White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of York.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/125897/

Version: Published Version

**Article:**
Lesage, Benjamin Michael Jean-Rene, Griffin, David Jack, Altmeyer, Sebastian et al. (2018) On the analysis of random replacement caches using static probabilistic timing methods for multi-path programs. Real-Time Systems. pp. 307-388. ISSN: 1573-1383

https://doi.org/10.1007/s11241-017-9295-2

**Reuse**
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

**Takedown**
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

Real-Time Syst
https://doi.org/10.1007/s11241-017-9295-2

# On the analysis of random replacement caches using static probabilistic timing methods for multi-path programs

**Benjamin Lesage**[1] · **David Griffin**[1] · **Sebastian Altmeyer**[2] · **Liliana Cucu-Grosjean**[3] · **Robert I. Davis**[1,3]

© The Author(s) 2017. This article is an open access publication

**Abstract** Probabilistic hard real-time systems, based on hardware architectures that use a random replacement cache, provide a potential means of reducing the hardware over-provision required to accommodate pathological scenarios and the associated extremely rare, but excessively long, worst-case execution times that can occur in deterministic systems. Timing analysis for probabilistic hard real-time systems requires the provision of probabilistic worst-case execution time (pWCET) estimates. The pWCET distribution can be described as an exceedance function which gives an upper bound on the probability that the execution time of a task will exceed any given execution time budget on any particular run. This paper introduces a more 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 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

✉ Benjamin Lesage
benjamin.lesage@york.ac.uk

David Griffin
david.griffin@york.ac.uk

Sebastian Altmeyer
altmeyer@uva.nl

Liliana Cucu-Grosjean
liliana.cucu@inria.fr

Robert I. Davis
rob.davis@york.ac.uk

1 University of York, York, UK

2 University of Amsterdam, Science Park 904, Room C3.101, 1098 XH, Amsterdam, Netherlands

3 INRIA, Paris, France

Published online: 18 December 2017

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. For some benchmarks the performance of LRU is better, while for others, the new analysis techniques show that random replacement has provably better performance.

**Keywords** Cache analysis · Probabilistic timing analysis · Random replacement policy · Multi-path

## Extensions

This paper builds upon previous work published in RTSS 2015 (Lesage et al. 2015a) with the following extensions:

- we introduce and prove additional properties relevant to the comparison of the contribution of different cache states to the probabilistic worst-case execution time of tasks in Sect. 3;
- an improved join transfer function, used to safely merge states from converging paths, is introduced in Sect. 5 and by construction dominates the simple join introduced in Lesage et al. (2015a);
- we present and prove the validity of path renaming in Sect. 6 which allows the definition of additional transformations to reduce the set of paths considered during analysis;
- our evaluation explores new configurations in terms of both the analysis methods used and the benchmarks considered (see Sect. 7).

## 1 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 overprovision 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. The use of cache with a random replacement policy means that the probability of pathological worst-case 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. This allows

the extreme worst-case behaviours to be safely ignored, instead of always included in the estimated worst-case execution times.

The random replacement policy further offers a trade-off between performance and cost thanks to a minimal hardware cost (Al-Zoubi et al. 2004). The policy and variants have been implemented in a selection of embedded processors (Hennessy and Patterson 2011) such as the ARM Cortex series (2010), or the Freescale MPC8641D (2008). Randomisation further offers some level of protection against side-channel attacks which allow the leakage of information regarding the running tasks. While methods relying solely on the random replacement policy may still be circumvented (Spreitzer and Plos 2013), the definition of probabilistic timing analysis is a step towards the analysis of other approaches such as randomised placement policies (Wang and Lee 2007; 2008).

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 Davis et al. (2013) for examples of pWCET distributions, and to Cucu-Grosjean (2013) 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. (2013) used a 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 (2014, 2015). This new analysis provides substantially improved results compared to the path merging approach. To allow the analysis of the behaviour of caches in isolation, we assume the existence of a valid decomposition of the architecture with regards to cache effects with bounded hit and miss latencies (Hahn et al. 2015).

### 1.1 Related work

We now set the work on SPTA in context with respect to related work on both probabilistic hard real-time systems and cache analysis for deterministic replacement policies. The methods introduced in this paper belong to the realm of analyses that estimate bounds on the execution time of a program. These bounds may be classified as either a worst-case probability distribution (pWCET) or a worst-case value (WCET).

The first class is a more recent research area with the first work on providing bounds described by probability distributions published by Edgar and Burns (2000, 2001). The methods for obtaining such distributions can be categorised into three different families: measurement-based probabilistic timing analyses, static probabilistic timing analyses, and hybrid probabilistic timing analyses.

The second class is a mature area of research and the interested reader may refer to Wilhelm et al. (2008) for an overview of these methods. A specific overview of cache analysis for deterministic replacement policies together with a comparison between deterministic and random cache replacement policies is provided at the end of this section.

### *1.1.1 Probabilistic timing analyses*

Measurement-based probabilistic timing analyses (Bernat et al. 2002; Cucu-Grosjean et al. 2012) collect observations on the execution time of the task under study on the target hardware. These observations are then combined, e.g. through the use of extreme value theory (Cucu-Grosjean et al. 2012), to produce the desired worst-case probabilistic timing estimate. Extreme Value Theory may potentially underestimate the pWCET of a program as shown by Griffin and Burns (2010). The work of Cucu-Grosjean et al. (2012) overcomes this limitation and also introduces the appropriate statistical tests required to treat worst-case execution times as rare events. The soundness of the results produced by such methods is tied to the observed execution times which should be representative of the ones at runtime. This implies a responsibility on the user who is expected to provide input data to exercise the worst-case paths, less the analysis results in unsound estimates (Lesage et al. 2015b). These methods nonetheless exhibit the benefits of time-randomised architectures. The occurrence probability of pathological temporal cases can be bounded and safely ignored provided they meet requirements expressed in terms of failure rates.

Path upper-bounding (Kosmidis et al. 2014) defines a set of program transformations to alleviate the responsibility of the user to provide inputs which cover all execution paths. The alternative paths of conditional constructs are padded with semantic-preserving instructions and memory accesses such that any path followed in the modified program is an upper-bound of any of the original alternatives. Measurement-based analyses can then be performed on the modified program as the paths exercised at runtime upper-bound any alternative in the original application. Hence, upper-bounding creates a distinction between the original code and the measured one. It may also result in paths which are the sum of the original alternatives.

Hybrid probabilistic timing analyses are methods that apply measurement-based methods at the level of sub-programs or blocks of code and then operations such as convolution to combine these bounds to obtain a pWCET for the entire program. The main principles of hybrid analysis were introduced by Bernat et al. (2002, 2003) with execution time probability distributions estimated at the level of sub-programs. Here, dependencies may exist among the probability distributions of the sub-programs and copulas are used to describe them (Bernat et al. 2005).

By contrast, SPTAs derive the pWCET distribution for a program by analysing the structure of the program and modelling 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 (Quinones et al. 2009) and evict-on-access (Cucu-Grosjean et al. 2012; Cazorla et al. 2013) 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. (2013) 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 (2014) proved the correctness of the lower bound derived in Davis et al. (2013), 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 Kosmidis et al. (2013) and Quinones et al. (2009) are unsound (optimistic) for use in SPTA. In 2013, a simple SPTA for multipath programs was introduced by Davis et al. (2013), 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 (2014). 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 (2014) also introduced a significantly more effective method based on combining exhaustive evaluation of the cache behaviour for a limited number of *relevant* memory blocks with cache contention. This method provides an effective trade-off between analysis precision and tractability. Griffin et al. (2014a) introduces orthogonal Lossy compression methods on top of the cache states enumeration to improve the trade-off between complexity and precision.

Altmeyer and Davis further refined their approach to SPTA for single path programs in 2015 (Altmeyer et al. 2015), bridging the gap between contention and enumeration-based analyses. The method relies on simulation of the behaviour of a random replacement cache. As opposed to exhaustive state analyses however, focus is set at each step on a single cache state to capture the outcome across all possible states. The resulting approach offers an improved precision over contention-based methods, at a lower complexity than exhaustive state analyses.

In this paper, we build upon the state-of-the-art approach (Altmeyer and Davis 2014), extending it to multi-path programs. The techniques introduced in the following notably allow for the identification on control flow convergence of relevant cache contents, i.e. the identification of the outcomes in multi-path programs. The approach focuses on the enumeration of possible cache states at each point in the program. To reduce the complexity of such an approach, only a few blocks, identified as the most relevant, are analysed at a given time.

### *1.1.2 Deterministic architectures and analyses*

Static timing analysis for deterministic caches (Wilhelm et al. 2008) relies on a two step approach with a low-level analysis to classify the cache accesses into hits and misses (Theiling et al. 1999) and a high-level analysis to determine the length of the worst-case path (Li and Malik 2006). The most common deterministic replacement policies are least-recently used (LRU), first-in first-out (FIFO) and pseudo-LRU (PLRU). Due to the high-predictability of the LRU policy, academic research typically focusses on LRU caches–with a well-established LRU cache analysis based on abstract interpretation (Alt et al. 1996; Theiling et al. 1999). Only recently, analyses for FIFO (Grund and

Reineke 2010) and PLRU (Grund and Reineke 2010; Griffin et al. 2014b) have been proposed, both with a higher complexity and lower precision than the LRU analysis due to specific features of the replacement policies. Despite the focus on LRU caches and its analysability, FIFO and PLRU are often preferred in processor designs due to the lower implementation costs which enable higher associativities.

Recently, Reineke (2014) observed that SPTA based on reuse distances (Davis et al. 2013) results, by construction, in less precise bounds than existing analyses based on stack distance for an equivalent system with a LRU cache (Wilhelm et al. 2008). However, this does not hold for the more sophisticated SPTA based on cache contention and collecting semantics given by Altmeyer and Davis (2014). Analyses for deterministic LRU caches 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 Altmeyer and Davis 2014), hence SPTA for a random replacement cache outperforms analysis of LRU in this case. We note that in spite of recent efforts (de Dinechin et al. 2014) 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 (David and Puaut 2004; Liang and Mitra 2008) 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 assumptions may not be available. 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.

### 1.2 Organisation

In this paper, we introduce a set of methods that are required for the application of SPTA to multi-path programs. Section 2 recaps the assumptions and methods upon which we build. These were used in previous work (Altmeyer and Davis 2014) to upper-bound the pWCET distribution of a trace corresponding to a single path program. We then proceed by defining key properties which allows the ordering of cache states w.r.t. their contribution to the pWCET of a program (Sect. 3). We address the issue of multi-path programs in the context of SPTA in Sect. 4. 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. Further improvements on cache state conservation at control flow convergence are introduced in Sect. 5. Section 6 introduces simple program transformations which improve the precision of the analysis while ensuring 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 Sect. 7 and the precision and run-time of the different approaches

compared. Section 8 concludes with a summary of the main contributions of the paper and a discussion of future work.

## 2 Static probabilistic timing analysis

In this section, we recap on state-of-the-art SPTA techniques for single path programs (Altmeyer and Davis 2014). We first give an overview of the system model assumed throughout the paper in Sect. 2.1. We further recap on the existing methods (Altmeyer and Davis 2014) to evaluate the pWCET of a single path trace using a collecting approach (Sect. 2.2) supplemented by a contention one. The pertinence of the model is discussed at the end of this section. The notations introduced in the present contributions have been summarised in Table 1.

We assume an architecture for which a valid decomposition exists with regards to the cache, such that its timing contribution can be analysed in isolation from other components (Hahn et al. 2015). Further, the overall execution time penalty emanating from cache misses and hits are assumed to be bounded by the latencies assumed by the analysis. Thus a local worst-case, a miss in the context of the cache, can be added to the local worst-case for other components to obtain a bound on the global worst case (Reineke et al. 2006). This enables analysis of the impact of the cache in isolation from other architectural features.

### 2.1 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, such that $\mathcal{H} < \mathcal{M}$. (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.)

### 2.2 Collecting semantics

We now recap on the collecting semantics introduced by Altmeyer and Davis (2014) 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 *relevant* 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 *relevant* at the same time. The *relevant* accesses are ones heuristically identified as benefiting the most from a precise analysis.

A trace $t$ is defined as an ordered sequence $[e_1, \ldots, e_n]$ of $n$ accesses to memory blocks, such that $e_i = e_j$ if accesses $e_i$ and $e_j$ target the same memory block. If access

**Table 1** Summary of introduced notations

| Notation | Description |
|---|---|
| pWCET | Upper-bound on the execution time distribution of a program over all paths |
| $\mathcal{H}$ | Upper-bound on the latency incurred by a cache hit |
| $\mathcal{M}$ | Upper-bound on the latency incurred by a cache miss |
| $N$ | Cache associativity |
| $\mathbb{E}$ | Set of accessed cache blocks |
| $\mathbb{E}^{\perp}$ | Set of accessed cache blocks including non-relevant elements $\perp$ |
| $t = [e_1, \ldots, e_i]$ | A trace, a sequence of accesses to memory blocks |
| $\mathcal{D}$ | Execution time or cache miss probabilistic distribution |
| $\mathcal{D}(x)$ | Occurrence probability of execution time $x$ |
| $P(\mathcal{D} \geq x)$ | Likelyhood that distribution $\mathcal{D}$ exceeds execution time $x$ |
| $s \in \mathbb{CS}$ | Analysed cache state |
| $(C, P, \mathcal{D}) = s$ | Analysed cache state including:<br>- $C$: Cache contents, set of blocks known to be present in cache;<br>- $P$: Occurrence probability of the cache state at a specific program point;<br>- $\mathcal{D}$: Execution time distribution up to a specific program point |
| $\mathcal{D}_{init}$ | Initial, empty execution time distribution |
| $S \in 2^{\mathcal{CS}}$ | Set of possible caches states at a specific program point |
| $S \uplus S'$ | Weighted merge on cache states, merge probability and distributions for cache states with identical contents |
| $u(s, e)$ | Update cache state $s$ upon access to element $e$, replacing a line and increasing the corresponding distribution $D$ upon a miss |
| $U(S, e)$ | Update each cache state in set $S$ upon access to element $e$ |
| $rd(e, t)$ | Reuse distance of element $e$ in trace $t$, upper-bound on the number of evictions since the last access to $e$ |
| $frd(e, t)$ | Forward reuse distance of element $e$ in trace $t$, upper-bound on the number of evictions before the next access to $e$ |
| $con(e, t)$ | Cache contention for element $e$ in trace $t$, bound on the number of blocks contending for cache space since the last access to $e$ |
| $\hat{P}(e_i^{hit})$ | Lower-bound on the probability of access $e_i$ to hit in cache |
| $\hat{\xi}_i$ | Upper-bound on the execution time probability of element $e_i$, expressed as a probability mass function |
| $\hat{\mathcal{D}}(t)$ | Upper-bound on the execution time distribution of trace $t$ |
| $\mathcal{D}(t, s)$ | Execution time distribution of trace $t$ starting from cache state $s$ |
| $\mathcal{D}(t, S)$ | Execution time distribution of trace $t$ starting from possible cache states $S$ |
| $\mathcal{D} \otimes \mathcal{D}'$ | Convolution of distributions $\mathcal{D}$ and $\mathcal{D}'$ |
| $\mathcal{D} \odot \mathcal{D}'$ | Least upper-bound of distributions $\mathcal{D}$ and $\mathcal{D}'$ |
| $\mathcal{D} \leq \mathcal{D}'$ | Distribution $\mathcal{D}'$ upper-bounds $\mathcal{D}$, iff $\forall x, P(\mathcal{D} \geq x) \leq P(\mathcal{D}' \geq x)$ |
| $G = (V, L, v_s, v_e)$ | Control flow graph $G$ capturing possible paths in a program, including:<br>$V$: Set of nodes in the program, each corresponding to an accessed element;<br>$L$: Set of edges between nodes; |

**Table 1** continued

| Notation | Description |
|---|---|
| | $v_s \in V$: Start node in the program; |
| | $v_e \in V$: End node in the program |
| $\pi = [v_1, \ldots, v_k]$ | Path from node $v_1$ to $v_k$, valid sequence of nodes in a CFG |
| $v_i \rightarrow *v_j$ | Set of paths from $v_i$ to $v_j$ |
| $dom(v_n)$ | Dominators of node $v_n$, nodes guaranteed to be traversed before $v_n$ from the CFG entry $v_s$ |
| $post\text{-}dom(v_n)$ | Post-dominators of node $v_n$, nodes guaranteed to be traversed after $v_n$ to the CFG exit $v_e$ |
| $\Pi(V)$ | All paths with nodes included exclusively in set of vertices $V$ |
| $\Pi(G)$ | All paths from the start to the end of CFG $G$ |
| $\hat{\mathcal{D}}(\pi)$ | Upper-bound on the execution time distribution of path $\pi$ |
| $\hat{\mathcal{D}}(G)$ | pWCET of $G$, upper-bound on the execution time of its paths |
| $rd^G(v)$ | Maximum reuse distance of node $v$ across all paths in $G$ leading to $v$ |
| $frd^G(v)$ | Maximum forward reuse distance of node $v$ across all paths in $G$ leading to $v$ |
| $con^G(v)$ | Maximum contention of node $v$ across all paths in $G$ leading to $v$ |
| $s \sqsubseteq S$ | Cache state $s$ holds less pessimistic information than the set of cache states $S$ |
| $S \sqsubseteq S'$ | The set of cache states $S$ holds less pessimistic information than states in $S'$ |
| $S \sqcup S'$ | Upper-bound on cache states $S$ and $S'$, more pessimistic than both $S$ and $S'$ |
| $C \preceq_{rank} C'$ | Ranking of cache contents $C$, used for heuristic comparison of contents based on their expected contribution to execution time distribution |
| $Flush(S)$ | Empty the contents of all cache states in $S$ |

$e_i$ is relevant, the block it accesses will be considered relevant until the next non-relevant access to the same block. The precise approach is only applied for relevant accesses while the contention-based method outlined in Sect. 2.2.1 is used for the others, identified as $\perp$ in the trace of relevant blocks. The set of elements in a trace 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 $CS = (C, P, \mathcal{D})$ with cache contents $C$, a corresponding probability $P \in \mathbb{R}, 0 < P \leq 1$, and a miss distribution $\mathcal{D}: \mathbb{N} \rightarrow \mathbb{R}$ when the cache is in state $C$. $C$ is a set of at most $N$ memory blocks picked from $\mathbb{E}$. A cache state which holds less than $N$ memory blocks represents partial knowledge about the cache contents without any distinction between empty lines or unknown contents.[1] The set of all cache states is denoted by $\mathbb{CS}$. Miss distribution $\mathcal{D}$ captures for each possible number of misses $n$, the probability that $n$ misses occurred from the beginning of the program up to the current point in the program. The method computes all possible behaviours of the random cache with the associated probabilities. It is thus correct by construction as it simply enumerates all states exhaustively.

[1] This suits evict-on-miss caches which do not prioritize empty lines when filling the cache.

