplus \{(\emptyset, P_a, \mathcal{D}_a) | (E, P_a, \mathcal{D}_a) \in S'_a \wedge (E, P_b, \mathcal{D}_b) \notin S'_b \wedge E \neq \emptyset\} \\ & \uplus \{(\emptyset, P_a - P_b, \mathcal{D}_a) | \\ & \qquad (E, P_a, \mathcal{D}_a) \in S'_a \wedge (E, P_b, \mathcal{D}_b) \in S'_b \wedge E \neq \emptyset \wedge P_a > P_b\} \end{aligned}$$

$\hat{C}_a$ and $\hat{C}_b$ contain one element with the same probability.

$$\hat{C} = \{(\emptyset, P, \mathcal{D}_a \odot \mathcal{D}_b) | (\emptyset, P, \mathcal{D}_a) \in \hat{C}_a \wedge (\emptyset, P, \mathcal{D}_b) \in \hat{C}_b\} \tag{20}$$

$C \uplus \hat{C}$ is a safe over-approximation of both S'_a and S'_b with regards to the ordering defined in (15). We can define a function F_a, which gives an over-approximation of each element of S'_a such that $(C \uplus \hat{C}) = \uplus_{s_a \in S'_a} F_a(s_a)$, as follows:

$$F_a(E, P_a, \mathcal{D}_a) = \begin{cases} \{(\emptyset, P_a, \mathcal{D}_a)\} & if E = \emptyset \\ \{(\emptyset, P_a, \mathcal{D}_a)\} & if \nexists (E, P', \mathcal{D}') \in S'_b \\ \{(E, P_b, \mathcal{D}_a \odot \mathcal{D}_b)\} \cup & \\ \{(\emptyset, P_a - P_b, \mathcal{D}_a)\} & if \exists (E, P_b, \mathcal{D}_b) \in S'_b \wedge P_a > P_b \\ \{(E, P_a, \mathcal{D}_a \odot \mathcal{D}_b)\} & if \exists (E, P_b, \mathcal{D}_b) \in S'_b \wedge P_a \leq P_b \end{cases}$$

The join operation is defined as: $S_a \bigsqcup S_b = C \uplus \hat{C}$.

Example: As an illustration, let us consider the state of a 4-way associative cache upon the convergence of two paths $\pi_a = [a, b, c]$ and $\pi_b = [c, a]$. The resulting set of cache states are denoted by S_a and S_b respectively.

S_a	S_b
$(\{a, b, c\}, 6/16, D)$	
$(\{a, c\}, 3/16, D)$	$(\{a, c\}, 12/16, D)$
$(\{b, c\}, 6/16, D)$	
	$(\{a\}, 4/16, D)$
$(\{c\}, 1/16, D)$	

The cache states in S_a and S_b can be reduced to only keep their common blocks $\{a, c\}$. Common states are merged:

S'_a	S'_b
$(\{a, c\}, 9/16, D)$	$(\{a, c\}, 12/16, D)$
	$(\{a\}, 4/16, D)$
$(\{c\}, 7/16, D)$	

The set of common cache states C, with their minimal, guaranteed probability, is defined as $C = \{(\{a, c\}, 9/16, D)\}$. There is no guarantee about the remaining states in S'_a and S'_b or their occurrence probability, they need to be approximated with the empty cache state: $\hat{C}_a = \hat{C}_b = \{(\emptyset, 7/16, D)\}$.

Hence, the result of the join operation on the convergence of paths π_a and π_b is given by:

$$S_a \bigsqcup S_b = \{(\{a, c\}, 9/16, D), (\emptyset, 7/16, D)\}$$

IV. Worst-case execution path expansion

Approximations of the cache contention or the contents of abstract cache states occur on control flow convergence, when two paths in the control flow graph meet. This ensures the validity of the bounds computed by SPTA whatever the exercised path at runtime, while keeping the complexity of the analyses under control. The complete set of possible paths need not be made explicit; however, the loss of information that may occur on flow convergence decreases the tightness of the computed pWCET.

In most applications, there exists some redundancy among paths with regards to their contribution to the pWCET. If a path can be guaranteed to always perform worse than another ($\mathcal{D}(\pi_b) \geq \mathcal{D}(\pi_a)$), the contribution of the former to the pWCET dominates that of the latter, $\mathcal{D}(\pi_b) = \mathcal{D}(\pi_b) \odot \mathcal{D}(\pi_a)$. In which case, the latter path can be removed from the set of paths considered by the analysis, hence reducing the complexity of the control flow, while preserving the soundness of the computed upper-bound.

In this section, we define the notion of inclusion between paths and prove that path inclusion is a sub-case of path redundancy; the execution time distribution of an including path dominates that of any paths it includes. Based on this principle, we introduce program transformations to safely identify and remove from the control flow paths that are included in others, hence improving the precision of the analysis.

A. Path inclusion

A path is said to include another if it contains at least the same sequence of ordered accesses, possibly interleaved with additional ones. As an example, consider paths $\pi_a = [a, b, c, e]$ and $\pi_b = [a, b, c, d, a, e]$ where π_a is included in π_b. The former path can be split into subpaths $\pi_S = [a, b, c]$ and $\pi_E = [e]$, such that $\pi_a = [\pi_S, \pi_E]$. π_b can then be expressed as the interleaving of π_S and π_E with $\pi_V = [d, a]$, i.e. $\pi_b = [\pi_S, \pi_V, \pi_E]$. Similarly, π_b includes $[a, c, d, e]$, but not $[b, a, a]$.