The analysis starts from the empty cache state $\{(\emptyset, 1, \mathcal{D}_{init})\}$ where

$$\mathcal{D}_{init}(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

The update function $u$ describes the update for a single cache state upon access to element $e \in \mathbb{E}^{\perp}$. Upon accessing a relevant 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 relevant access. Formally:

$$u : \mathbb{CS} \times \mathbb{E}^{\perp} \to 2^{\mathbb{CS}} \tag{2}$$

$$u((C, P, \mathcal{D}), e) = \begin{cases} \{(C, P, \mathcal{D})\} & \text{if } e \in C \wedge e \neq \perp \\ evict((C, P, \mathcal{D}), e) & \text{otherwise} \end{cases} \tag{3}$$

$$evict((C, P, \mathcal{D}), e) = \begin{cases} \{(C \backslash \{e'\} \cup \{e\}, P \cdot \frac{1}{N}, \mathcal{D}') \mid e' \in C\} \cup \{(C \cup \{e\}, P \cdot \frac{N-|C|}{N}, \mathcal{D}')\} & \text{if } e \neq \perp \\ \{(C \backslash \{e'\}, P \cdot \frac{1}{N}, \mathcal{D}') \mid e' \in C\} \cup \{(C, P \cdot \frac{N-|C|}{N}, \mathcal{D}')\} & \text{if } e = \perp \end{cases} \tag{4}$$

$$\mathcal{D}'(x) = \begin{cases} \mathcal{D}(x) & \text{if } e = \perp \\ 0 & \text{if } x = 0 \\ \mathcal{D}(x-1) & \text{otherwise} \end{cases} \tag{5}$$

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 $\biguplus$ combines two cache states if they contain exactly the same memory blocks. If merging occurs, each distribution is weighted by its probability:

$$\biguplus : 2^{\mathbb{CS}} \to 2^{\mathbb{CS}} \tag{6}$$

$$\biguplus \left( \left\{ \begin{array}{c} (C_0, P_0, \mathcal{D}_0) \\ \vdots \\ (C_n, P_n, \mathcal{D}_n) \end{array} \right\} \right) = \left\{ Merge\left( \{(C_i, P_i, \mathcal{D}_i) | C_i = C_j\} \right) \middle| 0 \leq j \leq n \right\} \tag{7}$$

$$Merge \left( \left\{ \begin{array}{c} (C_0, P_0, \mathcal{D}_0) \\ \vdots \\ (C_n, P_n, \mathcal{D}_n) \end{array} \right\} \right) = \left( C_0, \sum_{i=0}^{n} P_i, \sum_{i=0}^{n} \frac{P_i}{\sum_{k=0}^{n} P_k} \cdot \mathcal{D}_i \right) \tag{8}$$

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

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:

$$U : 2^{\mathbb{CS}} \times \mathbb{E}^{\perp} \to 2^{\mathbb{CS}} \tag{9}$$

$$U(S, e) = \biguplus \{u(CS, e) \mid CS \in S\} \tag{10}$$

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 the relevant blocks in $t$ is the sum of the individual distributions of each cache state weighted by their probability of occurrence:

$$\hat{\mathcal{D}}_{miss} = \sum \{P \cdot \mathcal{D} \mid (C, P, \mathcal{D}) \in S_{res}\} \tag{11}$$

The corresponding execution time distribution, $\hat{\mathcal{D}}$, can then be derived, for a trace of $n$ accesses, as follows:

$$\hat{\mathcal{D}}\,(m \times \mathcal{M} + (n-m) \times \mathcal{H}) = \hat{\mathcal{D}}_{miss}(m) \tag{12}$$

*2.2.1 Non-relevant blocks analysis*

One possible naive approach for non-relevant blocks would be to classify them as misses in the cache and add the resulting latency to the previously computed distributions. The collecting approach proposed by Altmeyer and Davis (2014) relies on the application of the contention methods to estimate the behaviour of the non-relevant blocks in a trace. Each access in a trace has a probability of being a cache hit $P(e_i^{hit})$, and of being a cache miss $P(e_i^{miss}) = 1 - P(e_i^{hit})$. These methods rely on different metrics to lower-bound the hit probability of each access such that the derived bound can be soundly convolved.

The reuse distance $rd(e)$ of element $e$ is the maximum number of accesses to consecutively different blocks since the last access to the same block. It captures an upper-bound on the maximum number of possible evictions between two accesses to the same block, similarly to the stack distance for LRU caches. It differs from the stack distance in that accesses to the same intermediate block may thus be accounted for multiple times if they may have been evicted during the access sequence. Should there be no such prior access to the same block, the reuse distance is defined as $\infty$. Given the set of all traces $\mathbb{T}$ and of all elements $\mathbb{E}$, the reuse distance is formally defined as:

$$rd : \mathbb{E} \times \mathbb{T} \to \mathbb{N} \cup \{\infty\}$$
$$rd(e_i, [e_1, \ldots, e_{i-1}]) = \begin{cases} |\{k | j < k < i \wedge e_k \neq e_{k-1}\}| & \text{if } e_i = e_j \wedge \\ & \forall k : j < k < i, e_i \neq e_k \\ \infty & \text{otherwise} \end{cases} \tag{13}$$

Note that this definition of the reuse distance is a variation of the one proposed in earlier work. The revised equation (13) computes the same property, but has to discard successive accesses to the same block. Successive accesses to the same memory block lead to guaranteed cache hits under an evict-on-miss cache replacement policy. Traces are thus collapsed in Altmeyer et al. (2015) to remove all successive accesses to the same memory block. The number of cache misses is not impacted and cache hits can later be accounted for as an additional contribution to the trace. This last step is not straightforward for multi-path programs as the number of guaranteed hits varies on different paths.

Conversely, we define the forward reuse distance $frd(e)$ of an element $e$ as the maximum number of possible evictions before the next access to the same block. If its block is not reused before the end of the trace, the forward reuse distance of an access is defined as $\infty$:

$$frd : \mathbb{E} \times \mathbb{T} \to \mathbb{N} \cup \{\infty\}$$
$$frd(e_i, [e_{i+1}, \ldots, e_m]) = \begin{cases} |\{k | j < k < i \wedge e_k \neq e_{k-1}\}| & \text{if } e_i = e_j, \\ & \forall k : i < k < j, e_i \neq e_k \\ \infty & \text{otherwise} \end{cases} \tag{14}$$

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 $con(e_i, t)$ (Altmeyer and Davis 2014) 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:

$$\hat{P}(e_i^{hit}) = \begin{cases} 0 & con(e_i, t) \geq N \\ \left(\frac{N-1}{N}\right)^{rd(e_i,t)} & \text{otherwise} \end{cases} \tag{15}$$

The cache contention $con(e)$ (Altmeyer and Davis 2014) of element $e$ captures the number of cache blocks which contend with $e$ for space in the cache. It includes all potential hits and the $R$ relevant blocks, denoted $relevant_blocks$, since we have to assume they 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 miss $e_r$ 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:

$$con : \mathbb{E} \times \mathbb{T} \to \mathbb{N} \cup \{\infty\}$$
$$con(e_i, t) = \begin{cases} \infty & \text{if } rd(e_i, t) = \infty \\ |\{e_k | k \in conS(e_i, t) \wedge e_k \notin relevant_blocks\}| + R & \text{otherwise} \end{cases} \tag{16}$$

with

$$\begin{aligned} conS(e_i, t) = \{j \mid e_j \in t \wedge \hat{P}(e_j^{hit}) \neq 0 \wedge k < j < i \wedge e_k \\ = e_i \wedge \forall x : k < x < i, e_i \neq e_x\} \\ \cup \{r \mid rd(e_i, t) \neq 0 \wedge \\ r = min(\{x \mid \hat{P}(e_x^{hit}) = 0 \wedge k < x < i \wedge e_k \\ = e_i \wedge \forall y : k < y < i, e_i \neq e_y\})\} \end{aligned} \tag{17}$$

*Example* We now illustrate the distinction between cache contention and reuse distance in identifying accesses with a null hit probability in (15). Consider the following sequence of accesses, on a 4 line fully-associative cache, where the reuse distance of each access is given as a super-script:

$$a, b, c, b^1, d, f, a^5, b^3, c^5, d^4, f^4$$

All second accesses to blocks $a$, $b$, $c$, $d$, and $f$ have a non-zero chance to hit when considered in isolation. However as highlighted in Altmeyer and Davis (2014), those cannot be simply combined as the hit probability of a block depends on the behaviour of other blocks; the last 5 accesses of the sequence, each accessing a different block, cannot hit at the same time assuming a 4 line cache. The hit probability of an access need to be set to 0 in (15) if enough blocks are inserted in cache since the last access to the same block. Should the reuse distance be considered to identify whether or not an access is a potential hit, the last occurrences of $a$, $c$, $d$, and $f$ would be considered as misses.

Using cache contention, some accesses are assumed to be potential hits, occupying cache space to the detriment of others. Cache contention captures a specific but potential hit/miss scenario the occurrence of which is bounded using each access hit probability in (15). As proven in Altmeyer and Davis (2014), the estimated hit probability of the overall sequence holds. In our example, contention identifies that $a$, $b$, and $c$ can be kept in the cache simultaneously. Using the contention as a super-script, we have:

$$a, b, c, b^1, d, f, a^2, b^2, c^3, d^4, f^4$$

$c^3$ implies that $c$ may be present in cache, assuming only three other blocks may have been kept alongside it, $a$ and $b$ as potential cache hits, and $d$ then replaced by $f$. This assumption regarding $d$ and $f$ is an important difference between contention and the stack distance metric used in LRU cache analysis. Using the stack distance, i.e. the number of different blocks accessed since the last access to $c$, $d$ and $f$ would be regarded as occupying a different line in cache, resulting in a guaranteed miss for $c$. $d^4$ is classified as a miss: $a^2$, $b^2$ and $c^3$ have been identified as potential misses, and $f$ is a miss resulting in the eviction of the fourth and only cache line where $d$ could be held. $f^4$ is similarly classified as a miss.

Note that this definition of contention is an improvement on the one proposed in earlier work. Instead of accounting for each access independently, we account for their accessed blocks instead. The reasoning behind this optimisation is that if an accessed block hits more than once, it does not occupy additional lines. In the previous example, $b$ is only accounted for once in the contention of $a^2$ and $c^3$. The subtle difference lies in (17) where the blocks $e_j$ are accounted for instead of each access $j$ individually ($e_i = e_j$ if they access the same block).

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

$$\hat{\xi}_i(x) = \begin{cases} \hat{P}(e_i^{hit}) & \text{if } x = \mathcal{H} \\ 1 - \hat{P}(e_i^{hit}) & \text{if } x = \mathcal{M} \\ 0 & otherwise \end{cases} \tag{18}$$

An estimated pWCET (Cucu-Grosjean 2013) distribution $\hat{\mathcal{D}}$ of a trace, is an upper-bound on the execution time distribution $\mathcal{D}$ induced by the randomised cache for the trace,[2] 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 $\mathcal{D}$ (López et al. 2008), 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 $e_i$:

$$\hat{\mathcal{D}}(t) = \bigotimes_{e_i \in t} \hat{\mathcal{E}}_i \tag{19}$$

where $\otimes$ represents the convolution of PMFs:

$$(\hat{\xi}_i \otimes \hat{\xi}_j)(x) = \sum_{k=-\infty}^{+\infty} \hat{\xi}_i(k) \cdot \hat{\xi}_j(x-k) \tag{20}$$

The resulting distribution for non-relevant accesses is independent of the relevant blocks considered in the cache during the collecting analysis step. A worst-case is assumed where the $R$ blocks are always kept in cache. The distributions resulting from the two analysis steps, collecting and contention, can therefore be soundly convolved to estimate the execution time of a trace. The pWCET of a trace can then be derived by convolving the execution time distributions produced by the contention, and collecting approaches, as derived from $\hat{\mathcal{D}}^{miss}$.

### 2.3 Discussion: relevance of the model

The SPTA techniques described apply whether the contents of the memory block are instruction(s), data or both. While address computation (Huynh et al. 2011) may not be able to pinpoint the exact target of an access, e.g. for data-dependent requests, relational analysis (Hahn and Grund 2012), 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 assume that there are no inter-task cache conflicts due to preemption, i.e. a run-to-completion semantics with non-preemptable program execution. Concur-

[2] Note the precise execution time distribution is effectively that which would be observed by executing the trace an infinite number of times.

rent cache accesses are also precluded, i.e. we assume a private cache or appropriate isolation (Chiou et al. 2000).

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 (Bourgade et al. 2008). Further, to reduce the pessimism inherent in using a simple bound, particularly for the miss latency, events such as memory refresh can be accounted for as part of higher level schedulability analyses (Atanassov and Puschner 2001; Bhat and Mueller 2011).

## 3 Comparing cache contents

The execution time distribution of a trace in our model depends solely on the behaviour of the cache. The contribution of a cache state to the execution time of a trace thus solely depends on its initial contents. The characterisation of the relation between the initial contents of different caches allows for a comparison of their temporal contribution to the same trace. This section introduces properties and conditions that allow this comparison. They are used in later techniques to improve the selection of cache contents on path convergence, and identify paths with the worst impact on execution time.

An $N$-tuple represents the concrete contents of an $N$-way cache, such that each element corresponds to the block held by a single line. The symbol _ is used to denote an empty line. For each such concrete cache $s$, there is a corresponding abstract cache contents $C$ which holds the exact same set of blocks. $C$ might also capture uncertainty regarding the contents of some lines.

Given cache state $s = \langle l_1, \ldots, l_N \rangle$,[3] $s[l_i = b]$ represents the replacement of memory block or line $l_i$ in cache by memory block $b$. Note that $b$ can only be present once in the cache, $b \in s \Rightarrow s[l_i = b] = s$. $s[-l_i]$ is a shorthand for $s[l_i = _]$ and identifies the eviction of memory block $l_i$ from the cache. $s[l_i = b][l_j = e]$ denotes a sequence of replacements where $b$ first replaces $l_i$ in $s$, then $e$ replaces $l_j$. Two cache states $s$ and $s'$ although not strictly identical may exhibit the same behaviour if they hold the exact same contents, e.g. $\langle a, _ \rangle = \langle _, a \rangle$ are represented using the same abstract contents $\{a\}$. Under the evict-on-miss random replacement policy, there is no correlation between the physical and logical position of a block with respects to the eviction policy.

We distinguish the execution time distribution of trace $t$ using input cache state $s$ with the notation $\mathcal{D}(t, s)$. The execution time distribution of the sequence $[[b], t]$, the concatenation of access $[b]$ to trace $t$, can be expressed as follows:

$$\mathcal{D}([[b], t], s = \langle l_1, \ldots, l_N \rangle) = \begin{cases} \mathcal{H} + \mathcal{D}(t, s) & \text{if } b \in s \\ \mathcal{M} + \sum\limits_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t, s[l_i = b]) & \text{otherwise} \end{cases} \tag{21}$$

[3] We assume a fully-associative cache, but this restriction can be lifted to set-associative caches through the independent analysis of each set.

where the sum of distributions and the product of a distribution with $\frac{1}{N}$ are defined as per (6), and $(\mathcal{L} + \mathcal{D})(x) = \mathcal{L} + \mathcal{D}(x)$ denotes the sum of distribution $\mathcal{D}$ with latency $\mathcal{L}$. Upon a hit, the input cache state $s$ is left unchanged, while evictions occur to make room for the accessed block upon a miss.

The extension of this definition to the concatenation of traces requires the identification of the outcomes of an execution, i.e. the cache state $C$ corresponding to each possible sequence of events, along with its occurrence probability $P$ and execution time distribution $\mathcal{D}$:

$$\mathcal{D}([t_p, t_s], s) = \sum_{(C,P,\mathcal{D}) \in outcomes(t_p,s)} P \cdot (\mathcal{D} \otimes \mathcal{D}(t_s, C)) \tag{22}$$

where $outcomes(t_p, s)$ is the set of cache states produced by the execution of $t_p$ from input cache state $s$ and $\otimes$ is the convolution of distributions.

**Theorem 1** *The eviction of a block from any input cache state $s$ cannot decrease the execution time distribution of any trace $t$, $\mathcal{D}(t, s) \leq \mathcal{D}(t, s[-e])$.*

*Proof* See Appendix. □

**Corollary 1** *In the context of evict-on-miss randomised caches, for any trace, the empty state is the worst initial state over any other input cache state $s$, $\mathcal{D}(t, s) \leq \mathcal{D}(t, \emptyset)$.*

The eviction of a block might trigger additional misses, resulting in a distribution that is no less than the one where the cache contents is left untouched. This provides evidence that the assumption upon a non-relevant access that a block in cache is evicted, as per the update function in (3), is sound. Similarly, the replacement of a block in the cache might trigger additional misses but might also result in additional hits instead upon reuse of the replacing block. The impact of such a behaviour is however bounded.

**Theorem 2** *The replacement of a random block in cache triggers at most one additional hit.*

*The distribution for any trace $t$ from any cache state $s$ is upper-bounded by the distribution for trace $t$ after the replacement of a random block in $s$ and assuming a single hit turns into a miss.*

$$\mathcal{H} + \mathcal{D}(t, s) \leq \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t, s[l_i = e]) \tag{23}$$

*Proof* See Appendix. □

The block selected for eviction impacts the likelihood of those additional latencies suffered during the execution of the subsequent trace. Intuitively, the closer the evicted block is to reuse, the worse the impact of the eviction. We use the forward reuse distance of blocks at the beginning of trace $t$, $frd(b, t)$ as defined in (14), to identify the blocks which are closer to reuse than others.

**Theorem 3** *The replacement of a block in input cache state $s$ by one which is reused later in trace $t$ cannot result in a decreased execution time distribution: $frd(b,t) \leq frd(e,t) \leq \infty \wedge b \in s \wedge e \notin s \Rightarrow \mathcal{D}(t,s) \leq \mathcal{D}(t,s[b=e])$*

*Proof* See Appendix. □

## 4 Application of SPTA to multi-path programs

In this section, we improve upon the state-of-the-art SPTA techniques for traces (Altmeyer and Davis 2014) recapitulated in Sect. 2 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 (6) 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:

$$\bigodot_{t \in \mathcal{T}} \mathcal{D}(t) \tag{24}$$

where we define the $\odot$ operation as follows

$$\odot \colon ((\mathbb{N} \to \mathbb{R}) \times (\mathbb{N} \to \mathbb{R})) \to (\mathbb{N} \to \mathbb{R}) \tag{25}$$

$$\mathcal{D}_a \odot \mathcal{D}_b := \mathcal{D}^H \tag{26}$$

with

$$\mathcal{D}^H(x) = \max\left(\sum_{y \geq x} \mathcal{D}_a(y) - \sum_{y > x} \mathcal{D}^H(y), \sum_{y \geq x} \mathcal{D}_b(y) - \sum_{y > x} \mathcal{D}^H(y), 0\right) \tag{27}$$

The $\odot$ operator computes the least upper-bound of the complementary cumulative distribution (1-CDF) of all its operands (similar to the upper-bound depicted in Fig. 1), a maximum of distributions which is valid irrespective of the path executed at runtime. By construction the following properties hold

$$\mathcal{D}_a \odot \mathcal{D}_b \geq \mathcal{D}_a \wedge \mathcal{D}_a \odot \mathcal{D}_b \geq \mathcal{D}_b \tag{28}$$

$$\mathcal{D}_a \leq \mathcal{D}_b \Rightarrow \mathcal{D}_a \odot \mathcal{D}_b = \mathcal{D}_b \tag{29}$$

Secondly, the number of distinct traces is exponential in the number of control flow divergences, conditional constructs and loop iterations, which means that this naive approach is computationally intractable. 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 relevant and non-relevant blocks respectively, as

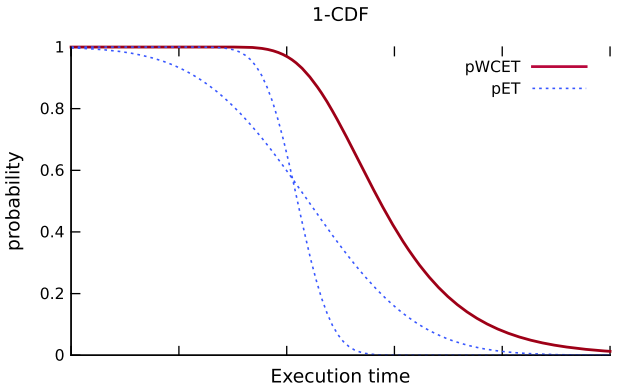



**Fig. 1** Relation between the execution time distribution of different paths (pET) and the pWCET of a program

per the cache collecting approach on traces given by Altmeyer and Davis (2014). 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 relevant blocks and the cache collection, 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.