Definition 1 (Including path)**.** *Let π_a and π_b be two paths, such that π_a is the concatenation of two sub-paths π_S and π_E: $\pi_a = [\pi_S, \pi_E]$. The inclusion of π_a in π_b, denoted $\pi_a \trianglelefteq \pi_b$, is recursively defined as either $\pi_b = [\pi_S, \pi_V, \pi_E]$ or, $\pi_b = [\pi_S, \pi_V, \pi'_E]$ where $\pi_E \trianglelefteq \pi'_E$ and $\pi_E \neq \pi'_E$.*

Lemma 1 (Prefix ordering)**.** *The execution time distribution of π_a is smaller than or equal to that of π_a prefixed by any access v_n, $\mathcal{D}(\pi_a) \leq \mathcal{D}([[v_n], \pi_a])$.*

Proof. We use π_n as shorthand for $([[v_n], \pi_a])$. If v_n hits in the cache, then the execution of π_a starts with an equivalent cache state in both cases, and so $\mathcal{D}(\pi_a) \leq \mathcal{D}(\pi_n)$ trivially holds, since $\mathcal{D}(\pi_n) = \mathcal{D}(\pi_a) + \mathcal{H}$ (one additional cache hit).

If v_n is a miss, then we prove that $\mathcal{D}(\pi_a) \leq \mathcal{D}(\pi_n)$ by showing that there is another distribution that both upper bounds $\mathcal{D}(\pi_a)$ and lower bounds $\mathcal{D}(\pi_n)$.

Assume for purposes of the proof, a hypothetical, special cache with the following behaviour. The special cache is identical to the normal random replacement cache, except that it has an additional special cache line (not subject to eviction). This special line is used to hold v_n, thus loading v_n in π_n does not cause an eviction in the rest of the cache. v_n remains in this special cache line until the next access v'_n (if any in π_a) to the same block. At that point there is an eviction in the normal part of the cache and contents of the special cache line are transferred to that location. v'_n is thus a cache hit. After v'_n, the special cache line is no longer used and behaviour reverts to that of a normal random replacement cache.

Let $\mathcal{D}^*(\pi_n)$ be the execution time distribution of path π_n with the special cache. It is evident that the execution time distribution for path π_n assuming the special cache lower bounds that for the normal cache (i.e. $\mathcal{D}^*(\pi_n) \leq \mathcal{D}(\pi_n)$), since at all times, the special cache contains all of the elements in the normal cache.

Next we consider a comparison of the execution time distribution of path π_n with the special cache, and that of path π_a with the normal cache. The behaviour of the normal cache for π_a and the special cache for π_n *after* v_n is identical up until v'_n. This is the case because the special cache contains exactly those elements in the normal cache plus block v_n in the special line, which is first accessed again at v'_n. At v'_n, both caches evict a normal cache line and load v'_n; however in the case of the special cache, this is at a cost of a hit $\mathcal{H}$, whereas in the normal cache the cost is a miss $\mathcal{M}$. After that point, the behaviour of both caches is identical, as they have the same contents and the special cache line is no longer used. Since the special cache incurs an additional cost of a miss $\mathcal{M}$ to initially load v_n, then it follows that $\mathcal{D}^*(\pi_n)$ upper bounds $\mathcal{D}(\pi_a)$. (Note this is also trivially the case if there is no v'_n in π_a, since then $\mathcal{D}^*(\pi_n) = \mathcal{D}(\pi_a) + \mathcal{M}$.)

Since $\mathcal{D}^*(\pi_n)$ upper bounds $\mathcal{D}(\pi_a)$ and lower bounds $\mathcal{D}(\pi_n)$, it follows that $\mathcal{D}(\pi_a) \leq \mathcal{D}(\pi_n)$. □

The relation in Lemma 1 holds for prefixes of arbitrary lengths. From Lemma 1, we know that $\mathcal{D}(\pi_a) \leq \mathcal{D}([[v_n], \pi_a])$ which can be extended to $\mathcal{D}(\pi_a) \leq \mathcal{D}([[v_1, v_2, ..., v_n], \pi_a])$ since $\mathcal{D}([[v_2, ..., v_n], \pi_a]) \leq \mathcal{D}([[v_1, v_2, ..., v_n], \pi_a])$ and so on. Hence, $\forall \pi_s, \pi_a, \mathcal{D}(\pi_a) \leq \mathcal{D}([\pi_s, \pi_a])$.

Theorem 1 (Included path ordering)**.** *If π_a is included in π_b, then the execution time distribution of π_b is greater than or equal to that of π_a, $\pi_a \trianglelefteq \pi_b \Rightarrow \mathcal{D}(\pi_a) \leq \mathcal{D}(\pi_b)$*

Proof. Base case: We need to prove that if $\pi_a \trianglelefteq \pi_b$ such that $\pi_a = [\pi_S, \pi_E]$ and $\pi_b = [\pi_S, \pi_V, \pi_E]$, then $\mathcal{D}(\pi_a) \leq \mathcal{D}(\pi_b)$. From Lemma 1, we know that $\mathcal{D}(\pi_E) \leq \mathcal{D}([\pi_V, \pi_E])$.

The execution of π_S cannot be impacted by accesses in either π_E or π_V. It is therefore the same on both paths π_a and π_b. Whatever cache state is left by the execution of π_S, the execution time distribution of $[\pi_V, \pi_E]$ is either greater than or equal to that of π_E (Lemma 1). Therefore, $\mathcal{D}(\pi_a) \leq \mathcal{D}(\pi_b)$.