### 4.1 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 node $v$. A path $\pi$ 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 $\forall i : (v_i, v_{i+1}) \in L$ and defines a corresponding trace. By extension, $[\pi, \pi']$ denotes the path composed of path $\pi$ followed by path $\pi'$. Given a set of nodes $V'$, the symbol $\Pi(V')$ denotes the set of all paths with nodes that are included exclusively 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 the least upper-bound on the execution time distributions (pET) of all possible paths. Hence, $\forall \pi \in \Pi(G), \hat{\mathcal{D}}(G) \geq \mathcal{D}(\pi)$. Figure 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 and post-dominators of node $v_n$ as $dom(v_n)$ and $post\text{-}dom(v_n)$ respectively.

We assume that the program always terminates. Bounded recursion and loop iterations are requirements to ensure this termination property of the analysed application.

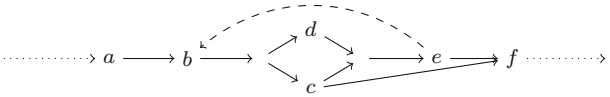



**Fig. 2** Simple do-while loop structure with an embedded conditional. $b$ is the loop head, with its body comprising $\{b, c, d, e\}$ and the $e$ to $b$ edge as the back-edge. $e$ and $c$ are both valid exits

The additional restrictions described below are for the most part tied to the WCET analysis framework (Wilhelm et al. 2008) and not exclusive to the new method. These are reasonable assumptions for the software in critical real-time systems.

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 $V_l \subseteq V$. Considering the example in Fig. 2, $b$ is the head of the loop composed of accesses $V_l = \{b, d, c, e\}$. 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, e.g. 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, iterations which are 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 small 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 target functions must be exact or an over-estimate of the actual ones, so as to avoid unsound estimates which do not take all valid paths into account.

### 4.2 Complete loop unrolling

In the first analysis step, we conceptually transform the control-flow graph into a directed acyclic graph by loop unrolling and function inlining (Muchnick 1997). In contrast to the naive approach of enumerating all possible traces, analysis through complete loop unrolling has linear rather than exponential complexity with the number of loop iterations.

Loop unrolling and function inlining are well-known techniques to improve the precision of data-flow analyses. A complete physical unrolling that removes all back-edges significantly increases the size of the control-flow graph. A virtual unrolling and inlining is instead performed during analysis such that calls and iterations are processed as required by the control flow. The analysis then distinguishes the different call and iteration contexts of a vertex. In either case, the size of the graph explored

during analysis and its complexity scales with the number of accesses in the program under consideration.

Unrolling simplifies the analysis and significantly improves the precision. As opposed to state of the art analyses for deterministic replacement policies (Alt et al. 1996), the analysis of random caches through cache state enumeration does not rely on the computation of a fixpoint. The abstract domain for the analysis is by nature growing with every access since it includes the estimated distribution of misses. Successive iterations increase the probability of blocks in the loop's working set being in the cache, and in turn increase the likelihood of hits in the next iteration. The exhaustive analysis, if not supplemented by other methods, must process all accesses in the program.

We assume in the following that unrolling is performed on all analysed programs. Section 6.4.2 discusses preliminary work to bypass this restriction. The analysis of large loops, with many predicted iterations, can be broken down into the analysis of a single iteration or groups thereof provided a sound upper-bound of the input state is used. The contributions of different segments are then combined to compute that of the complete loop or program. Such an upper-bound input can be derived as an example using cache state compression (Griffin et al. 2014a) to remove low value information. The definition of techniques to exploit the resulting trade-off between precision and analysis complexity is left as future work.

### 4.3 Reuse distance/cache contention on CFG

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

$$rd^G : \mathbb{V} \rightarrow \mathbb{N} \cup \{\infty\} \tag{30}$$

$$rd^G(v) = \max_{\pi=[v_s,\ldots,v]} (rd(v,\pi)) \tag{31}$$

The cache contention is extended accordingly:

$$con^G : \mathbb{V} \rightarrow \mathbb{N} \tag{32}$$

$$con^G(v) = \max_{\pi=[v_s,\ldots,v]} (con(v,\pi)) \tag{33}$$

An upper-bound of both metrics for each access can be computed through a forward data flow analysis. The reuse distance analysis uses the maximum of the possible values on path convergence. Similarly, we lift the definition of the forward reuse distance to control-flow graphs. It can be computed through a backward data flow analysis. The contention for each block at each point in the program is computed through a forward data flow analysis. The computation of the contention relies on the estimation of the set of contending cache blocks. Its analysis domain is more complex than the reuse distance as different sets of contending blocks may arise on different paths. The analysis tracks all such sets from incoming paths, as long as they are conclusive to a

potential cache hit, i.e. all sets are smaller than the associativity of the cache, and not included into each other, i.e. one does not upper-bound the other.

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 maximum operator $\odot$ as the join operator.

### 4.4 Selection of relevant blocks

The selection of relevant blocks in Altmeyer and Davis (2014) also needs to be modified to accommodate for a control-flow graph. Cache state enumeration is only performed for relevant accesses, ensuring more precise analysis results for the selected accesses. Earlier work (Altmeyer and Davis 2014) relied on an absolute set of $R$ relevant blocks for the whole trace. Instead, we only restrict ourselves to at most $R$ relevant blocks at any point in the program. Given a position in the control-flow, the heuristic tracks the $R$ blocks with the shortest lifespan, i.e. the shortest 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.

The $R$ blocks with the smallest lifespan are analysed using the collecting semantics, as they are the most likely to be kept in cache. For each of these blocks $b$, the access prior to $b$ must ensure its insertion in the cache during analysis. As such, the access needs to be marked as relevant, included in the *relevant_accesses* set, and excluded from accesses contributing to contention. The computation of cache contention is modified to account for relevant accesses instead of blocks:

$$con(e_i, t) = \begin{cases} \infty & \text{if } rd(e_i, t) = \infty \\ |\{e_k | k \in conS(e_i, t) \wedge k \notin relevant_accesses\}| + R & \text{otherwise} \end{cases} \tag{34}$$

### 4.5 Approximation of cache states

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 all incoming paths, i.e. it contains the same or degraded information. To capture this property, we introduce the partial ordering $\sqsubseteq$ between a cache state and a set thereof such that $s \sqsubseteq S_b$ implies that $S_b$ holds more pessimistic information than $s$, resulting in more pessimistic timing estimates. We overload this operator to relate sets of cache states where $S_a \sqsubseteq S_b$ implies that $S_b$ holds more pessimistic information than $S_a$. More formally, the $\sqsubseteq$ notation (Peleska and Löding 2008) identifies $S_b$ as an upper-bound of $S_a$ in $2^{\mathbb{CS}}$.

Consider a simple cache state $s = (\{a, b\}, 0.5, \mathcal{D})$. Intuitively, the information represented by $s_a = (\{a\}, 0.5, \mathcal{D})$ is more pessimistic than that captured by $s$, $s \sqsubseteq s_a$.

Conversely, $s_c = (\{a, c\}, 0.5, \mathcal{D})$ holds less pessimistic 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 ($s$) 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. The relation $s \sqsubseteq S$ holds if the set of cache states $S$ approximates cache state $s = (C, P, \mathcal{D})$. In other words, (i) $S$ is as likely to occur, (ii) all blocks known to be in states of $S$ are present in $s$, and (iii) the contribution of $S$ to the pWCET is greater than or equal to the contribution $\mathcal{D}$ of $s$. We formally define $s \sqsubseteq S$ as follows:

$$(C, P, \mathcal{D}) \sqsubseteq S \Rightarrow \left(P = \left(\sum_{(C', P', \mathcal{D}') \in S} P'\right)\right) \wedge \left(\forall (C', P', \mathcal{D}') \in S, C \supseteq C' \wedge \mathcal{D} \leq \mathcal{D}'\right) \tag{35}$$

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:

$$S_a \sqsubseteq S_b \Rightarrow \exists F : \mathbb{CS} \rightarrow 2^{\mathbb{CS}}, (\forall s \in S_a, s \sqsubseteq F(s)) \wedge S_b = \biguplus_{s \in S_a} F(s) \tag{36}$$

A join function $\sqcup$ is valid if given any set of cache states $S_a \in 2^{\mathbb{CS}}$ and $S_b \in 2^{\mathbb{CS}}$, $S_a \sqsubseteq (S_a \sqcup S_b)$ and $S_b \sqsubseteq (S_a \sqcup S_b)$. An optimal join function $\sqcup$ should return the least upper-bound of its parameters, i.e. the smallest state which upper-bounds all its inputs. Our definition of the $\sqsubseteq$ operator is however independent of the executed path: $S_a$ and $S_b$ may admit multiple upper-bounds incomparable to each other. The definition of an optimal join function would require a more complete ordering, taking into account the upcoming sequence of accesses to order sets of cache states depending on the likelihood their contents are reused. Optimality would still be challenged in multiple path applications where different paths stem from the join.

To prove over-approximation results in more pessimistic timing estimates, we derive the execution time distribution of a trace $t$ using the set of input cache states $S$ from its definition for a single state and the concatenation of traces respectively in (21) and (22):

$$\mathcal{D}(t, S) = \sum_{(C', P', \mathcal{D}') \in S} P' \cdot \left(\mathcal{D}' \otimes \mathcal{D}(t, C')\right) \tag{37}$$

where the sum of distributions and the product of a distribution with $P$ are defined as per (6), and $\otimes$ is the convolution of distributions.

The definition of over-approximations and their contribution to the execution time distribution of a trace relies on the merge $\uplus$ and convolution $\otimes$ operators defined respectively in (6) and (20). Both offer properties used in the evaluation of the con-

tribution of their operands. The convolution operator preserves the relative ordering between its inputs, and the merge operation adds the contribution of its operands.

**Lemma 1** *The convolution operation preserves the ordering between execution time distributions:*

$$\mathcal{D} \leq \mathcal{D}' \Rightarrow \mathcal{D} \otimes \mathcal{A} \leq \mathcal{D}' \otimes \mathcal{A}$$

*Proof* See Appendix. □

**Lemma 2** *The contributions of merged sets of cache states S and A is the sum of their individual contributions:*

$$\forall t, \mathcal{D}(t, S) + \mathcal{D}(t, A) = \mathcal{D}(t, S \uplus A)$$

*Proof* See Appendix. □

**Theorem 4** *The over-approximation $S_b$ of a set of cache states $S_a$ holds more pessimistic information than $S_a$,*

$$\forall t, S_a \sqsubseteq S_b \Rightarrow \mathcal{D}(t, S_a) \leq \mathcal{D}(t, S_b)$$

*Proof* The relation between $S_b$ and $S_a$, defined in (36), implies the existence of an approximation function $F$ for the cache states in $S_a$ such that:

$$(\forall s \in S_a, s \sqsubseteq F(s)) \wedge S_b = \biguplus_{s \in S_a} F(s) \tag{38}$$

From (38) and (35), we know that each cache contents $C'$ in the approximation $F(s) = (C', P', \mathcal{D}')$ is included in the contents $C$ of cache state $s = (C, P, \mathcal{D})$. $C'$ can thus be derived by evicting blocks from $C$. From Theorem 1 we can infer:

$$\forall (C, P, \mathcal{D}) \in S_a, \forall (C', P', \mathcal{D}') \in F((C, P, D)), \mathcal{D}(t, C) \leq \mathcal{D}(t, C') \tag{39}$$

From Lemma 1, we can convolve both sides of the inequality with the same distribution $\mathcal{D}$:

$$\forall (C, P, \mathcal{D}) \in S_a, \forall (C', P', \mathcal{D}') \in F((C, P, D)), \mathcal{D} \otimes \mathcal{D}(t, C) \leq \mathcal{D} \otimes \mathcal{D}(t, C') \tag{40}$$

Approximate distributions $\mathcal{D}'$ in $F(s)$ are also by definition greater than their counterpart $\mathcal{D}$ in $s$. We can similarly factor $\mathcal{D}(t, C)$ into both sides of inequality $\mathcal{D} \leq \mathcal{D}'$:

$$\forall (C, P, \mathcal{D}) \in S_a, \forall (C', P', \mathcal{D}') \in F((C, P, D)), \mathcal{D} \otimes \mathcal{D}(t, C') \leq \mathcal{D}' \otimes \mathcal{D}(t, C') \tag{41}$$

By transitivity of the $\leq$ operator, we can compare the contribution to the execution time distribution of $s = (C, P, \mathcal{D})$ and each of the corresponding approximations in

$F((C, P, D))$. That is a comparison between the leftmost term in (40) and rightmost term in (41) through $\mathcal{D} \otimes \mathcal{D}(t, C')$:

$$\forall (C, P, \mathcal{D}) \in S_a, \forall (C', P', \mathcal{D}') \in F((C, P, D)), \mathcal{D} \otimes \mathcal{D}(t, C) \leq \mathcal{D}' \otimes \mathcal{D}(t, C') \tag{42}$$

We multiply both sides of the inequality by the positive occurrence probability $P'$:

$$\begin{aligned} &\forall (C, P, \mathcal{D}) \in S_a, \forall (C', P', \mathcal{D}') \in F((C, P, D)), P' \cdot (\mathcal{D} \otimes \mathcal{D}(t, C)) \\ &\quad \leq P' \cdot (\mathcal{D}' \otimes \mathcal{D}(t, C')) \end{aligned} \tag{43}$$

The property holds for each approximation in $F(s)$ and can be extended to their sum:

$$\begin{aligned} \forall (C, P, \mathcal{D}) \in S_a, &\sum_{(C', P', \mathcal{D}') \in F((C, P, D))} P' \cdot (\mathcal{D} \otimes \mathcal{D}(t, C)) \\ &\leq \sum_{(C', P', \mathcal{D}') \in F((C, P, D))} P' \cdot \mathcal{D}' \otimes \mathcal{D}(t, C') \end{aligned} \tag{44}$$

From (35) and (38), a state $s \in S_a$ has the same occurrence probability as its approximation $F(s)$:

$$\forall (C, P, \mathcal{D}) \in S_a, P \cdot (\mathcal{D} \otimes \mathcal{D}(t, C)) \leq \sum_{(C', P', \mathcal{D}') \in F((C, P, D))} P' \cdot \mathcal{D}' \otimes \mathcal{D}(t, C') \tag{45}$$

Both terms of the inequality correspond to the contribution of a set of cache states to the execution time distribution of trace $t$ as per (37):

$$\forall (C, P, \mathcal{D}) \in S_a, P \cdot (\mathcal{D} \otimes \mathcal{D}(t, C)) \leq \mathcal{D}(t, F((C, P, D))) \tag{46}$$

The property holds for any cache state $s \in S_a$ and can be extended to their sum such that:

$$\sum_{(C, P, D) \in S_a} P \cdot (\mathcal{D} \otimes \mathcal{D}(t, C)) \leq \sum_{s \in S_a} \mathcal{D}(t, F(s)) \tag{47}$$

From Lemma 2, the inequality also holds for the merge $\uplus$ across $S_a$ of the approximations $F(s)$:

$$\sum_{(C, P, D) \in S_a} P \cdot (\mathcal{D} \otimes \mathcal{D}(t, C)) \leq \mathcal{D}\left(t, \biguplus_{s \in S_a} F(s)\right)$$

By definition of $S_b$ in (38) and the application of (37) to $S_a$, we conclude that:

$$\forall t \in \mathbb{T}, \mathcal{D}(t, S_a) \leq \mathcal{D}(t, S_b)$$

□

The $\sqsubseteq$ relation defines a partial ordering between two sets of cache states $S_a$ and $S_b$. Namely, $S_a \sqsubseteq S_b$ implies that $S_b$ holds more pessimistic information than $S_a$. In other words, the execution of any trace from $S_b$ results in a larger execution time distribution than the execution of the same trace from $S_a$. This provides sufficient ground for the definition of a sound join operation, one that upper-bounds the upcoming contribution of cache states coming from different paths.

### 4.6 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 memory blocks $\mathbb{M}^{S_a \cap S_b}$, and then restrict $S_a$ and $S_b$ to this set:

$$\mathbb{M}^{S_a \cap S_b} = \left( \bigcup_{(C_a, P_a, \mathcal{D}_a) \in S_a} C_a \right) \cap \left( \bigcup_{(C_b, P_b, \mathcal{D}_b) \in S_b} C_b \right) \tag{48}$$

$$S'_a = \biguplus \{(C_a \cap \mathbb{M}^{S_a \cap S_b}, P_a, \mathcal{D}_a) | (C_a, P_a, \mathcal{D}_a) \in S_a\} \tag{49}$$

$$S'_b = \biguplus \{(C_b \cap \mathbb{M}^{S_a \cap S_b}, P_b, \mathcal{D}_b) | (C_b, P_b, \mathcal{D}_b) \in S_b\} \tag{50}$$

$S'_a$ 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 $H$ 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 $\mathcal{D}_b$:

$$H = \{(C, \min(P_a, P_b), \mathcal{D}_a \odot \mathcal{D}_b) | (C, P_a, \mathcal{D}_a) \in S'_a \wedge (C, P_b, \mathcal{D}_b) \in S'_b \wedge C \neq \emptyset\} \tag{51}$$

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 $S'_b$:

$$\begin{aligned} \hat{H}_a = {} & \{(\emptyset, P_a, \mathcal{D}_a) | (C, P_a, \mathcal{D}_a) \in S'_a \wedge C \neq \emptyset \wedge \nexists (P_b, \mathcal{D}_b), (C, P_b, \mathcal{D}_b) \in S'_b\} \\ & \uplus \{(\emptyset, P_a - P_b, \mathcal{D}_a) | (C, P_a, \mathcal{D}_a) \in S'_a \wedge (C, P_b, \mathcal{D}_b) \in S'_b \wedge C \neq \emptyset \wedge P_a > P_b\} \\ & \uplus \{(\emptyset, P, \mathcal{D}) | (\emptyset, P, \mathcal{D}) \in S'_a\} \end{aligned} \tag{52}$$

$$\begin{aligned} \hat{H}_b = {} & \{(\emptyset, P_b, \mathcal{D}_b) | (C, P_b, \mathcal{D}_b) \in S'_b \wedge C \neq \emptyset \wedge \nexists (P_a, \mathcal{D}_a), (C, P_a, \mathcal{D}_a) \in S'_a\} \\ & \uplus \{(\emptyset, P_b - P_a, \mathcal{D}_b) | (C, P_b, \mathcal{D}_b) \in S'_b \wedge (C, P_a, \mathcal{D}_a) \in S'_a \wedge C \neq \emptyset \wedge P_b > P_a\} \\ & \uplus \{(\emptyset, P, \mathcal{D}) | (\emptyset, P, \mathcal{D}) \in S'_b\} \end{aligned} \tag{53}$$

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

$$\hat{H} = \{(\emptyset, P, \mathcal{D}_a \odot \mathcal{D}_b) | (\emptyset, P, \mathcal{D}_a) \in \hat{H}_a \wedge (\emptyset, P, \mathcal{D}_b) \in \hat{H}_b\} \tag{54}$$

$H \uplus \hat{H}$ is a safe over-approximation of both $S'_a$ and $S'_b$ with regards to the ordering defined in (36). We can define a function $F_a$, which gives an over-approximation of each element of $S'_a$ such that $(H \uplus \hat{H}) = \uplus_{s_a \in S'_a} F_a(s_a)$, as follows:

$$F_a(C, P_a, \mathcal{D}_a) = \begin{cases} \{(\emptyset, P_a, \mathcal{D}_a)\} & \text{if } C = \emptyset \\ \{(\emptyset, P_a, \mathcal{D}_a)\} & \text{if } \nexists (C, P_b, \mathcal{D}_b) \in S'_b \\ \{(C, P_b, \mathcal{D}_a \odot \mathcal{D}_b)\} \cup \{(\emptyset, P_a - P_b, \mathcal{D}_a)\} & \text{if } \exists (C, P_b, \mathcal{D}_b) \in S'_b \wedge P_a > P_b \\ \{(C, P_a, \mathcal{D}_a \odot \mathcal{D}_b)\} & \text{if } \exists (C, P_b, \mathcal{D}_b) \in S'_b \wedge P_a \leq P_b \end{cases} \tag{55}$$

We define the over-approximation function $F_b$ for elements in $S'_b$ analogously:

$$F_b(C, P_b, \mathcal{D}_b) = \begin{cases} \{(\emptyset, P_b, \mathcal{D}_b)\} & \text{if } C = \emptyset \\ \{(\emptyset, P_b, \mathcal{D}_b)\} & \text{if } \nexists (C, P_a, \mathcal{D}_a) \in S'_a \\ \{(C, P_a, \mathcal{D}_a \odot \mathcal{D}_b)\} \cup \{(\emptyset, P_b - P_a, \mathcal{D}_b)\} & \text{if } \exists (C, P_a, \mathcal{D}_a) \in S'_a \wedge P_b > P_a \\ \{(C, P_b, \mathcal{D}_a \odot \mathcal{D}_b)\} & \text{if } \exists (C, P_a, \mathcal{D}_a) \in S'_a \wedge P_b \leq P_a \end{cases} \tag{56}$$

The join operation is defined as follows:

$$S_a \bigsqcup S_b = H \uplus \hat{H} \tag{57}$$

*Example 1* 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 $\pi_b = [b, c, a, d]$. The resulting set of cache states are denoted by $S_a$ and $S_b$ respectively.

| $S_a$ | $S_b$ |
|---|---|
| $(\{a, b, c\}, 24/64, \mathcal{D})$ | $(\{a, b, c, d\}, 6/64, \mathcal{D})$ |
| | $(\{a, b, d\}, 12/64, \mathcal{D})$ |
| | $(\{a, c, d\}, 18/64, \mathcal{D})$ |
| | $(\{b, c, d\}, 6/64, \mathcal{D})$ |
| $(\{a, c\}, 12/64, \mathcal{D})$ | $(\{a, d\}, 12/64, \mathcal{D})$ |
| $(\{b, c\}, 24/64, \mathcal{D})$ | $(\{b, d\}, 3/64, \mathcal{D})$ |
| | $(\{c, d\}, 6/64, \mathcal{D})$ |
| $(\{c\}, 4/64, \mathcal{D})$ | $(\{d\}, 1/64, \mathcal{D})$ |

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

| $S'_a$ | $S'_b$ |
|---|---|
| $(\{a, b, c\}, 24/64, \mathcal{D})$ | $(\{a, b, c\}, 6/64, \mathcal{D})$ |
| | $(\{a, b\}, 12/64, \mathcal{D})$ |
| $(\{a, c\}, 12/64, \mathcal{D})$ | $(\{a, c\}, 18/64, \mathcal{D})$ |
| $(\{b, c\}, 24/64, \mathcal{D})$ | $(\{b, c\}, 6/64, \mathcal{D})$ |
| | $(\{a\}, 12/64, \mathcal{D})$ |
| | $(\{b\}, 3/64, \mathcal{D})$ |
| $(\{c\}, 4/64, \mathcal{D})$ | $(\{c\}, 6/64, \mathcal{D})$ |
| | $(\{\}, 1/64, \mathcal{D})$ |

The set of common cache states $H$, with their minimal, guaranteed probability, is defined as $H = \{(\{a, b, c\}, 6/64, \mathcal{D}), (\{a, c\}, 12/64, \mathcal{D}), (\{b, c\}, 6/64, \mathcal{D}), (\{c\}, 4/64, \mathcal{D})\}$.

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:

| $\hat{C}_a$ | $\hat{C}_b$ |
|---|---|
| $(\{\}, 18/64, \mathcal{D})$ | $(\{\}, 12/64, \mathcal{D})$ |
| | $(\{\}, 6/64, \mathcal{D})$ |
| $(\{\}, 18/64, \mathcal{D})$ | $(\{\}, 12/64, \mathcal{D})$ |
| | $(\{\}, 3/64, \mathcal{D})$ |
| | $(\{\}, 2/64, \mathcal{D})$ |
| | $(\{\}, 1/64, \mathcal{D})$ |

Hence, the result of the join operation on the convergence of paths $\pi_a$ and $\pi_b$ is given by:

| $S_a \bigsqcup S_b$ |
|---|
| $(\{a, b, c\}, 6/64, D)$ |
| $(\{a, c\}, 12/64, D)$ |
| $(\{b, c\}, 6/64, D)$ |
| $(\{c\}, 4/64, D)$ |
| $(\{\}, 36/64, D)$ |

## 5 Improving on the join operation

The basic join operation introduced in the previous section focuses on the conservation of common cache states. Others, because their contents differ or their occurrence is bounded on alternative paths, are merged into the empty state. This results in a safe estimate of the information gathered from different paths. Yet, the method exhibits some limitations with regards to the information it conserves; the probability of occurrence of some blocks in cache, which we refer to as their capacity, is lost during the join process. We introduce a join function based on conserving this additional capacity

of cache states. The function degrades the information about the presence of blocks in a cache to allocate, in a sound manner, its occurrence probability to a more pessimistic state. We first present a ranking heuristic used to identify the cache states to which capacity should be allocated to in priority in Sect. 5.1. The improved capacity-conserving join is itself presented in Sect. 5.2.

### 5.1 Ranking cache states

The ordering $\sqsubseteq$ introduced in Sect. 4.5 allows for the comparison of some cache states to each other irrespective of the upcoming trace of memory accesses. It is however a partial ordering and only compares two states with similar or included cache contents. As illustrated in Theorem 3, ordering the contribution of cache contents which do not include each other requires the consideration of future accesses as captured by their forward reuse distance. The definition of an optimal join operation, through the optimal allocation of capacity to cache states, should ideally minimise the execution time on the worst-case path. However, multiple, incomparable paths would need to be considered of which the worst-case is unknown. We instead rely on a heuristic to prioritise the most beneficial cache states through a ranking system.

The proposed ranking is based on a two sieves approach: (i) the number of useful blocks in cache are first compared with more blocks ranking higher, (ii) cache states are then compared based on their expected hit probability. As a result, we can compare the ranks of two cache states:

$$C \preceq_{rank} C' = \begin{cases} \texttt{true} & \left|\{e | e \in C \wedge frd^G_{min}(e) \neq \infty\}\right| \leq \left|\{e | e \in C' \wedge frd^G_{min}(e) \neq \infty\}\right| \\ \texttt{true} & \sum_{e \in C} \hat{P}(e^{hit}) \leq \sum_{e \in C'} \hat{P}(e^{hit}) \wedge \\ & \left|\{e | e \in C \wedge frd^G_{min}(e) \neq \infty\}\right| = \left|\{e | e \in C' \wedge frd^G_{min}(e) \neq \infty\}\right| \\ \texttt{false} & otherwise \end{cases} \tag{58}$$

The first sieve prioritises cache states whose contents are likely to include others and hold more information. As per Theorem 1, the loss of information in a cache state cannot decrease the execution time distribution of an upcoming trace of accesses, implicitly $C \subseteq C' \Rightarrow C \preceq_{rank} C'$. The sum of their blocks' hit probabilities settles the rank of same-sized cache states, with a higher sum resulting in a higher rank. Each cache state is reduced to the minimum forward reuse distances of the blocks it holds. Those are used to estimate the corresponding hit probabilities of upcoming accesses by adapting the formula proposed in earlier approaches:

$$\hat{P}(e^{hit}) = \begin{cases} 0 & frd^G_{min}(e) \geq N \\ \left(\frac{N-1}{N}\right)^{frd^G_{min}(e)} & otherwise \end{cases} \tag{59}$$

It would seem intuitive to rely solely on the reuse distances of the blocks held in a cache to define its rank. Yet, a block in the cache may have a low minimum forward reuse distance, increasing its rank, but be reused solely on a path where no other block is reused. To reduce the complexity of the heuristic, it does not distinguish between

the different subsequent paths and so captures and compares only the best possible reuse patterns, even though in some cases these may be optimistic. The forward reuse distances of blocks in the two states might interleave, like the upcoming accesses to these blocks, or vary depending on the considered subsequent path. Theorem 3 on the impact of a replacement on execution time cannot be used in such a context. Some cache states may be beneficial on a specific path, but be outranked on others. This prevents the direct comparison of cache states in the general case.

Our aim is to improve precision in the pWCET estimate, hence the heuristic aims to preserve capacity for cache blocks that will upon their next access result in a high probability of a cache hit. This happens, at least on some forward paths, for blocks with a small forward re-use distance. Preserving capacity for blocks with a larger forward re-use distance would likely result in a smaller probability of a cache hit and a more pessimistic overall pWCET estimate. (Note the ranking is only a heuristic and we do not claim that it makes optimal choices.)

### 5.2 Capacity conserving join

The join operator introduced earlier may result in lost capacity if the contents of states on alternative paths do not exactly match. Consider states $\{a, b, e\}$ and $\{b, c, e\}$ respectively in $S'_a$ and $S'_b$ along with others. They both include states $\{b, e\}$, $\{b\}$, $\{e\}$ and $\emptyset$ and their capacity could be allocated to whichever is the highest ranking one. $\{a, e\}$ on the other hand is a valid approximation of $\{a, b, e\}$ in which it is included, but does not approximate $\{b, c, e\}$.

The capacity conserving join, to reduce waste, considers the cache states included in states from either incoming path, $S'_a$ and $S'_b$, by decreasing rank. Each considered cache state $C$ is allocated as much of the remaining capacity of the states $C_a$ (respectively $C_b$) it approximates in $S'_a$ (resp. $S'_b$) as possible. The capacity that can be allocated to $C$ is bounded by the minimum cumulative capacity of the states it approximates in $S'_a$ and $S'_b$. We also ensure that the overall contribution of a state $C_a$ or $C_b$ to state $C$ does not exceed its capacity. This is a requirement for the resulting $C$ to be a valid approximation as per the $\sqsubseteq$ operator defined in Sect. 4.5. Algorithm 1 outlines the join process, and we further illustrate it with a simple example.

*Example 2* Consider the previous example (Example 1) after the cache states have been reduced to only their common blocks in lines 1–3. All cache states included in $S'_a$ ($\{a, b, c\}$, $\{a, b\}$, $\{b\}$, etc.) are present in $S'_b$ (line 4). Assuming the upcoming sequence of accesses is $[a, c, b]$, the considered states ordered by decreasing rank are:

**Algorithm 1** $S_a \bigsqcup^{capa} S_b$

```
   //Reduce cache states to common blocks, as per (48)
1: blocks := CommonBlocks(S_a, S_b)
2: S'_a = ⊎{(E ∩ blocks, P, D)|(E, P, D) ∈ S_a}
3: S'_b = ⊎{(E ∩ blocks, P, D)|(E, P, D) ∈ S_b}
   //Iterate over remaining states
4: R := StatesIncludedIn(S'_a)
5: R := R ∩ StatesIncludedIn(S'_b)
6: for C in OrderByDecreasingRank(R) do
     //Compute available capacity for C
7:   capacity_a := {(C_a, P_a, D_a)|(C_a, P_a, D_a) ∈ S'_a ∧ C ⊆ C_a}
8:   prob_a := Σ_{(_,P_a,_)∈capacity_a} P_a
9:   capacity_b := {(C_b, P_b, D_b)|(C_b, P_b, D_b) ∈ S'_b ∧ C ⊆ C_b}
10:  prob_b := Σ_{(_,P_b,_)∈capacity_b} P_b
     //Compute resulting capacity for C
11:  p := min(prob_a, prob_b)
12:  prob_a := p
13:  prob_b := p
14:
     //Pick capacity from states in S'_a (up to p)
15:  d_a := EmptyDistribution()
16:  while prob_a > 0 ∧ (C_a, P_a, D_a) in OrderByIncreasingRank(capacity_a) do
17:    r := min(P_a, prob_a)
18:    d_a := d_a + r · D_a
19:    P_a := P_a − r
20:    prob_a := prob_a − r
21:  end while
     //Pick capacity from states in S'_b (up to p)
22:  [...]
23:
     //Save resulting state
24:  states := states ∪ {C, p, d_a ⊙ d_b}
25: end for
   //Merge remaining states into the empty state, as per (52)
26: states := states ∪ EmptyAndMerge(S'_a, S'_b)
27: return states
```

| $S'_a$ | $S'_b$ |
|---|---|
| $(\{a, b, c\}, 24/64, \mathcal{D})$ | $(\{a, b, c\}, 6/64, \mathcal{D})$ |
| $(\{a, c\}, 12/64, \mathcal{D})$ | $(\{a, c\}, 18/64, \mathcal{D})$ |
| | $(\{a, b\}, 12/64, \mathcal{D})$ |
| $(\{b, c\}, 24/64, \mathcal{D})$ | $(\{b, c\}, 6/64, \mathcal{D})$ |
| | $(\{a\}, 12/64, \mathcal{D})$ |
| $(\{c\}, 4/64, \mathcal{D})$ | $(\{c\}, 6/64, \mathcal{D})$ |
| | $(\{b\}, 3/64, \mathcal{D})$ |
| | $(\{\}, 1/64, \mathcal{D})$ |

The first iteration of the capacity conserving join focuses on $\{a, b, c\}$ which no other state can provide capacity to. As a consequence, after the first iteration, $states_1 = \{(\{a, b, c\}, 6/64, \mathcal{D})\}$. In $S'_a$, the state has a remaining capacity of $\frac{18}{64}$ which could be used to accommodate any of the contents of size 2 or less (i.e. $\{a, c\}$, $\{a, b\}$, $\{b, c\}$,

etc.). In particular, during the second loop iteration when contributions to the capacity of $\{a, c\}$ are gathered from $S'_a$ and $S'_b$ (lines 7–10), we have:

| $contribution_a$ | $contribution_b$ |
|---|---|
| $(\{a, b, c\}, 18/64, \mathcal{D})$ | |
| $(\{a, c\}, 12/64, \mathcal{D})$ | $(\{a, c\}, 18/64, \mathcal{D})$ |

The presence of both $a$ and $c$, captured by state $\{a, c\}$, can therefore be guaranteed with probability $\frac{18}{64}$ on both paths. The capacity of states in $S'_a$ and $S'_b$ is decreased accordingly (lines 15–21). Capacity is first picked from the lowest ranking states, such that in our example $\{a, b, c\} \in S_a$ still has a remaining capacity of $\frac{12}{64}$ (the $\frac{18}{64}$ allocated to $\{a, c\}$ minus the contribution of the lower ranking $\{a, c\}$ in $S_a$, $\frac{12}{64}$).

During this step, the execution time distribution obtained through the combination of the contributors' distributions is also computed in $d_a$ (see line 18) and $d_b$ respectively for $S'_a$ and $S'_b$. An upper-bound of $d_a$ and $d_b$ is used when computing the resulting distribution for the conserved state (line 24). After the second iteration of the algorithm, $states_2 = states_1 \cup \{(\{a, c\}, 12/64, \mathcal{D})\}$.

Once all states have been explored, the remaining capacity is gathered into the empty state (line 26). The conserved contents are:

| $states = S_a \sqcup^{capa} S_b$ |
|---|
| $(\{a, b, c\}, 6/64, \mathcal{D})$ |
| $(\{a, c\}, 18/64, \mathcal{D})$ |
| $(\{a, b\}, 12/64, \mathcal{D})$ |
| $(\{b, c\}, 6/64, \mathcal{D})$ |
| $(\{a\}, 0/64, \mathcal{D})$ |
| $(\{c\}, 6/64, \mathcal{D})$ |
| $(\{b\}, 3/64, \mathcal{D})$ |
| $(\{\}, 13/64, \mathcal{D})$ |

Keeping only states with a non-null occurrence probability, the capacity conserving join results in:

| $states = S_a \sqcup^{capa} S_b$ |
|---|
| $(\{a, b, c\}, 6/64, \mathcal{D})$ |
| $(\{a, c\}, 18/64, \mathcal{D})$ |
| $(\{a, b\}, 12/64, \mathcal{D})$ |
| $(\{b, c\}, 6/64, \mathcal{D})$ |
| $(\{c\}, 6/64, \mathcal{D})$ |
| $(\{b\}, 3/64, \mathcal{D})$ |
| $(\{\}, 13/64, \mathcal{D})$ |

Compare the resulting contents to that of the previously introduced join operation repeated for convenience:

| $S_a \bigsqcup S_b$ |
|---|
| $(\{a, b, c\}, 6/64, D)$ |
| $(\{a, c\}, 12/64, D)$ |
| $(\{b, c\}, 6/64, D)$ |
| $(\{c\}, 4/64, D)$ |
| $(\{\}, 36/64, D)$ |

The solution resulting from the application of $\sqcup^{capa}$ dominates that of the previously introduced join operation. Indeed, a state $C$ can only accommodate soundly for itself or a state it includes. With the proposed ranking heuristic this corresponds to a lower ranking state which the algorithm explores after $C$ itself. The capacity of $C$ is first used for $C$ in the algorithm. As a consequence, the capacity allocated to a state is at least its minimum capacity in $S_a$ or $S_b$, e.g. $\frac{12}{64}$ for $\{a, c\}$. This minimum is the capacity that was allocated to the state in the previous join implementation. Different ranking heuristics could potentially lose this dominance relation.

The capacity join further keeps the same timing information as the standard $\sqcup$ operation. The combined distributions and their weights are the same, but attached as a result of the operation to different, less pessimistic cache states. The same fragment of distribution in the standard operation will account for fewer or the same amount of misses using the capacity join.