Inductive step: Let us assume $\pi_a = [\pi_S, \pi_E]$ and π'_E such that $\pi_E \trianglelefteq \pi'_E$ and $\mathcal{D}(\pi_E) \leq \mathcal{D}(\pi'_E)$. We need to prove that for $\pi_b = [\pi_S, \pi_V, \pi'_E]$, $\mathcal{D}(\pi_a) \leq \mathcal{D}(\pi_b)$. From Lemma 1, we know that $\mathcal{D}([\pi_V, \pi'_E]) \geq \mathcal{D}(\pi'_E)$, and as a consequence $\mathcal{D}([\pi_V, \pi'_E]) \geq \mathcal{D}(\pi_E)$. Further, the execution time distribution of π_S is not impacted by accesses in either π_V, π_E, or π'_E and is the same in π_a and π_b, hence $\mathcal{D}(\pi_a) \leq \mathcal{D}(\pi_b)$. □

We now extend the notion of path inclusion to sets of paths. A set of paths Π is a path-included set of Π° if each path in Π is included in a corresponding path in Π°, $\Pi \trianglelefteq \Pi^\circ \Rightarrow \forall \pi \in \Pi, \exists \pi^\circ \in \Pi^\circ, \pi \trianglelefteq \pi^\circ$. The pWCET of a path is an upper-bound on its execution time distribution, $\mathcal{D}(\pi) \leq \hat{\mathcal{D}}(\pi)$. As a consequence, for each path $\pi \in \Pi$, there is a path in Π° the pWCET of which also upper-bounds the execution time distribution of π. The pWCET of Π° is an upper-bound on the execution time distributions of all paths in Π, $\forall \pi \in \Pi, \hat{\mathcal{D}}(\Pi^\circ) \geq \mathcal{D}(\pi)$. Hence, it is sufficient to perform the pWCET analysis of a CFG G on a reduced set of paths which path-includes the set $\Pi(G)$.

B. Empty conditions removal

Simple conditional constructs may induce paths that are included in others. In particular, any path that goes through an empty branch or case is included in any alternative branch which triggers memory accesses. The edges in a CFG which represent such cases can be safely removed to reduce path indeterminism during pWCET analysis, improving the precision of the results.

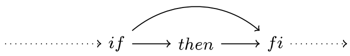

Fig. 2. Simple if-then conditional structure. The edge from if to fi, through the empty $else$ case, can be removed for pWCET estimation.

Fig. 2 gives an example of this for an if-$then$ construct with an empty $else$ branch. At point fi, the analysis accounts for the eviction by accesses in $then$ of blocks present at the end of if. But if the empty edge is kept, any cache block loaded by the $then$ branch cannot be considered as present by the analysis at fi. This reduces the knowledge of the cache contents, and the precision of the resulting pWCET distribution.

An edge from vertex v_p to v_i corresponds to an empty path if there is an alternative exit from v_p through v_j which later reaches v_i, as captured using the notion of post-dominators. In Fig. 2, any path to the program exit through if or $then$ will traverse fi, which post-dominates both if and $then$.

C. Loop unrolling

Natural loop constructs are a source of path redundancy. In particular, paths which do not exercise the maximum number of iterations of a loop they traverse have an including counterpart. An iteration of loop $l = (v_h, V_l)$ starts with a transition from its header v_h to any of its nodes $v_n \in V_l$. Conversely, any iteration, with the exception of the last, ends with a transition back to the header v_h, through a back-edge. The set of paths $\Pi_{iter} = [\Pi(V_l \setminus \{v_h\}, [v_h]]$ captures the paths followed during a complete iteration through loop l.

A valid path which captures n iterations can be expressed as $[[v_h], \pi_1, ..., \pi_{n-1}, \pi_{last}]$ with $\forall i, 1 \leq i < n, \pi_i \in \Pi_{iter}$, and π_{last} is the last iteration of the loop. π_{last} is a path in $\Pi(V_l \setminus \{v_h\})$ followed by a node outside the loop. We denote by Π_n, the set of paths which iterate n times through the loop l. A path in Π_{n+1} can be expressed as $[[v_h], \pi_1, ..., \pi_{n-1}, \pi_n, \pi_{last}]$ with $\pi_n \in \Pi_{iter}$; each path in Π_n is included in a path of Π_{n+1}. By extension, the set of paths $\Pi_{max\text{-}iter(l)}$ path-includes all other sets of paths which iterate over l at least once.

As an example, consider the loop $l = (b, \{b, c, d, e\})$ in Fig. 3. The path $\pi_1 = [a, b, d, e, f]$ iterates a single time through l. The valid iteration sequences, Π_{iter}, in this example are $[d, e, b]$ and $[c, e, b]$. By inserting one iteration before the last in π_1, we obtain the valid paths $[[a, b], [d, e, b], [d, e, f]]$ and $[[a, b], [c, e, b], [d, e, f]]$ respectively. Both paths include π_1 and belong to Π_2.

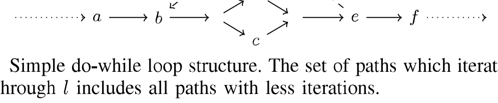

Fig. 3. Simple do-while loop structure. The set of paths which iterates $x + 1$ times through l includes all paths with less iterations.