## 6 Worst-case path reduction

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 analysis 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. This improves the precision of the analysis.

Worst-case execution path (WCEP) reduction includes a set of varied modifications: empty conditions removal, worst-case loop unrolling, and simple path elimination. They apply on the logical level, during analysis, and unlike path upper-bounding approaches (Kosmidis et al. 2014) do not require modifications of the object or source code for pWCET computation.

### 6.1 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 $\pi_a$ is included in $\pi_b$. The former path can be split into sub-paths $\pi_S = [a, b, c]$ and $\pi_E = [e]$, such that $\pi_a = [\pi_S, \pi_E]$. $\pi_b$ can then be expressed as the interleaving of $\pi_S$ and $\pi_E$ with $\pi_V = [d, a]$, i.e. $\pi_b = [\pi_S, \pi_V, \pi_E]$. Similarly, $\pi_b$ includes $[a, c, d, e]$, but not $[b, a, c]$.

**Definition 1** (*Including path*) Let $\pi_a$ and $\pi_b$ be two paths, such that $\pi_a$ is the concatenation of two sub-paths $\pi_S$ and $\pi_E$: $\pi_a = [\pi_S, \pi_E]$. The inclusion of $\pi_a$ in $\pi_b$, 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 $\pi_E \neq \pi'_E$

**Theorem 5** *The execution time distribution of a path $\pi$ prefixed by an access to block $b$ upper-bounds that of path $\pi$, $\mathcal{D}(\pi, s) + \mathcal{H} \leq \mathcal{D}([[b], \pi], s)$.*

*Proof* As per (21), the property trivially holds if $[b]$ is a hit, $\pi$ executes starting from the same cache state $s$ in both cases. We focus on the case where $[b]$ is a miss from $s$. This results in $N$ possible cache states $s[l_i = b]$ such that, thanks to Theorem 2:

$$\mathcal{H} + \mathcal{D}(\pi, s) \leq \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(\pi, s[l_i = b]) \tag{60}$$

$$\mathcal{H} + \mathcal{D}(\pi, s) \leq \mathcal{D}([[b], \pi, s) \tag{61}$$

□

For the sake of readability, we omit in the following the cache state $s$ when comparing the execution time distributions of two paths in the following; two paths are always compared using the same input cache state, $\mathcal{D}(\pi) \leq \mathcal{D}(\pi') \Leftrightarrow \mathcal{D}(\pi, s) \leq \mathcal{D}(\pi', s)$.

**Theorem 6** *The execution time distribution of a path $\pi_a$ prefixed by path $\pi_s$ upper-bounds that of path $\pi_a$ alone, $\forall \pi_s, \pi_a, \mathcal{D}(\pi_a) \leq \mathcal{D}([\pi_s, \pi_a])$.*

*Proof* From Theorem 5, we know that $\mathcal{D}(\pi_a) \leq \mathcal{D}([[v_n], \pi_a])$ which can be extended to $\mathcal{D}(\pi_a) \leq \mathcal{D}([[v_1, v_2, \ldots, v_n], \pi_a])$ since $\mathcal{D}([[v_2, \ldots, v_n], \pi_a]) \leq \mathcal{D}([[v_1, v_2, \ldots, v_n], \pi_a])$ and so on. The relation holds for prefixes of arbitrary lengths. □

**Theorem 7** (Included path ordering) *If $\pi_a$ is included in $\pi_b$, then the execution time distribution of $\pi_b$ is greater than or equal to that of $\pi_a$, $\pi_a \trianglelefteq \pi_b \Rightarrow \mathcal{D}(\pi_a) \leq \mathcal{D}(\pi_b)$*

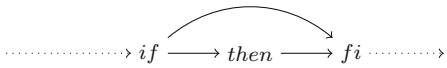



**Fig. 3** Simple if-then conditional structure. The edge from *if* to $fi$, through the empty *else* case, can be removed for pWCET estimation

*Proof* We prove this property by induction.

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 Theorem 6, we know that $\mathcal{D}(\pi_E) \leq \mathcal{D}([\pi_V, \pi_E])$.

The execution of $\pi_S$ cannot be impacted by accesses in either $\pi_E$ or $\pi_V$. It is therefore the same on both paths $\pi_a$ and $\pi_b$. As proved in Theorem 6, whatever cache state is left by the execution of $\pi_S$, the execution time distribution of $[\pi_V, \pi_E]$ is either greater than or equal to that of $\pi_E$. Therefore, $\mathcal{D}(\pi_a) \leq \mathcal{D}(\pi_b)$.

Inductive step: Let us assume $\pi_a = [\pi_S, \pi_E]$ and $\pi'_E$ is 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 Theorem 5, 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 $\pi_S$ is not impacted by accesses in either $\pi_V$, $\pi_E$, or $\pi'_E$ and is the same in $\pi_a$ and $\pi_b$, 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 $\Pi$ is a path-included set of $\Pi^\circ$ if each path in $\Pi$ is included in a corresponding path in $\Pi^\circ$, $\Pi \trianglelefteq \Pi^\circ \Rightarrow \forall \pi \in \Pi, \exists \pi^\circ \in \Pi^\circ, \pi \trianglelefteq \pi^\circ$. As a consequence, for each path $\pi \in \Pi$, there is a path in $\Pi^\circ$ the actual pWCET of which also upper-bounds the execution time distribution of $\pi$. The actual pWCET of $\Pi^\circ$ is thus an upper-bound on the execution time distributions of all paths in $\Pi$, $\forall \pi \in \Pi, \mathcal{D}(\Pi^\circ) \geq \mathcal{D}(\pi)$. As the estimated pWCET of a path $\hat{\mathcal{D}}(\pi)$ is an upper-bound on its execution time distribution, $\mathcal{D}(\pi') \leq \hat{\mathcal{D}}(\pi')$, it is sufficient to perform the pWCET analysis of a CFG $G$ on a reduced set of paths which path-includes the set $\Pi(G)$.

### 6.2 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.

Figure 3 gives an example of this for an *if-then* construct with an empty *else* branch. At point $fi$ in the program, 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. By removing the edge corresponding to the empty branch we remove this source of pessimism.

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 $v_i$. The notion of post-dominators, as an example, can be used to simply capture a subset of those empty branches. In Fig. 3, any path to the program exit through *if* or *then* will traverse $fi$, which post-dominates both *if* and *then*. More formally:

$$\begin{aligned} &\forall v_p \in V, \forall v_i \in \mathit{successors}(v_p) \backslash \{v_p\}, \\ &\quad \exists v_j \in \mathit{successors}(v_p) \backslash \{v_p\} \wedge v_i \neq v_j \wedge v_i \in \mathit{post\text{-}dom}(v_j) \\ &\quad \Rightarrow \Pi(L) \trianglelefteq \Pi(L \backslash \{(v_p, v_i)\}) \end{aligned} \tag{62}$$

The collecting approach integrates worst-case path computation and cache contribution estimation, referred to as high and low level analyses respectively in prior WCET estimation approaches (Puschner and Koza 1989), and ignores most feasibility constraints. This may result in unnecessary pessimism if the infeasible paths are expensive. Reduction of the different scenarios in the CFG, e.g. by expanding the CFG to only model feasible paths, allows the capture of some flow constraints at the cost of an increase in the size of the considered flow.

### 6.3 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 $\Pi_{iter} = [\Pi(V_l \backslash \{v_h\}), [v_h]]$ captures the paths followed during a complete iteration through loop $l$.

A valid path which captures $n$ iterations can be expressed as $[[v_h], \pi_1, \ldots, \pi_{n-1}, \pi_{last}]$ with $\forall i, 1 \leq i < n, \pi_i \in \Pi_{iter}$, and $\pi_{last}$ as the last iteration of the loop. $\pi_{last}$ is a path in $\Pi(V_l \backslash \{v_h\})$ followed by a node outside the loop. We denote by $\Pi_n$, the set of paths which iterate $n$ times through the loop $l$. A path in $\Pi_{n+1}$ can be expressed as $[[v_h], \pi_1, \ldots, \pi_{n-1}, \pi_n, \pi_{last}]$ with $\pi_n \in \Pi_{iter}$, i.e. each path in $\Pi_n$ is included in a path of $\Pi_{n+1}$. 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. 4. The path $\pi_1 = [a, b, d, e, f]$ iterates a single time through $l$, with $\pi_{last} = [d, e]$. The valid iteration sequences in this example are $[d, e, b]$ and $[c, e, b]$. By inserting one iteration before the last in $\pi_1$, 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 do indeed include $\pi_1$.

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 7 and the inclusion property for consecutive loop iterations, it is sufficient for pWCET estimation to only

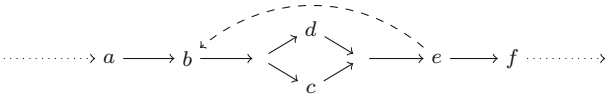



**Fig. 4** Simple do-while loop structure with an embedded conditional. The set of paths which iterate $x + 1$ times through loop $l$ includes all paths with fewer iterations

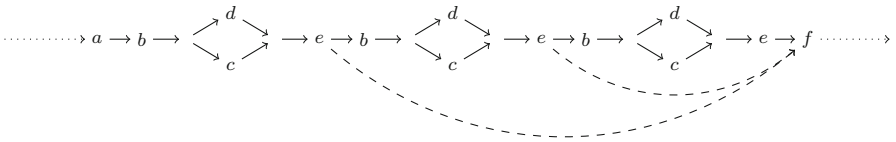



**Fig. 5** Simple do-while loop structure (Fig. 4) unrolled assuming *max-iter*($l$) = 3. The unrolled (dashed) back-edges are only kept when a generic loop unrolling algorithm is used. They are removed when *max-iter*($l$) iterations are enforced

consider paths where each loop, when executed, goes through its current maximum number of iterations. The unrolling of loop $l$ assumes *max-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.

This property holds in natural loops as long as any path taken during an iteration can be taken as well during any other iteration. Complex access patterns or flow constraints, e.g. if a path can only be executed once per execution of a loop, are a challenge to this assumption. As discussed in Sect. 6.2, the collecting approach integrates both worst-case path computation and estimation of the cache contribution, ignoring most path feasibility constraints. Expansion of the CFG to capture those constraints in its flow can be applied at the cost of a more complex flow (Fig. 5).

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.

### 6.4 Access renaming

Path inclusion relies on the verbatim sequence of accesses to detect redundancy between paths. Even the slightest dissimilarity between alternative sequences throws off the property. Some accesses are known to perform worse than others at a given point in time. Renaming an access in a sequence to a worse performing target one, i.e. changing the target of the access, can smooth the differences between paths such that the renamed path is included in an alternative path of its original counterpart. The renamed path then acts as an intermediate bound between the original one and

the including alternative, hence providing an argument for the removal of the original path. We now introduce a set of conditions that ensure the dominance of the execution time distribution of a renamed path over its original counterpart. If all transformations from the original validate these properties, the renamed path dominates the original. The renamed path may further be included in an alternative path. The original is then known to be redundant with this alternative and can be omitted during analysis.

Let $\pi = [v_1, v_2, \ldots, v_{k-1}, v_k]$ be a sequence of $k$ accesses. $\pi(e \rightarrow b)$ denotes the renaming of all accesses to memory block $e$ to $b$ in $\pi$, $\pi(e \rightarrow b) = [v'_1, v'_2, \ldots, v'_{k-1}, v'_k]$ where $\forall i \in [1, k]$, $v'_i = v_i$ if $v_i \neq e$ and $v_i = b$ otherwise. By definition, renaming $e$ to $b$ has no impact on $\pi$ if it does not access $e$. $\pi(e \rightarrow b)(c \rightarrow d)$ identifies a rename from $e$ to $b$ followed by a rename from $c$ to $d$ on the resulting sequence. Note that if no destination block is used as a source block, the order of the renames is irrelevant. For instance $\pi(e \rightarrow b)(c \rightarrow d) = \pi(c \rightarrow d)(e \rightarrow b)$, but $\pi(e \rightarrow b)(b \rightarrow c) \neq \pi(b \rightarrow c)(e \rightarrow b)$.

We identified three conditions to ensure the dominance of the pWCET of a renamed path $\pi' = [\pi_S, \pi_V(e \rightarrow b), \pi_E]$ over its original $\pi = [\pi_S, \pi_V, \pi_E]$, where $\pi_V = [e]$ or $\pi_V = [e, v_1, \ldots v_j, e]$, and further prove their impact:

- *No enclosure* There is no access to $b$ over the renamed sequence $\pi_V$, $\forall v_i \in \pi_V, v_i \neq b$.
- *Prefix ordering b* is no more likely to be in the cache than $e$ after $\pi_S$ (before $\pi_V$). This occurs when the closest access to $e$ before $\pi_V$, that is the last access to $e$ in $\pi_S$, is posterior to the last access to $b$ in $\pi_S$, $rd(e, \pi_S) < rd(b, \pi_S)$.
- *Suffix ordering b* is no more likely to trigger a hit than $e$ if present in cache after $\pi_V$ (before $\pi_E$). The first access to $e$ after $\pi_V$, i.e. in $\pi_E$, is before the first access to $b$, $frd(e, \pi_E) < frd(b, \pi_E)$.

Some inputs may result in lower estimated execution time distributions through analysis for the renamed path over the original one. This is because of the reduced pessimism in its analysis. Nevertheless, the computed pWCET for the renamed path, irrespective of the actual input, upper-bounds the exact pWCET for the original path.

**Theorem 8** (Renamed path ordering) *Given a path $\pi$ divided into three sub-paths $\pi = [\pi_S, \pi_V, \pi_E]$, where $\pi_V = [e, v_1, \ldots, v_k, e]$. The pWCET of $\pi$ is smaller than or equal to that of the renamed sequence $\pi_r = [\pi_S, \pi_V(e \rightarrow b), \pi_E]$, $\mathcal{D}(\pi) \leq \mathcal{D}(\pi_r)$, if:*

- *there is no access to b in $\pi_V$;*
- *the reuse distance of e before $\pi_V$ is smaller than that of b at this point;*
- *the forward reuse distance of e at the end of $\pi_V$ is smaller than that of b at this point.*

*Proof* See Appendix. □

### *6.4.1 Simple path elimination*

Access renaming allows for a wide range of transformations between paths within a program. We aim at reducing the set of paths that need to be considered during the

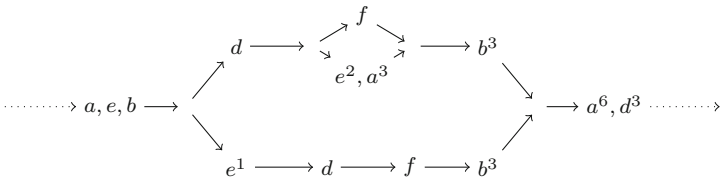



**Fig. 6** Embedded conditional structures. The maximum reuse distance of accesses is given as superscript. $[f, b]$ and $[e, a, b]$ qualify as simple paths from $d$ to $b^3$ and are matched against each other. $[d, f, b]$ and $[e, d, f, b]$ are not since the control flow may diverge at $d$

analysis of an application without increasing its pWCET. An ideal solution would consider each path individually. Each should then be matched against its larger alternatives to check for inclusion using rename operations. This approach is impractical in practice due to the sheer number of paths and the complexity of the matching problem over large sequences of accesses.

Our initial approach instead matches and eliminates simple paths in conditionals, that is branches which do not exhibit control flow divergence. Consider the example in Fig. 6, the branches of conditional $d$ are simple paths. Only the lower branch of the first enclosing conditional $b$ is a simple path as the upper one diverges at $d$. Focusing on simple paths reduces the exploration space, both in terms of considered paths and their relative size. The considered paths are likely to be similar and match using few rename operations. This simplifies path elimination in the CFG. Figure 6 however illustrates the restrictions of this approach. The topmost branch $[d, f, b]$ is redundant with the bottom one $[e, d, f, b]$, but is only compared with $[d, e, a, b]$ which it does not match.

We use a simple method outlined in Algorithms 2 and 3 to test inclusion and perform renaming at the same time. The first algorithm illustrates the traversal of a CFG, and the identification of the suitable candidates for simple path elimination through renaming. The successors of each conditional vertex are considered pairwise (lines 5–14). Should there be a simple path starting from each vertex in the pair to the same node ($dom$), the redundant one if any is removed (line 10). Paths may converge inside a simple path, as the simple path definition only restricts flow divergence. The removal of nodes subsequent to such a convergence may result in the removal of other non-redundant paths. The *RemovePath* method removes at least the edge from *vertex* to *renamed* but may need to conserve other nodes. Using the flow depicted in Fig. 7 as an example, the middle path $[a, b, f, b]$ is identified as redundant with the top one $[e, b, f, b]$, but not $[b, e, f, b]$. Only $[a^0]$ and $[b^1]$ can effectively be removed. Removing vertices $[f]$ and $[b^2]$ on the lowest branch would remove $[b, e, f, b]$ from the set of possible paths which is unsafe.

The recursive $IsRedundant$ method, outlined in Algorithm 3, focuses on asserting the redundancy of two sub-paths of a CFG using renaming. The algorithm progresses access by access, each call to $IsRedundant$ considers the first access in the renamed path $\pi_v$ and possible matches in $\pi_r$. It explores the following options (i) match the address on the two paths (line 8), (ii) attempt renaming the access on path $\pi_v$ to one on path $\pi_r$ (line 12), or (iii) skip an access on the longest trace (on line 7, the operation

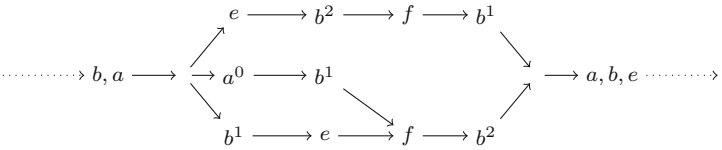



**Fig. 7** Embedded conditional structures. The maximum reuse distance of accesses is given as superscript. All three paths $[e, b, f, b]$, $[a, b, f, b]$ and $[b, e, f, b]$ qualify as simple paths and are matched against each other. $[a, b, f, b]$ is captured as redundant with respect to $[e, b, f, b]$, but not $[b, e, f, b]$

removes the head of path $\pi_r'$). If it reaches the end of path $\pi_v$, that path is identified as redundant with respect to $\pi_r$; there is a sequence of renames which results in its inclusion in $\pi_r$. Conversely, if there are not enough accesses left in $\pi_r'$ to match the ones in $\pi_v'$, the algorithm returns false. Hence, renames only occur on the shortest path, as it does not hold enough accesses to include the longer one.

The two sub-paths compared in the $IsRedundant$ method may be reached through multiple paths in the CFG and lead to the execution of different suffixes. To rename block $e$ to $b$, the operation must be valid for all prefixes and suffixes of the considered path $\pi_v'$. Any access to $b$ prior to the renamed segment should always be followed by an access to $e$ before $\pi_V$ later in the CFG (*Prefix ordering* condition). Conversely, an access to $e$ must precede the next access to $b$ on all subsequent paths where $b$ is accessed (*Suffix ordering* condition). Using the minimum forward and backward reuse distance of accesses in the CFG does not yield the required guarantee, only a necessary condition. Indeed, $b$ may be accessed on a path where $e$ was not accessed and still have higher minimum reuse and forward reuse distances. However, the reuse distances can be used to speed up the validation process. Similarly, the first met access to either block reduces the search space as it validates the property for the current branch (on $e$) or proves it does not hold (on $b$).

### *6.4.2 Control flow graph segmentation*

WCEP reduction methods aim to remove included paths whose contribution to the execution time distribution is no greater than some alternative worst-case paths. This reduces the number of accesses to be analysed and impacts the complexity of the approach. To further reduce this contribution, we present preliminary work towards the reduction of the analysed program segments through CFG partitioning (Ballabriga and Cassé 2008). This method has been first explored by Pasdeloup (2014) through heuristics tailored for SPTA.

Conceptually, the cache is flushed at defined points in the program, on partition boundaries, to reduce the number of in-flight states. Flushing is in that case an abstraction of the analysis, the system is not expected to enforce this behaviour at runtime. Partitions divide the CFG into non-overlapping sections of consecutive nodes. We select flush points such that a minimum number of $M$ misses occurs between two flushes. This allows control over the complexity and precision trade-off for the analysis. The process is sound as the loss of information regarding cache contents cannot decrease the execution time distribution of a trace as per Theorem 1. The flush operation relies on the merge defined in (63):

**Algorithm 2** Remove redundant simple paths from CFG $G$

```
1: for vertex in ReversePostOrder(G) do
     //Skip non conditional vertices
2:   if |Successors(vertex, G)| ≤ 1 then
3:     skip to next vertex
4:   end if
     //Remove redundant simple paths from successors
5:   for src in Successors(vertex, G) do
6:     for renamed in Successors(vertex, G) do
7:       dom := CommonPostDominator(src, renamed)
8:       if src ≠ renamed and IsSimplePath(src →* dom) and IsSimplePath(renamed →* dom) then
9:         if IsRedundant(renamed →* dom, src →* dom)) then
10:          RemovePath(vertex → renamed →* dom)
11:        end if
12:      end if
13:    end for
14:  end for
     //Consider enclosing conditional again after path removal.
15:  if |Successors(vertex, G)| ≤ 1 then
16:    reset vertex to closest enclosing conditional
17:  end if
18: end for
```

**Algorithm 3** *IsRedundant*$(\pi_v, \pi_r)$

```
1: if |π_v| is empty then
2:   return true
3: end if
   //Focus on the first access in π_v
4: [[e], π'_v] := π_v
5: π'_r := π_r
   //Explore possible matches in π_r
6: while |π'_r| > |π'_v| do
7:   [[b], π'_r] := π'_r
8:   if e = b then
9:     if IsRedundant(π'_v, π'_r) then
10:      return true
11:    end if
12:  else if IsRenameValid(π'_v, e → b) then
13:    return true if IsRedundant(π'_v(e → b), π'_r)
14:  end if
15: end while
16: return false
```

$$Flush : 2^{\mathbb{CS}} \rightarrow 2^{\mathbb{CS}} \tag{63}$$

$$Flush\left(\left\{\begin{array}{c}(C_0, P_0, \mathcal{D}_0)\\ \vdots \\ (C_n, P_n, \mathcal{D}_n)\end{array}\right\}\right) = \biguplus\left(\left\{\begin{array}{c}(\emptyset, P_0, \mathcal{D}_0)\\ \vdots \\ (\emptyset, P_n, \mathcal{D}_n)\end{array}\right\}\right) \tag{64}$$

Our partition of the CFG focuses on consecutive single-entry single-exit (SESE) regions (Ballabriga and Cassé 2008). All control flow enters a SESE region through its single entry and leaves through its exit. Examples of valid SESE regions in a CFG are highlighted in Fig. 8a. Consecutive SESE regions are connected to at most one

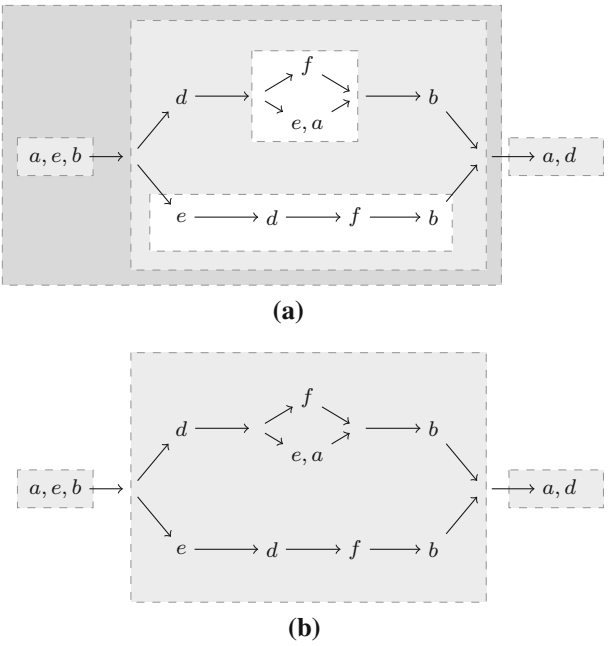



**Fig. 8** Decomposition of a CFG into single-entry single-exit regions. **a** Example of single-entry single-exit regions. **b** Example decomposition into consecutive regions

predecessor and one successor SESE region such that all control flow in the application is captured by a single path through the SESE region. As an example, consider the decomposition in Fig. 8b corresponding to the CFG in Fig. 8a. There is no cache-related dependency during the analysis of consecutive SESE regions, each is analysed assuming an empty initial cache state. Segments can be analysed independently with regards to the cache and their estimated pWCET convolved to compute that of the complete CFG.

For a decomposition into consecutive SESE regions to be valid, the nodes that delimit the segments have to be executed in all paths in the CFG. Alternative paths stemming from the same branch must be part of the same region. Similarly, all nodes in a loop nest belong to a same region. Such nodes can be captured by the notion of post-dominators: a node $v_p$ post-dominates $v_n$ if every path from $v_n$ to the end node $v_e$ goes through $v_p$. All valid candidate nodes have to be post-dominators of the entry node $v_s$.

Algorithm 4 outlines the general process of selecting the flush points. The common path of the CFG $G$, post-dominators of its entry $v_s$, is traversed in control flow order from the entry to the end of the graph. A new flush point is set if more than $M$ misses can occur between the current post-dominator and the last selected flush point. The number of potential misses between two nodes, $CountPotentialMisses(v_n, v_e)$ is computed similarly to the maximum reuse distance, accounting for all accesses that are not guaranteed hits on paths between $v_n$ and $v_e$.

**Algorithm 4** Select flush points for CFG $G$ with at least $M$ interleaved misses

```
1: flushes = ∅
2: v_s = Entry(G)
3: last_flush = v_s
4: for v in SortByFlow(post-dom(v_s)) do
5:   if CountPotentialMisses(last_flush, v) > M then
6:     flushes = flushes ∪ {v}
7:     last_flush = v
8:   end if
9: end for
10: return flushes
```

## 7 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 PapaBench application (Nemer et al. 2006), Debie (Holsti et al. 2000), and the Mälardalen benchmarks (Gustafsson et al. 2010). We present the results for a subset of benchmarks whose behaviour is representative of the ones observed across all experiments or illustrate interesting corner cases. Table 2 includes details for each benchmark on the maximum number of accesses, the distinct number of cache blocks, and the cyclomatic complexity $Y$ of the CFG (without and with WCEP reduction) which lower bounds the number of paths. Also given are the analysis runtimes with 4 and 8 relevant blocks.

The control-flow graph and address extraction were performed using the Heptane (Colin and Puaut 2001) analyser, from the compiled MIPS R2000/R3000 executable obtained using GCC v4.5.2 without optimisations. We used the various different methods to evaluate the contribution of a 16-way fully-associative instruction cache with 32B lines.

The miss distribution for different benchmarks was computed using either the contention-based approach, the collection one, using different numbers of relevant blocks $R$, or the reuse distance-based path merging method outlined by Davis et al. (2013). To provide a comparison with methods and replacement policies, a state-of-the-art analysis (Theiling et al. 1999) was used to determine the single, predicted worst-case bound on the number of misses for a LRU cache using the same parameters. 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.

All of the WCEP reduction techniques described in Sect. 6 were used for analysis of the random replacement cache. LRU caches do not exhibit the properties required by Theorem 7. The pWCET estimates obtained for each configuration of the analysis, estimation method and number of relevant blocks, were always tighter with WCEP reduction. Regarding simulation, WCEP reduction reduces the set of paths to one more representative of the worst-case scenarios, in some cases resulting in a single worst possible execution path. Yet there is no guarantee that these transformations

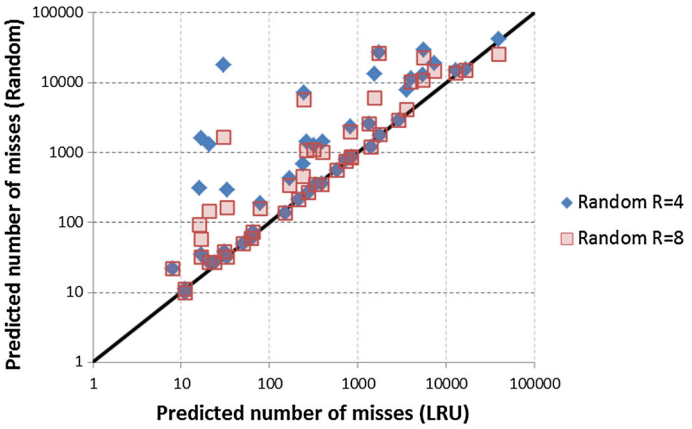



**Fig. 9** Estimated number of misses under LRU and Random replacement caches

are sufficient, here the simulation results are only an indicative means of assessing the pessimism in our approach. Table 2 covers the impact of WCEP reduction on the cyclomatic complexity $Y$ of the analysed benchmarks, an indicative lower bound on the number of paths in a CFG. Table 3 and Figure 9 present the estimated number of misses for the analysis of LRU and random replacement caches with a cutoff probability of $10^{-7}$, i.e. the number of predicted misses exceeded with a probability no greater than $10^{-7}$ at runtime. Of the 48 analysed benchmarks, 17, highlighted in Table 3, show the same or better estimated performance with a random replacement cache while 31 perform better with an LRU cache. Improvements over the LRU analysis tend to be limited. However as further illustrated for *Papabench t4* in Fig. 10c, there is a potential margin for improvement in the analysis of the random replacement policy to further tighten its results over the LRU replacement policy.

The capacity-conserving join heuristic which allocates capacity to the cache states identified as the most valuable dominates the standard implementation. When comparing the precision of the different analysis techniques in Sect. 7.1 we therefore rely on the most favourable configuration, i.e. with WCEP reduction active and using the capacity-conserving join. The impact of the different mechanisms, joins and WCEP reduction, is further considered in Sects. 7.2 and 7.3 respectively. Finally, the complexity and runtime for different analysis configurations is evaluated in Sect. 7.4.

### 7.1 Relative precision of the analysis techniques

We first compare the result for different configurations in Fig. 10a–f. 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 circles and blue crosses respectively. The number of relevant blocks $R$ for the collecting approach is restricted to values of either 4 or 8 (identified by orange triangles) which is sufficient to capture most of the locality in the considered applications. The distribution obtained through simulation (identified by green squares) is

**Table 2** Properties of the analysed benchmarks and analysis runtime with $R$ relevant blocks

| | Longest path (accesses) | \|Blocks\| | Runtime (s) | | Y with reduction | |
|---|---|---|---|---|---|---|
| | | | $R = 4$ | $R = 8$ | Off | On |
| Mälardalen | | | | | | |
| adpcm | 35,010 | 240 | 556 | 3747 | 6281 | 3069 |
| bsort100 | 108,718 | 20 | 1545 | 31,301 | 9902 | 101 |
| bs | 42 | 11 | < 1 | < 1 | 9 | 5 |
| cnt | 1576 | 27 | 1 | 1 | 201 | 101 |
| compress | 31,382 | 86 | 151 | 1047 | 3976 | 493 |
| crc | 27,752 | 44 | 478 | 1023 | 4173 | 4169 |
| edn | 67,631 | 166 | 549 | 17340 | 5 | 1 |
| expint | 11,314 | 31 | 10 | 111 | 404 | 104 |
| fdct | 841 | 106 | < 1 | 2 | 1 | 1 |
| fft | 18,409 | 141 | 78 | 432 | 609 | 587 |
| fibcall | 125 | 8 | < 1 | < 1 | 2 | 1 |
| fir | 992 | 22 | < 1 | 2 | 31 | 11 |
| insertsort | 769 | 16 | < 1 | 1 | 1 | 1 |
| jfdctint | 1059 | 96 | < 1 | 4 | 65 | 1 |
| lcdnum | 233 | 20 | < 1 | 1 | 171 | 61 |
| ludcmp | 3950 | 98 | 1 | 24 | 70 | 8 |
| matmult | 63,839 | 28 | 481 | 5967 | 801 | 1 |
| minmax | 26 | 22 | 1 | < 1 | 9 | 5 |
| minver | 726 | 167 | 2 | 1 | 7 | 1 |
| ndes | 21,377 | 121 | 47 | 355 | 4219 | 1273 |
| nsichneu | 2944 | 1377 | 107 | 103 | 1249 | 1 |
| ns | 4349 | 20 | 1 | 33 | 2 | 2 |
| prime | 5768 | 17 | 3 | 21 | 725 | 5 |
| qurt | 1526 | 77 | < 1 | 4 | 187 | 67 |
| select | 1721 | 60 | < 1 | 1 | 177 | 17 |
| sqrt | 430 | 26 | < 1 | 1 | 59 | 20 |
| statemate | 1844 | 275 | 49 | 49 | 1841 | 1132 |
| st | 67,538 | 163 | 127 | 780 | 971 | 221 |
| ud | 2984 | 75 | 1 | 12 | 82 | 1 |
| Papabench | | | | | | |
| t1 | 150 | 135 | < 1 | < 1 | 41 | 17 |
| t2 | 57 | 27 | < 1 | < 1 | 6 | 5 |
| t3 | 62 | 57 | < 1 | 1 | 20 | 9 |
| t4 | 215 | 13 | < 1 | < 1 | 47 | 24 |
| t5 | 62 | 55 | < 1 | < 1 | 19 | 13 |
| t6 | 286 | 272 | < 1 | < 1 | 103 | 27 |
| t7 | 52 | 45 | < 1 | < 1 | 9 | 8 |
| t8 | 11 | 9 | < 1 | < 1 | 3 | 2 |

**Table 2** continued

| | Longest path (accesses) | \|Blocks\| | Runtime (s) | | Y with reduction | |
|---|---|---|---|---|---|---|
| | | | $R = 4$ | $R = 8$ | Off | On |
| t9 | 472 | 324 | < 1 | < 1 | 89 | 11 |
| t10 | 39,658 | 1073 | 2500 | 4742 | 16,602 | 10,513 |
| t11 | 11 | 9 | < 1 | < 1 | 6 | 4 |
| t12 | 33 | 34 | < 1 | < 1 | 18 | 10 |
| t13 | 581 | 675 | < 1 | < 1 | 204 | 26 |
| fly_by_wire | 18,723 | 229 | 293 | 358 | 4355 | 1930 |
| Debie | | | | | | |
| acquisition_task | 18,664 | 205 | 18 | 490 | 3829 | 1273 |
| hit_trigger_handler | 3367 | 83 | 4 | 9 | 671 | 471 |
| tc_execution_task | 3131 | 417 | 3 | 13 | 368 | 251 |
| tc_interrupt_handler | 77 | 91 | 1 | 1 | 39 | 27 |
| tm_interrupt_handler | 24 | 30 | 2 | 2 | 9 | 7 |

**Table 3** Estimated number of misses with LRU and random replacement caches

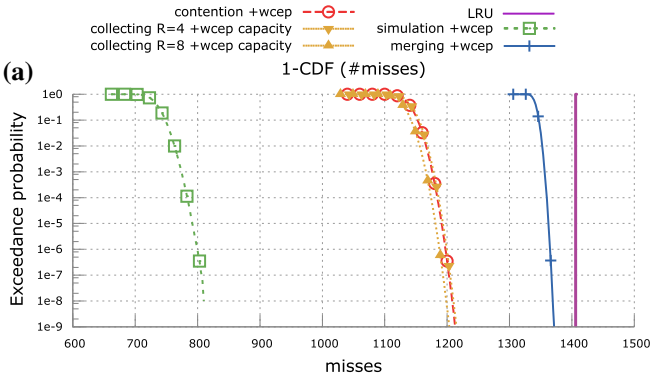
| | Number of estimated misses | | |
|---|---|---|---|
| | LRU | Random ($10^{-7}$) | |
| | | $R = 4$ | $R = 8$ |
| Mälardalen | | | |
| adpcm | 1570 | 13,173 | 6097 |
| bsort100 | 39,518 | 41,642 | 25,319 |
| bs | 17 | 35 | 32 |
| cnt | 239 | 674 | 450 |
| compress | 3564 | 7808 | 4058 |
| crc | 248 | 7138 | 5693 |
| edn | 5608 | 29,018 | 22,546 |
| expint | 320 | 1253 | 1107 |
| fdct | 840 | 842 | 842 |
| fft | 16,847 | 15,259 | 15,050 |
| fibcall | 8 | 22 | 22 |
| fir | 33 | 291 | 161 |
| insertsort | 16 | 304 | 91 |
| jfdctint | 739 | 800 | 748 |
| lcdnum | 214 | 211 | 209 |
| ludcmp | 836 | 2310 | 1990 |
| matmult | 30 | 17,812 | 1665 |
| minmax | 24 | 27 | 27 |
| minver | 171 | 427 | 335 |
| ndes | 5524 | 13,101 | 10,882 |

**Table 3** continued

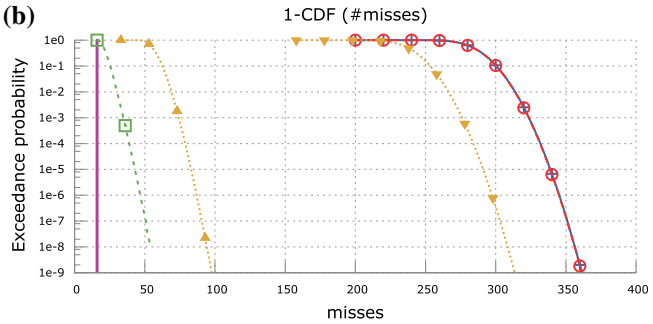
| | Number of estimated misses | | |
|---|---|---|---|
| | LRU | Random ($10^{-7}$) | |
| | | $R = 4$ | $R = 8$ |
| nsichneu | 2943 | 2840 | 2844 |
| ns | 21 | 1296 | 145 |
| prime | 17 | 1591 | 58 |
| qurt | 1406 | 1205 | 1193 |
| select | 856 | 856 | 856 |
| sqrt | 392 | 355 | 351 |
| statemate | 1802 | 1749 | 1775 |
| st | 1740 | 27,044 | 26,372 |
| ud | 406 | 1435 | 1005 |
| Papabench | | | |
| t1 | 150 | 137 | 137 |
| t2 | 31 | 38 | 38 |
| t3 | 62 | 59 | 59 |
| t4 | 79 | 188 | 158 |
| t5 | 62 | 59 | 59 |
| t6 | 278 | 268 | 268 |
| t7 | 51 | 49 | 49 |
| t8 | 11 | 10 | 10 |
| t9 | 334 | 343 | 343 |
| t10 | 7421 | 18,825 | 14,506 |
| t11 | 11 | 11 | 11 |
| t12 | 33 | 32 | 32 |
| t13 | 581 | 559 | 559 |
| fly_by_wire | 12,840 | 15,126 | 13,822 |
| Debie | | | |
| acquisition_task | 4033 | 11,475 | 10,147 |
| hit_trigger_handler | 1345 | 2534 | 2529 |
| tc_execution_task | 262 | 1432 | 1060 |
| tc_interrupt_handler | 65 | 73 | 73 |
| tm_interrupt_handler | 21 | 27 | 27 |

also presented. The number of misses predicted by analyses for the deterministic LRU configuration is identified by a dark purple vertical line.