In our model, we only restrict the maximum number of iterations of a loop. Every iteration may be the last; there is no guarantee that a loop goes always through the same number of iteration when it is executed. The loop unrolling algorithm hence operates without knowledge of the exact number of iterations of the loop. Every unrolled iteration is connected to the successors of the loop. As per Theorem 1 and the inclusion property for consecutive loop iterations, it is sufficient for pWCET estimation to only consider paths where each loop, when executed, goes through its current maximum number of iterations. The unrolling of loop l assumes $max\text{-}iter(l, ctx)$ as the exact iteration count of loop l. In effect, when unrolling any iteration of loop l besides the last, edges from nodes in the loop to nodes outside l are discarded. Conversely, unrolling the last iteration implies conserving only the nodes and edges of l which lead to a loop exit.

The same principles hold for call inlining. Recursion is also a source of path redundancy. Recursive calls manifest as repetitions in the call stack of an application. Here, a single source node is attached to the CFG of each procedure, which identifies its start. The source node therefore behaves similarly to the head of a loop, and is a guaranteed entry to each call. The same logic applies to both natural loops and recursive calls. When performing virtual or physical inlining, the analysis forces recursion up to the defined bound.

V. Evaluation

In this section, we examine the precision and runtime behaviour of the multi-path analysis introduced in this paper. In order to study the behaviour of the analysis with respect to different flow constructs, we provide results for a subset of the Mälardalen, Papabench and debie benchmarks (from the TACLeBench suite[1]). The CFG and address extraction were performed using the Heptane [23] analyser, from the compiled MIPS R2000/R3000 executable obtained using GCC v4.5.2 without optimisations. We used different methods to evaluate the contribution of a 16-way fully associative instruction cache with 32B lines, with $\mathcal{H} = 1$ and $\mathcal{M} = 10$.

A. Relative precision of the multipath analysis

The miss distribution for different benchmarks was computed using either the contention-based approach, the collecting one, using different numbers of focused blocks R, or the reuse distance-based path merging method outlined in [1]. For reference purposes, a state-of-the-art analysis [24] was used to determine the number of misses predicted for a LRU cache using the same parameters. As opposed to SPTA results, the deterministic estimate is a single value upper-bound without any variation. We also performed a set of 10^8 simulations of the random cache behaviour to use as a baseline, effectively providing a lower bound on the pWCET. Here, the successor to each vertex in the simulated path was picked randomly among all of its valid successors, thus exploring the possible paths.

Worst-Case Execution Path (WCEP) expansion was used for analysis and simulation of the random replacement cache. (Theorem 1 does not apply to LRU caches.) The pWCET estimates obtained for each configuration of the analysis, estimation method and number of focused blocks, were always tighter with WCEP expansion. Regarding simulation, WCEP expansion reduces the set of paths to one more representative of the worst-case scenarios. In some cases this results in a single worst possible execution path. Yet there is no guarantee that these transformations are sufficient, here the simulation

[1] http://www.tacle.knossosnet.gr/activities/taclebench

results are only an indicative means of assessing the pessimism in our approach.

The figures show the complementary cumulative miss distributions (1-CDF) for a representative subset of benchmarks and configurations. The contention and path merging approaches are identified by red crosses and purple squares respectively. The number of focused blocks R for the collecting approach is restricted to values of either 4 or 8 (identified by orange triangles and blue diamonds respectively) which is sufficient to capture most of the locality in the considered applications. The distribution obtained through simulation (identified by green circles) is also presented. The number of misses predicted by deterministic analyses for the LRU configuration is identified by a dark blue vertical line.

In general, the use of the cache collecting method improves the precision of the analysis over the merging or purely contention-based approaches even on complex control flows, as illustrated by compress in Fig. 4. Complex flows challenge the precision of the path merging approach, for the compress benchmark our approach predicts less than one third as many misses as the merging approach. The merged path is as long as the longest path in the application but keeps the worst behaving accesses from shorter paths. On simple control flows, the two approaches behave similarly but the contention method still dominates the path merging method (see Fig. 5). When WCEP expansion can extract the worst-case execution path, as with ud, the main difference between the two approaches comes from the more precise estimation of the hit probability of individual accesses using contention methods.

The precision of the collecting methods and the relative performance of LRU and random caches mostly depends on the size of the working set of tasks w.r.t. to the cache size or the number of focused blocks. Similar behaviours were observed whether WCEP expansion successfully resulted in a single path or not. As the number of focused blocks increases from 4 to 8, the estimates computed by the analysis improve. The gain is important on benchmarks like ud (see Fig. 5) where some nested loops fit in the number of focused blocks (8). However, precision is lost w.r.t the simulation results when the loops all fit inside the cache but not within the number of focused blocks. This also results in decreased performance w.r.t. LRU. The latter is in this case only subject to cold misses.

Another general observation is that as expected none of the distributions derived by analysis underestimates simulation. However, the simulation-based distributions cannot be guaranteed to be precise pWCET estimates. The simulations, lacking representative input data, may not exercise the worst-case paths. At best they provide lower bounds on the pWCET. We note that provision of representative input data is a key problem for measurement-based methods. There is no such general conclusion regarding the dominance of the analysis of a LRU cache over the simulation or analysis results for a randomised cache. When all iterative structures fit in the cache (see Fig. 5), the deterministic LRU analysis outperforms analysis of the random cache. As intra-loop conflicts grow, the benefits of the random replacement policy emerge (see Fig. 4) and the new methods can capture such locality, resulting in tighter estimates than deterministic analyses for LRU (see Fig. 6). Note that simulation results for fft were omitted for the sake of clarity, the projection predicts 12000 misses or more with probability 10^{-6}.