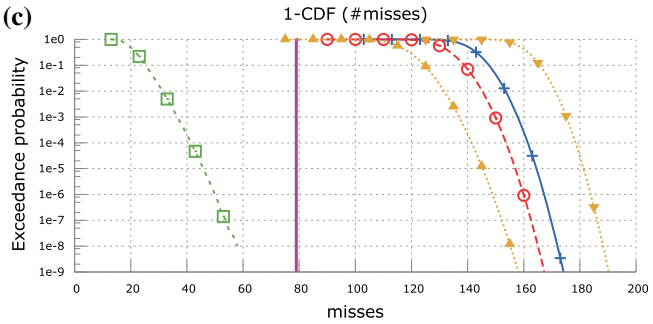
In general, the use of the cache collecting method improves the precision of the analysis over the merging or purely contention-based approaches even on complex control flows, as illustrated by papabench t4 in Fig. 10c. On simple control flows, the two approaches behave similarly but the contention method still dominates the path merging method (see Fig. 10e). The merged path is as long as the longest path in the application but keeps the worst behaving accesses from shorter paths. When WCEP




Estimated miss distribution for qurt, 77 distinct memory blocks, 1526 accesses on the longest path

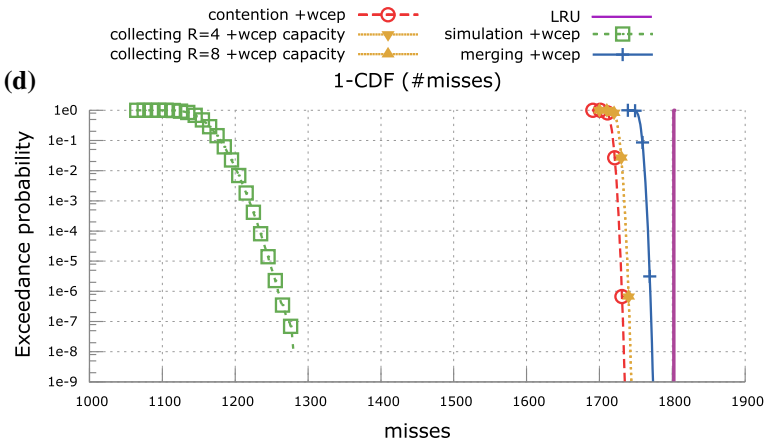



Estimated miss distribution for insertsort, 16 distinct memory blocks, 769 accesses on the longest path

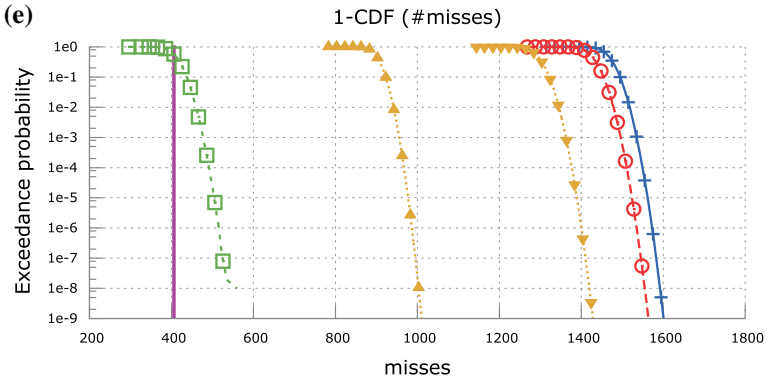



Estimated miss distribution for papabench t4, 13 distinct memory blocks, 215 accesses on the longest path

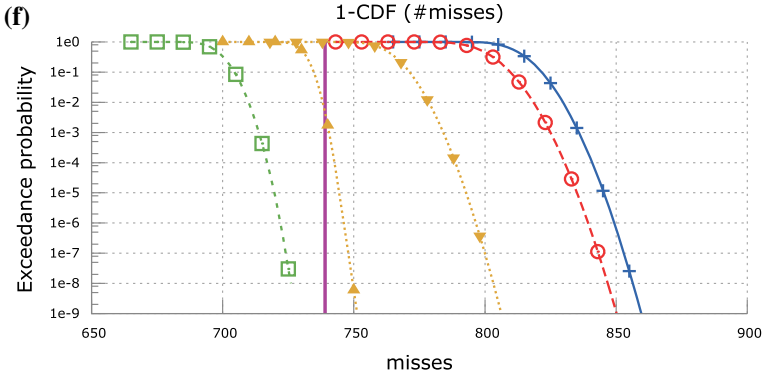
**Fig. 10** Estimated miss distribution of the different analysis methods under LRU or random replacement policies and different analysis configurations (Color figure online)




Estimated miss distribution for statemate, 275 distinct memory blocks, 1844 accesses on the longest path




Estimated miss distribution for ud, 75 distinct memory blocks, 2984 accesses on the longest path




Estimated miss distribution for jfdctint, 96 distinct memory blocks, 1059 accesses on the longest path

**Fig. 10** continued

reduction can mostly extract the worst-case execution path, as with qurt in Fig. 10a, 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 collection 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 relevant blocks. Similar behaviours were observed whether WCEP reduction successfully resulted in a single path or not. As the number of relevant blocks increases from 4 to 8, the estimates computed by the analysis improve. The gain is important on benchmarks like insertsort (see Fig. 10b) where some nested loops fit in the number of relevant blocks. However, precision is lost in qurt or ud w.r.t. the simulation results (see Fig. 10a, e) as the loops almost fit inside the cache but not within the number of relevant 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 general conclusion regarding the dominance of the analysis of a LRU cache over simulation or analysis results for a randomised cache. When all iterative structures fit in the cache (see Fig. 10b), the LRU analysis outperforms the analysis of the random cache. As intra-loop conflicts grow, the benefits of the random replacement policy emerge and the new methods can capture such locality, resulting in tighter estimates than the analysis for a deterministic platform (see Fig. 10f). WCEP reduction reduces the reuse distance considered during analyses, whereas the stack distance for the LRU analysis remains the same since Theorem 7 does not apply. The path-merging approach under WCEP reduction may result in tighter estimates than the analysis of a deterministic replacement policy (see Fig. 10d).

The analysis results for the t4 and statemate benchmarks (see Fig. 10c, d) indicate that the cache collecting approach may sometimes compute more pessimistic estimates than the contention method. This behaviour stems from flow divergence in the control flow of both benchmarks. Path indeterminism hinders the relevant block heuristic, different blocks may be deemed as relevant on parallel paths. In such cases, upon flow convergence, the join function cannot keep blocks of either alternative. Further, the $R$ relevant blocks are still considered as occupying cache space from the point of view of the non-relevant ones, effectively reducing the cache size. This illustrates the need for more sophisticated heuristics which take into account the behaviour of the analysis on alternative paths, or vary the number of relevant 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.

### 7.2 Benefits of the join operations to collecting approaches

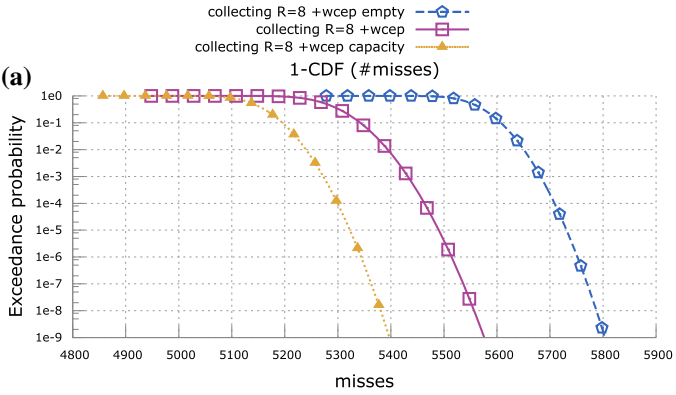
The selection of relevant blocks is undoubtedly an important factor in the precision of the cache collecting approach. We compared additional configurations of the analyser, assuming a fixed number of 8 relevant blocks, to examine the impact of the join operations on the precision of the analysis. In particular, the experiments presented from Fig. 11a–f introduce a non state-conserving approach on path convergence. Using configuration *empty* (identified by blue pentagons) the cache contents are set to $\emptyset$ on path convergence and the miss distribution is the maximum distribution of the alternative paths. The *capacity* configuration (identified by orange triangles) on the other hand corresponds to the use of the improved join operator. The intermediate line identifies the simple join operation we first introduced in Sect. 4.6 (purple squares).

Benchmarks which exhibit locality across branches of their conditionals benefit from the join function, as illustrated by crc, lcdnum, expint and compress in Fig. 11a, b, d and e respectively. The combination of both WCEP reduction and capacity conservation on flow convergence leads to tighter pWCET estimates in the case of crc, lcdnum, expint and compress. Reduction cannot remove all branches as they may not fall under the required constraints. The lcdnum benchmark is composed of a *switch* statement. The later cases share blocks with the conditions of the earlier ones, but add their own blocks. Hence, the resulting cache states differ but include each other. They can be captured by the capacity conserving heuristic. By construction, the capacity conserving join results in the tightest estimates and provides important improvements over the standard join on the crc application. The benefits of the capacity-conserving join over the standard one are more marginal on the compress benchmark (see Fig. 11e) which exhibits few branches with reused blocks not captured by the WCEP reduction.

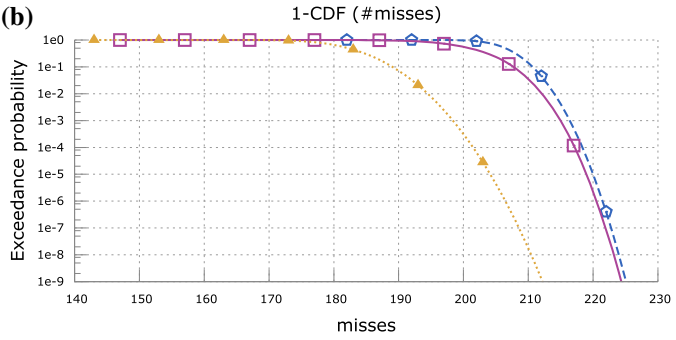
Some benchmarks see little benefit from the proposed join function. statemate (see Fig. 11c) is composed of many nested conditional constructs which share few or no blocks. The cache contents diverge with the flow, and the join operation cannot assume any block is present. Locality in the statemate benchmark is captured thanks to the empty conditional approach of WCEP reduction. Some applications like matmult (see Fig. 11f) are reduced to a single path through WCEP reduction. Such scenarios obviously do not benefit from any join operation.

### 7.3 Impact of WCEP reduction on analysis and simulation

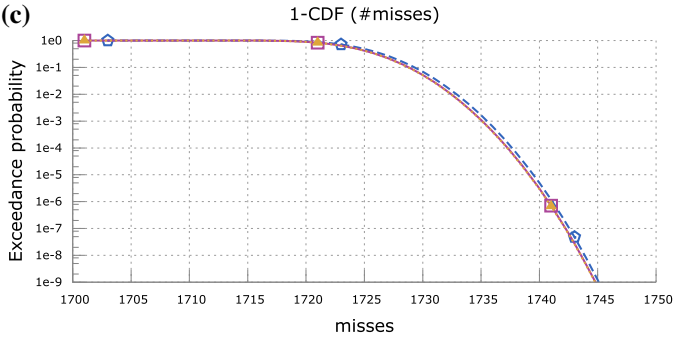
WCEP reduction reduces path redundancy through the elimination of selected paths, such that both the analysis and the simulations are performed on a reduced control flow. We computed the miss distribution of the benchmarks with both families of methods with and without WCEP reduction. We present the result for a fixed probability of $10^{-7}$, i.e. recording the number of predicted misses that will be exceeded at runtime with a probability no greater than $10^{-7}$. For each of the ludcmp, cnt and compress benchmarks in Fig. 12a, b, and c respectively, we present the result using the original CFG, then adding WCEP unrolling (+*unroll*), empty branch elimination (+*branch*), and the renaming-based simple branch elimination (+*rename*).




Estimated miss distribution for crc, 44 distinct memory blocks, 27,752 accesses on the longest path

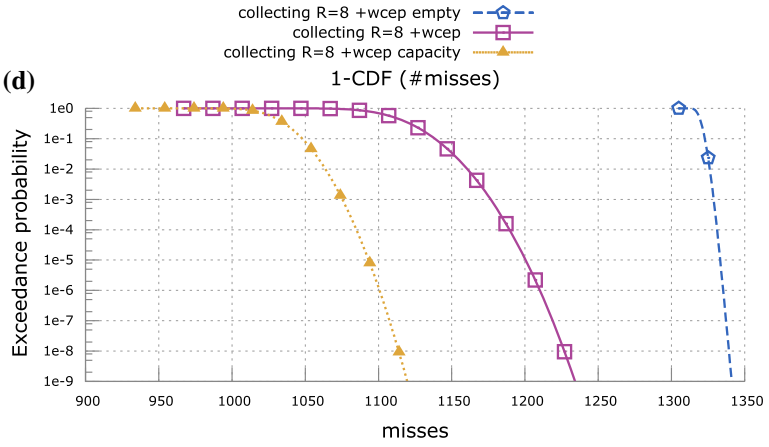



Estimated miss distribution for lcdnum, 20 distinct memory blocks, 233 accesses on the longest path

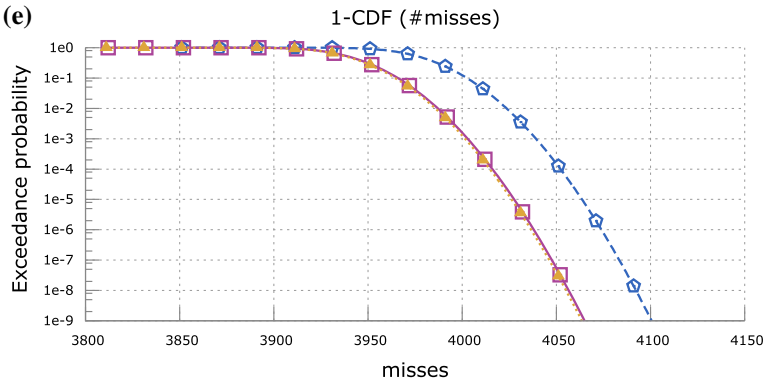



Estimated miss distribution for statemate, 275 distinct memory blocks, 1844 accesses on the longest path

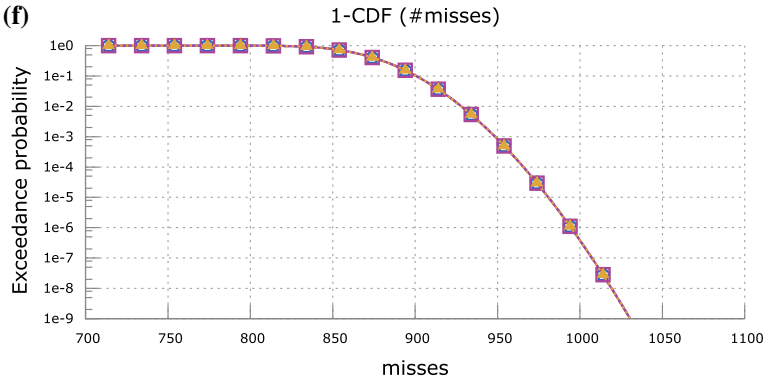
**Fig. 11** Estimated miss distribution for the random replacement policy under a fixed number of relevant blocks $R = 8$ and WCEP reduction, and different join operations (Color figure online)




Estimated miss distribution for expint, 31 distinct memory blocks, 11,314 accesses on the longest path




Estimated miss distribution for compress, 86 distinct memory blocks, 31,382 accesses on the longest path




Estimated miss distribution for matmult, 28 distinct memory blocks, 63,839 accesses on the longest path

**Fig. 11** continued

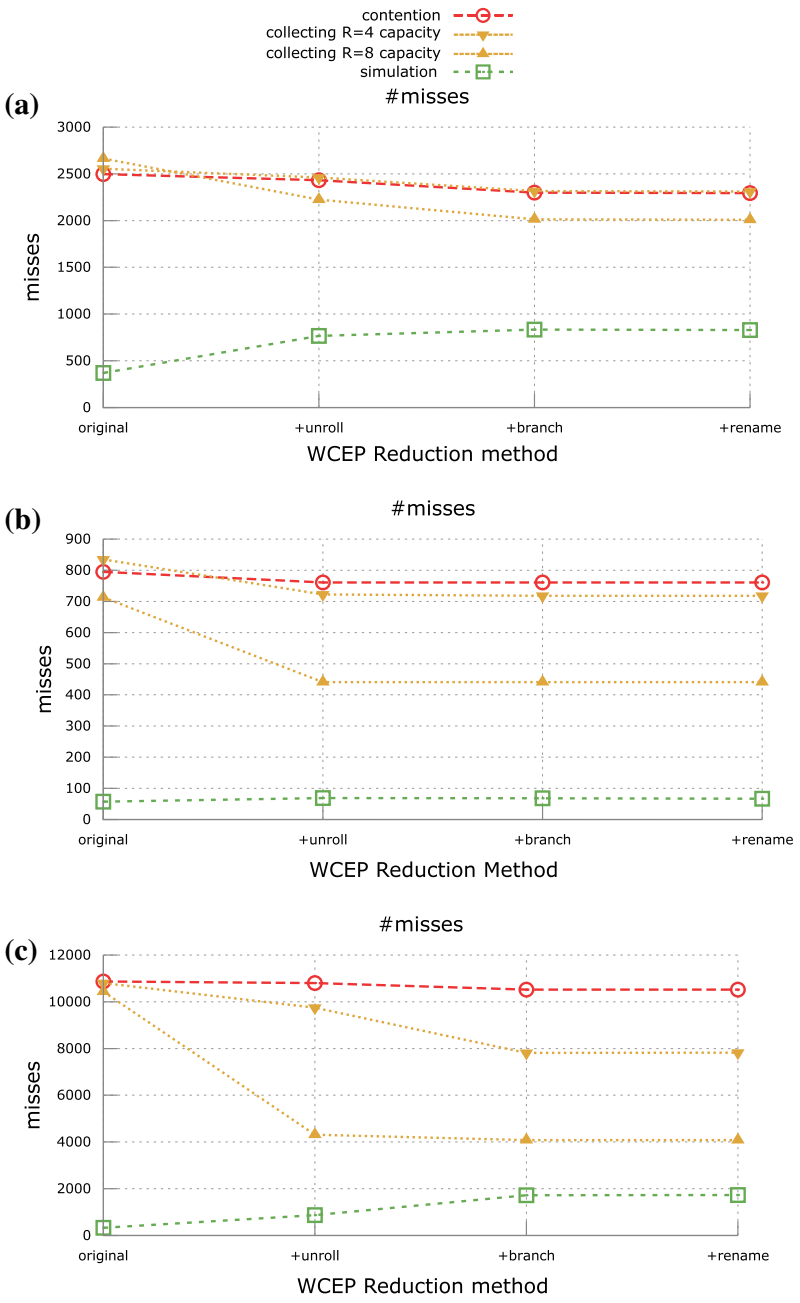



**Fig. 12** Estimated miss counts at a fixed probability of $10^{-7}$ under random replacement using different reduction configurations and $R$ relevant blocks. **a** Estimated miss count at $10^{-7}$ for the ludcmp benchmark. **b** Estimated miss count at $10^{-7}$ for the cnt benchmark. **c** Estimated miss count at $10^{-7}$ for the compress benchmark (Color figure online)

Given a fixed configuration of the analysis (identified by a symbol and a colour), the distribution obtained with WCEP reduction is always smaller than the one obtained without it. In other words, the analysis is more precise when all transformations are active. Because of path redundancy, an increase in the number of relevant blocks can sometimes reduce the precision of the resulting estimate. This phenomenon still occurs when WCEP reduction is applied, but it is less prevalent.