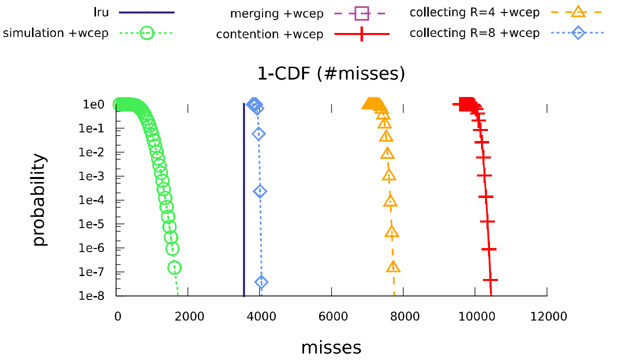

Fig. 4. Results for compress, 86 distinct blocks, 31K accesses on the longest path. For clarity, results for the path merging approach, 31K misses, were omitted.

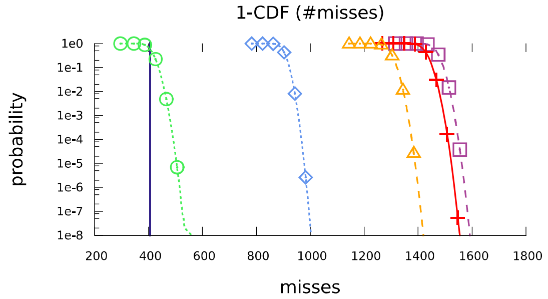

Fig. 5. Results for ud, 75 distinct blocks, 2984 accesses on the longest path.

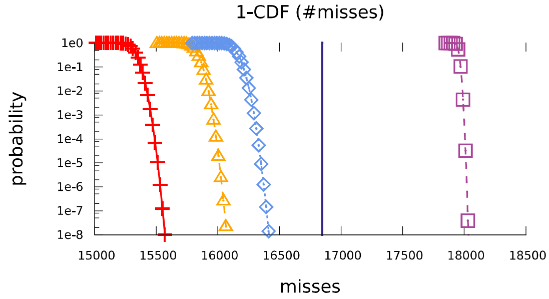

Fig. 6. Results for fft, 141 distinct blocks, 18K accesses on the longest path.

The analysis results for the fft benchmark (see Fig. 6) indicate that the cache collecting approach may sometimes compute more pessimistic estimates than the contention method. This behaviour stems from the combination of flow divergence and large loop constructs in the control flow. Path indeterminism hinders the focused block heuristic, as different blocks may be deemed as focused on parallel paths. In such cases, upon flow convergence, the join function cannot keep blocks of either alternative. Further, the R focused blocks are still considered as occupying cache space from the point of view of the non-focused ones, effectively reducing the cache size. This illustrates the need for more effective heuristics which take into account the behaviour of the analysis on alternative paths, or vary the number of focused blocks depending on the expected benefits, and the computational cost.

In summary, our evaluation results show that the approaches to multi-path SPTA derived in this paper dominate and significantly improve upon the state-of-the-art path merging approach, determining less than one third as many misses in some instances. They were also shown to be incomparable with LRU analysis.

B. Execution time

The runtime of the analysis, using a C++ prototype implementation, is presented in Fig. 7 using the WCEP expansion method and 0 to 12 focused blocks on a subset of the considered benchmarks. Measurements were made on an 8-core 64-bit 3.4Ghz CPU using the Ubuntu 12.04 operating system, with 2 instances of the analyser running in parallel. WCEP expansion was used as it increases the precision of the estimated cache states, and therefore the analysis runtime. We observe a growth in runtime as the number of focused blocks increases. The runtime of the analysis is also significantly higher for larger benchmarks. Table I covers additional applications and includes details of the maximum number of accesses, the distinct number of cache blocks, and the cyclomatic complexity Y of the CFG (without and with WCEP expansion) which lower bounds the number of paths. Also given are the analysis runtimes with 4 and 8 focused blocks.

The abstract cache state representation is partially responsible for the high runtime on the largest benchmark. The complexity of the update and join operations is tied to both the number of focused blocks R and the number of potential misses on the longest path. The number of focused blocks affects the number of different cache contents which are tracked by the analysis at each step. As the number of analysed accesses increases, so does the size of the distributions held in the cache states and therefore the cost of operations such as the merge. The complexity of the analysis is of the order of $O(|S| \times m \times log(m))$, where m is the number of accesses in the program and $|S|$ upper-bounds the number of possible cache states. $|S|$ is the number of combinations of N or less elements picked among the R focused blocks, when $R < N$, $|S| = 2^R$.

Fig. 7 combines the impact of both the program length and number of focused blocks whereas Fig. 8 shows the variation with the number of instructions m only. Fig. 8 presents the runtime of the analysis of a repeated sequence of n accesses while assuming the same 16-way cache as in our previous experiments. The number of blocks in the repeated sequence n, the number of focused blocks R and the cache associativity $N = 16$ impact the possible number of cache states $|S|$ and therefore the initial growth of the complexity. Once the set of cache states to consider stabilises, the runtime for the different configurations follows a similar $m \times log(m)$ growth curve.

As demonstrated in the previous set of experiments, a limited number of focused blocks is effective for typical cache associativities. We recognise that the trade off between complexity is an important one in the use of these techniques on industrial programs. As explored in the next section, leverage can be gained from the use of parameters introduced in the context of deterministic analyses, e.g. partial unrolling [24], distribution re-sampling [25], or control flow graph partitioning [26].

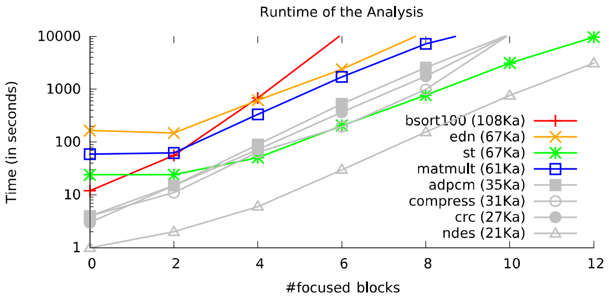

Fig. 7. Runtime evaluation for the largest benchmarks in kilo-accesses(Ka).

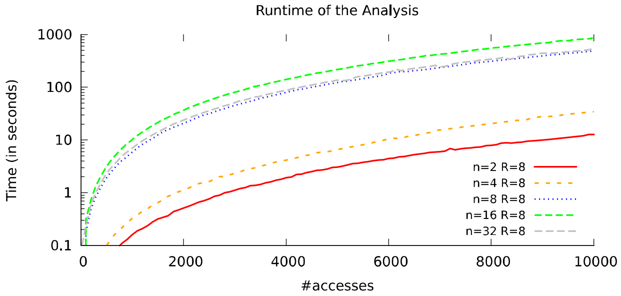

Fig. 8. Runtime for repeated accesses to a sequence of n distinct blocks.

C. Control flow partitioning

In this section, we examine an example of additional leverage to reduce the complexity of our approach. This method, based on control flow graph partitioning, improves the conscious trade off between complexity and precision induced by the selection of a limited set of focused blocks. A loss of precision, can be balanced by focusing on more blocks while resulting in a decrease of analysis runtimes.

We defined a simple algorithm to split a program into consecutive single-entry single-exit regions with M potential misses on their longest path. Segments can then be analysed independently [26], assuming an empty input cache, and their pWCET computed as per (8). The pWCET of the application is then convolved from that of all segments. This approach effectively reduces the set of cache states on region boundaries to the empty state, a safe over-approximation as defined in Section III-E. Fig. 9 presents the resulting analysis runtime for the largest benchmarks assuming a segment size $M = 1000$.

Program partitioning reduces the runtime of our method over the analysis of the program as a single segment (see Fig. 7). As the analysis is applied to same-sized regions in all cases, the runtime of all benchmarks follow a similar growth with the number of focused blocks. The differences in runtime come from several factors. First, the length of the program impacts the complexity of the final convolution. Second, the consecutive segments on a multi-path program may hold more than M misses. Splits can only occur on a reduced set of vertices, namely those which dominates the exit of the CFG.

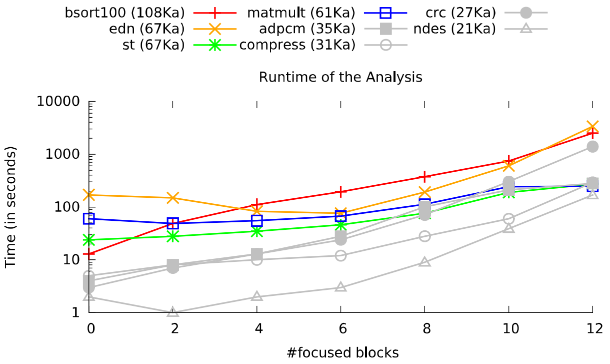

Fig. 9. Runtime evaluation for the largest benchmarks, in kilo-accesses (Ka), using analysis segments of 1000 potential misses.

Further, as shown in Fig. 8, misses and the working set of each segment impact the number of cache states kept during analysis. Finally, flow complexity also increases analysis time as more paths need to be considered in a single segment.

Fig. 10 to 12 presents the distributions computed by the analyses for a relevant subset of the considered configurations. They present the analyses results for $R = 8$ focused blocks using a single or multiple segments (blue squares and red hollow triangles respectively). They also include the results for 12 focused blocks under partitioning (filled red triangles), as the runtime of this configuration is below that of the $R = 8$ single segment one. Simulations and LRU analyses results are also included (resp. with green circles and a dark blue line). WCEP expansion is active in all cases, except LRU.

The approximation of the cache contents on segment boundaries has adverse effects on the precision of the analysis. Indeed, the first few access in a segment may be classified as misses while the contents of the cache are being reloaded. This is illustrated for the matmult and edn benchmark respectively in Fig. 10 and 11. matmult exhibits an important locality at runtime, the impact of segment boundaries is such that it overshadows the increase in the number of focused blocks. Yet, the segmented analysis with 12 focused blocks only takes 285 seconds, against more than 7000 for the single segment with $R = 8$. The precision gain from the increase in the number of focused blocks is much more important for edn, while the runtime of the high R segmented analysis remains lower than that of the low R full program one (2000s versus 13000s).

The fft benchmark (Fig. 12) exhibits an interesting exception. We previously observed that it does not benefit from an increase in the number of focused blocks. As a consequence, the approximations on segment boundaries have almost no impact on the precision of the computed estimates given a fixed number of focused blocks. Increasing that number again further degrades the precision of the resulting distribution.

Here, we have investigated changes to the number of focused blocks and the size of analysed segments, in order to decrease the complexity of the analysis. Lossy analysis state compression [27] also applies in our context as those contributions are orthogonal. Additional techniques, developed for deterministic analyses, could also be used. Partial unrolling [24] can be used to unroll the first few iterations of loops to fast-track the analysis of the remaining iterations using a safe approximation of their input state. Similarly, control flow graph partitioning [26] can reduce the size of the analysed segments while limiting the impact of the segmentation. Re-sampling the distributions allows for a decrease in the size of the convolved distributions, trading an improved runtime for a small loss in precision [25]. The introduction of more leverage using existing methods comes with additional analysis parameters the combinations of which need to be tuned to balance precision and complexity.

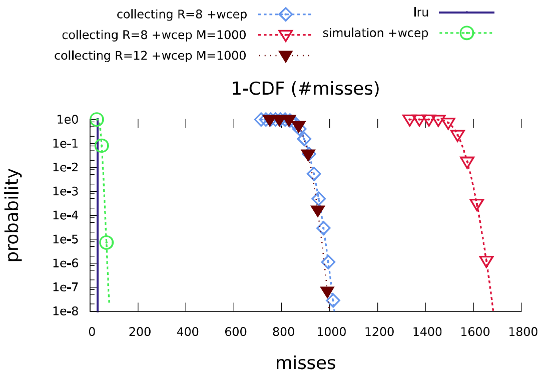

Fig. 10. Analysis results for matmult, 28 distinct blocks, 63Ka.

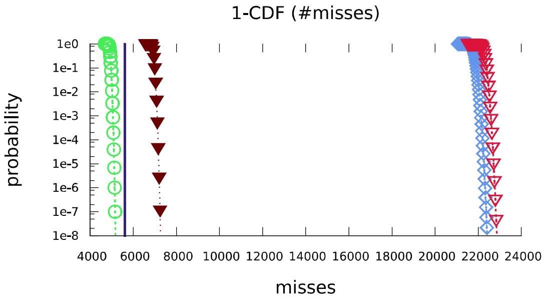

Fig. 11. Analysis results for edn, 166 distinct blocks, 67Ka.

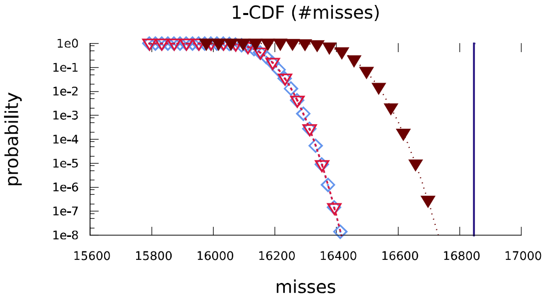

Fig. 12. Analysis results for fft, 141 distinct blocks, 18Ka.

VI. Conclusions

The main contribution of this paper is the introduction of a first effective and non-trivial approach to multipath SPTA for systems that use a cache with an evict-on-miss random replacement policy. The methods presented in this paper build upon existing approaches for analysing single-path programs. While deterministic analyses have benefited from years of refinements, this work is necessarily a relatively simple first

TABLE I
PROPERTIES OF THE ANALYSED BENCHMARKS AND ANALYSIS RUNTIME WITH DIFFERENT NUMBER OF FOCUSED BLOCKS R.

	Longest path	**|Blocks|**	**Runtime (s)**		**Y** with expansion	
	(accesses)		$R=4$	$R=8$	Off	On
MÄLARDALEN						
adpcm	35010	240	90	2549	6281	3069
bsort100	108718	20	683	22819	9902	101
cnt	1576	27	< 1	1	201	101
compress	31382	86	64	988	3976	493
crc	27752	44	72	1759	4173	4169
edn	67631	166	618	12257	5	1
expint	11314	31	7	102	404	104
fft	18409	141	13	250	609	587
fir	992	22	< 1	1	31	11
jfdctint	1059	96	< 1	2	65	1
lcdnum	233	20	< 1	< 1	171	61
ludcmp	3950	98	< 1	23	70	8
matmult	63839	28	332	7240	801	1
minver	726	167	< 1	< 1	7	1
ndes	21377	121	6	154	4219	1273
nsichneu	2944	1377	9	10	1249	1
ns	4349	20	1	28	2	2
prime	5768	17	1	47	725	5
qurt	1526	77	< 1	3	187	67
select	1721	60	< 1	< 1	177	17
sqrt	430	26	< 1	< 1	59	20
statemate	1844	275	< 1	< 1	1841	1132
st	67538	163	51	769	971	221
ud	2984	75	< 1	11	82	1
PAPABENCH						
t1	150	135	< 1	< 1	41	17
t4	215	13	< 1	< 1	47	24
t5	62	55	< 1	< 1	19	13
t6	286	272	< 1	< 1	103	27
t9	472	324	< 1	< 1	89	11
t10	39658	1073	158	1624	16602	10513
t13	581	675	< 1	< 1	204	26
fly_by_wire	18723	229	6	39	4355	1930
DEBIE						
acquisition_task	18664	205	10	450	3829	1273
hit_trigger_handler	3367	83	< 1	6	671	471
tc_execution_task	3131	417	< 1	8	368	251
tc_interrupt_handler	77	91	< 1	< 1	39	27
tm_interrupt_handler	24	30	< 1	< 1	9	7

step, nevertheless, we have pointed out where existing techniques from deterministic analysis could be applied to make improvements [26], [25], [18], [24].

We introduced conditions for the computation of valid upper-bounds on the possible cache states on control flow convergence and presented a compliant join function. We defined path redundancy, identifying path inclusion as a subcase of redundancy. Based on these results, we presented worst-case execution path (WCEP) expansion to reduce the set of paths explored by the analysis, improving the tightness of the resulting pWCET estimates.

Our evaluations show that the analysis derived is efficient at capturing the cache locality exhibited by different applications. The new methods significantly outperform the existing path merging approach, predicting less than a third as many misses in one of the benchmarks. Precise results can be attained at the cost of an increased, user-controlled, complexity. They are incomparable to deterministic estimates for LRU caches. Further, the program transformations introduced proved effective at improving the precision of all SPTA configurations.

ACKNOWLEDGEMENT

This work was partially funded by the EU FP7 Integrated Project PROXIMA (611085), the Inria International Chair program, and ICT COST Action IC1202: Timing Analysis On Code-Level (TACLe).

REFERENCES

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

[2] L. Cucu-Grosjean, "Indépendance - a misunderstood property of and for probabilistic real-time systems," in *Alan Burns 60th Anniversary*, 2013.

[3] S. Altmeyer and R. I. Davis, "On the Correctness, Optimality and Precision of Static Probabilistic Timing Analysis," in 17^{th} *Conference on Design, Automation and Test in Europe (DATE)*, 2014.

[4] S. Altmeyer, L. Cucu-Grosjean, and R. Davis, "Static probabilistic timing analysis for real-time systems using random replacement caches," *Real-Time Systems*, vol. 51, no. 1, pp. 77–123, 2015.

[5] L. Cucu-Grosjean et al., "Measurement-Based Probabilistic Timing Analysis for Multi-path Programs." in *24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[6] G. Bernat, A. Colin, and S. Petters, "WCET analysis of probabilistic hard real-time systems," in *23rd Real-Time Systems Symposium (RTSS)*, 2002.

[7] E. Quinones, E. D. Berger, G. Bernat, and F. J. Cazorla, "Using randomized caches in probabilistic real-time systems," in *21st Euromicro Conference on Real-Time Systems (ECRTS)*, 2009.

[8] F. Cazorla et al., "PROARTIS: Probabilistically Analysable Real-Time Systems," *Transactions on Embedded Computing Systems*, 2013.

[9] L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla, "A Cache Design for Probabilistically Analysable Real-time Systems," in *16th Conference on Design, Automation and Test in Europe (DATE)*, 2013, pp. 513–518.

[10] J. Reineke, "Randomized Caches Considered Harmful in Hard Real-Time Systems," *LITES*, vol. 1, no. 1, pp. 03:1–03:13, 2014.

[11] R. Wilhelm et al., "The Worst-case Execution-time Problem;Overview of Methods and Survey of Tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, May 2008.

[12] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager, "Time-critical Computing on a Single-chip Massively Parallel Processor," in *Conference on Design, Automation & Test in Europe (DATE)*, 2014.

[13] L. David and I. Puaut, "Static determination of probabilistic execution times," in *16th Euromicro Conference on Real-Time Systems*, 2004.

[14] Y. Liang and T. Mitra, "Cache Modeling in Probabilistic Execution Time Analysis," in *45th Design Automation Conference (DAC)*, 2008.

[15] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A Definition and Classification of Timing Anomalies." in *6th Workshop on Worst-Case Execution Time Analysis (WCET)*, 2006.

[16] J. López, J. Díaz, J. Entrialgo, and D. García, "Stochastic analysis of real-time systems underpreemptive priority-driven scheduling," *Real-Time Systems*, vol. 40, no. 2, 2008.

[17] B. K. Huynh, L. Ju, and A. Roychoudhury, "Scope-Aware Data Cache Analysis for WCET Estimation," in *17th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[18] S. Wegener, "Computing Same Block Relations for Relational Cache Analysis," in *12th Workshop on Worst-Case Execution Time Analysis (WCET)*, 2012.

[19] D. Chiou, D. Chiouy, L. Rudolph, L. Rudolphy, S. Devadas, S. Devadasy, B. S. Ang, and B. S. Angz, "Dynamic Cache Partitioning via Columnization," 2000.

[20] R. Bourgade, C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "Accurate analysis of memory latencies for WCET estimation," in *16th Conference on Real-Time and Network Systems (RTNS)*, 2008.

[21] B. Bhat and F. Mueller, "Making DRAM refresh predictable," *Real-Time Systems*, vol. 47, no. 5, pp. 430–453, 2011.

[22] S. S. Muchnick, *Advanced Compiler Design and Implementation*, 1997.

[23] A. Colin and I. Puaut, "A Modular and Retargetable Framework for Tree-based WCET Analysis," in *13th Euromicro Conference on Real-Time Systems (ECRTS)*, 2001.

[24] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and Precise WCET Prediction by Separated Cache and Path Analyses," *REAL-TIME SYSTEMS*, vol. 18, pp. 157–179, 1999.

[25] D. Maxim et al., "Re-sampling for Statistical Timing Analysis of Real-time Systems," in *20th Conference on Real-Time and Network Systems (RTNS)*, 2012.

[26] B. Pasdeloup, "Static probabilistic timing analysis of worst-case execution time for random replacement caches," INRIA, Tech. Rep., 2014.

[27] D. Griffin, B. Lesage, A. Burns, and R. Davis, "Lossy compression for static probabilistic timing analysis of random replacement caches," in *22st Conference on Real-Time Networks and Systems (RTNS)*, 2014.