The impact of the different transformations on the precision of the analysis results depends on the characteristics of the application to which they are applied. All transformations can be beneficial to benchmarks for the collecting approach. The contention approach may even benefit from empty path elimination (see Fig. 12a), when a block is accessed only on the non-empty alternative of a conditional its reuse distance gets lowered. For other accesses, such paths impact neither the reuse distance nor the contention as they hold no access. The elimination of redundant paths on the other hand increases the precision of the two methods.

The cnt benchmark, in Fig. 12b, illustrates an interesting scenario. When the empty branch elimination is used in combination with WCEP unrolling collecting methods get slightly less precise than when using WCEP unrolling on its own. This illustrates a limit of the ranking heuristic used by the capacity-conserving join. Empty branches result in a reduced minimum forward reuse distance for some accesses. This in turn impacts the allocation of capacity to cache states on path convergence, resulting in a better allocation without empty branch elimination.

We performed a set of $10^8$ simulations on the control flow graphs of benchmarks with and without reduction. WCEP reduction results in greater measured execution time distributions. The transformations proposed in this paper eliminate some but not all redundant paths and reduce the set of possible paths to a set more focussed on worst-case scenarios. As for the analyses methods, the impact of each transformation depends on the benchmark to which it is applied. However, the application of WCEP reduction in the general case is not sufficient to guarantee the representative character of the resulting paths. In the case of the expanded compress benchmark, conditionals within loop structures are kept and there is no guarantee as to which alternation of paths results in the worst-case. On the other hand, the expanded matmult benchmark consists of a single trace of accesses.

### 7.4 Execution time

The runtime of the analysis, using a C++ prototype implementation, is presented in Fig. 13 using the WCEP reduction method and 0 to 12 relevant blocks. 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 reduction was used as it increases the precision of the estimated cache states, and also the analysis runtime. We observe a growth in runtime as the number of relevant blocks increases. The runtime of the analysis is also significantly higher for larger benchmarks, edn, compress, and ndes, which contain the largest number of nodes.

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

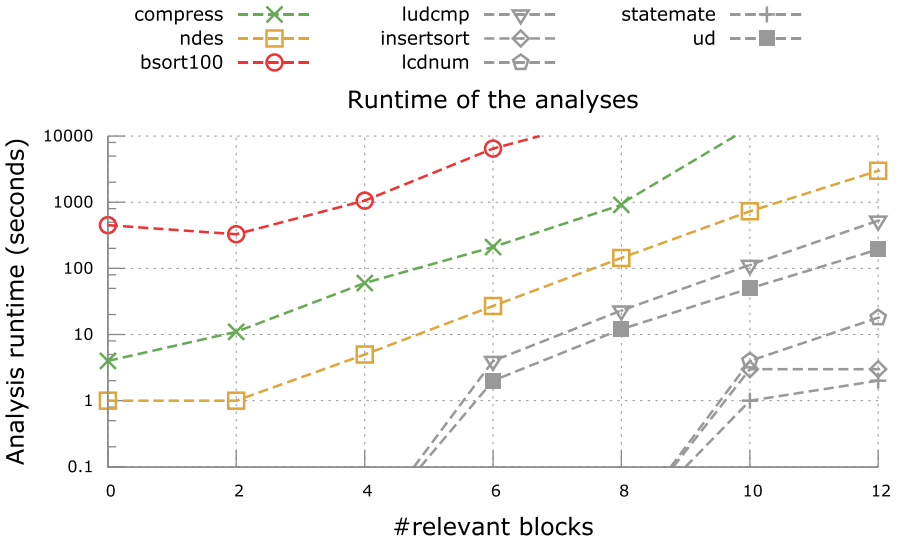



**Fig. 13** Runtime of the analysis for the presented benchmarks

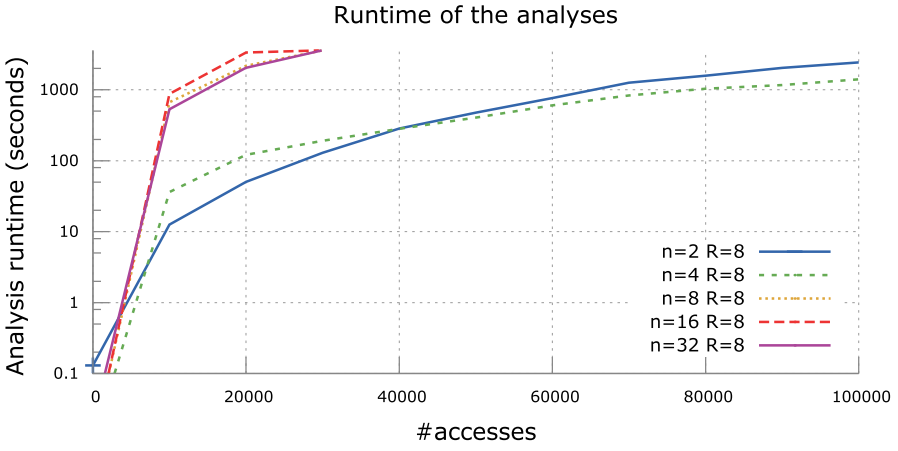



**Fig. 14** Runtime of the analysis for repeated accesses to a sequence of $n$ distinct blocks

to both the number of relevant blocks $R$ and the number of potential misses on the longest path. (Fig. 13 combines the impact of both the program length and number of relevant blocks whereas Fig. 14 focuses on the number of instructions.) The number of relevant blocks affects the number of different cache contents which are tracked by the analysis at each step. Further 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 amongst

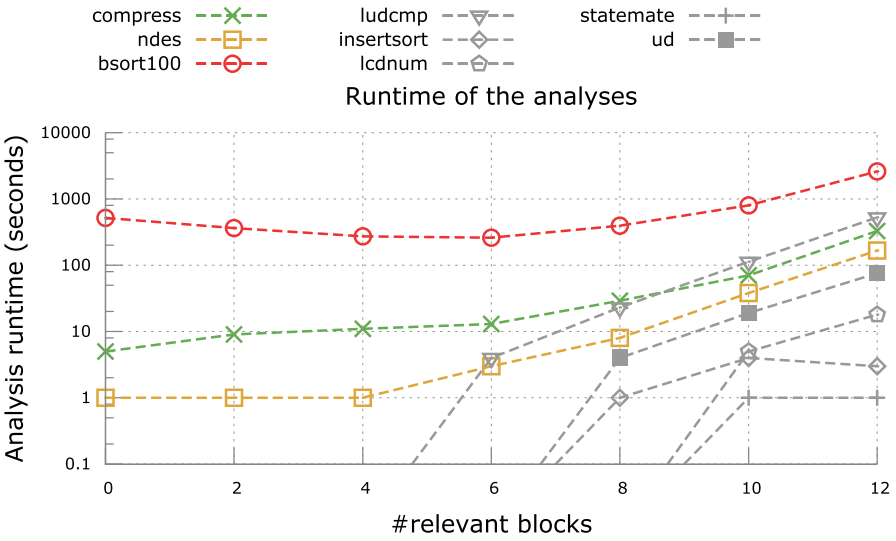



**Fig. 15** Runtime of the analysis under CFG partitioning with segments of 1000 potential misses (Color figure online)

the $R$ relevant blocks, when $R < N$ then $|S| = 2^R$. As demonstrated in the previous set of experiments, a limited number of relevant blocks is effective for typical cache associativities.

### *7.4.1 Reducing the complexity of the approach*

The complexity of the introduced approach to SPTA for multi-path programs depends on both the number of relevant blocks $R$ and the number of accesses $m$ in the program. This section further examines the contribution of the program size to the runtime of the analyses and presents preliminary work towards its reduction using CFG partitioning as presented in Sect. 6.4.2.

The results presented in Fig. 13 focussed on the impact of the number of cache states through its ties to the relevant blocks $R$. The number of accesses $m$ in each benchmark is fixed. We evaluate the impact of $m$ on the complexity of the analysis in Fig. 14. It 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 relevant blocks $R$ and the cache associativity $N$ impact the possible number of cache states $|S|$ and therefore the initial growth of the runtime. Once the set of cache states to consider stabilises, the runtime for the different configurations follows a similar $m \times log(m)$ growth curve.

We defined a simple algorithm to split a program into consecutive SESE with at least $M$ non-guaranteed hits on their longest path (Sect. 6.4.2). Segments are analysed independently assuming an empty input cache, and the resulting pWCET convolved to compute that of the full program. This approach effectively reduces the set of cache states on region boundaries to the empty state, a safe over-approximation as defined in Sect. 4.5. The resulting analysis runtime for the largest benchmarks is presented in Fig. 15 assuming a segment size $M$ of 1000 misses.

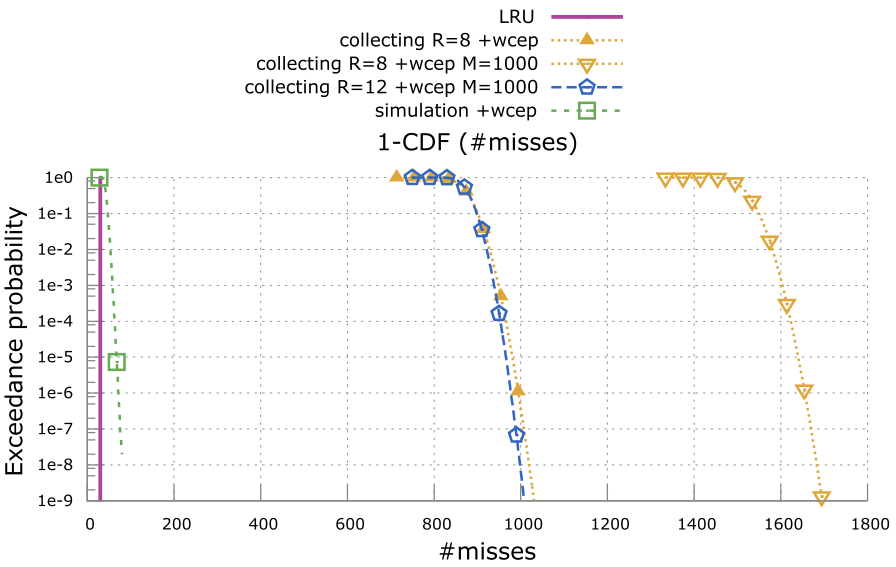



**Fig. 16** Estimated miss distribution for matmult under LRU and random replacement with analysed segments of $M = 1000$ potential misses (Color figure online)

Program partitioning reduces the runtime of our method over the analysis of the program as a single segment (see Fig. 13). 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 relevant blocks. The remaining differences in runtime come from several factors. First, the length of the program impacts the complexity of the final convolution operation of the pWCET of each segment. Second, the consecutive segments on a multi-path program may hold more than $M$ misses. Splits can only occur on a restricted set of vertices, namely those which post-dominate the entry of the CFG. Further, as shown in Fig. 14, 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.

Figures 16, 17 and 18 present the distributions computed by the analyses for a relevant subset of the considered configurations. They present the analyses results for $R = 8$ relevant blocks using a single or multiple segments (filled or hollow triangles respectively). They also include the results for 12 relevant blocks under partitioning (hollow blue pentagons), as the runtime of this configuration is below that of the $R = 8$ single segment one. Simulations and deterministic LRU analyses results are also included (resp. with green squares and a dark purple line). WCEP reduction 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 accesses 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 Figs. 16 and 17. matmult exhibits an important locality at runtime, the impact of segment boundaries is such

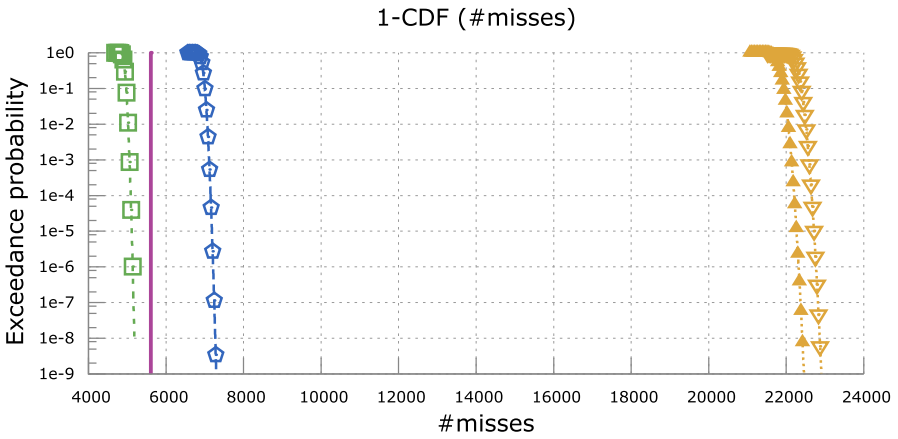



**Fig. 17** Estimated miss distribution for edn under LRU and random replacement with analysed segments of $M = 1000$ potential misses (Color figure online)

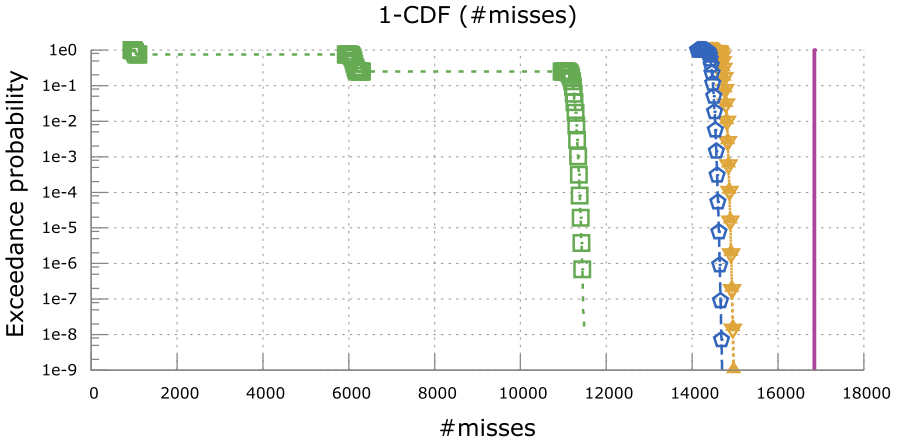



**Fig. 18** Estimated miss distribution for fft under LRU and random replacement policies with analysed segments of $M = 1000$ potential misses (Color figure online)

that it overshadows the increase in the number of relevant blocks. Yet, the segmented analysis with 12 relevant 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 relevant blocks is much more important for edn, while the runtime of the $R = 12$ segmented analysis remains lower than that of the $R = 8$ full program one (2000s vs. 13,000s).

We observed that the fft benchmark (Fig. 18) only marginally benefits from an increase in the number of relevant blocks. The approximations on segment boundaries have almost no impact on the precision of the computed estimates given a fixed number of relevant blocks, $R = 8$. There is little reuse between the identified SESE regions in the program.

## 8 Conclusion and perspectives

The main contribution of this paper is the introduction of a more effective 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. We have pointed out where existing techniques for deterministic or probabilistic analyses could be applied to make improvements (Pasdeloup 2014; Maxim et al. 2012; Wegener 2012; Theiling et al. 1999).

We introduced conditions for the computation of valid upper-bounds on the possible cache states on control flow convergence and presented a compliant transfer function to illustrate the requirements. We further refined this join operation to improve the precision of the information kept on control flow convergence. This more sophisticated join operation relies on a heuristic ordering of cache states depending on their expected benefits in the upcoming accesses.

We also defined path redundancy, identifying path inclusion as a sub-case of redundancy. Based on these results, we presented worst-case execution path (WCEP) reduction to reduce the set of paths explored by the analysis, improving the tightness of the resulting timing estimates. We identified and proved the validity of sufficient conditions for the application of access renaming. This transformation allows for the identification of redundant paths beyond simple inclusion.

Our evaluations show that the analysis derived is effective at capturing the cache locality exhibited by different applications. The new methods significantly outperform the existing path merging approaches, predicting less than a third as many misses in one of the benchmarks. More precise results can be attained at the cost of an increased, user-controlled, complexity. They are also incomparable to estimates for deterministic LRU caches. The program transformations introduced proved effective at improving the precision of all SPTA configurations; of the 48 analysed benchmarks, 18 show the same or better estimated performance with a Random replacement cache while 31 perform better with an LRU cache.

### 8.1 Perspectives

This research can be extended in many ways. The transfer functions on control flow convergence compute valid bounds with regards to the ordering of cache states. They exhibit pessimism, different but more complex ranking heuristics could spread the capacity of cache states over more appropriate candidates. Second, the complexity of operations on the abstract domain contributes to the increasing runtime of the analysis as it traverses deep flow graphs. Future work could look at the interaction between existing methods to balance the complexity and the precision of the analysis. Another avenue for improvement is the heuristic for the selection of relevant cache blocks. More advanced approaches might improve the tightness of the results, or even introduce a varying number of relevant blocks across the application to focus the analysis effort on a specified area of the code.

Our approach integrates both cache behaviour and worst-case path estimation. Flow facts regarding loop iterations can be taken care of during unrolling. We nevertheless

intend to take more flow facts into account to increase the applicability of the approach and further improve the WCEP reduction effect on reducing path complexity. We also intend to investigate the use of static methods to improve the representative character of the considered paths, and as a consequence ensure the soundness and improve the precision of the measurement-based approaches. Finally, the application of static probabilistic timing analysis to more complex cache configurations, including multiple levels of cache, remains an open problem (Lesage et al. 2013).

**Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

## Appendix: Proofs related to cache contents comparison

This appendix provides the mathematical proofs of different properties related to the comparison of cache contents based on their impact on the execution time of an access trace. We also demonstrate the validity of the sufficient conditions for the renaming program transformation. The following lemmas and properties have been first introduced in Sects. 3, 4.5, and 6 and rely on the definition (21) of the contribution of a cache state to the execution time of a trace.

**Theorem 1** *The eviction of a block from any input cache state $s$ cannot decrease the execution time distribution of any trace $t$, $\mathcal{D}(t, s) \leq \mathcal{D}(t, s[-e])$.*

*Proof* We focus on the case where the evicted block $e$ is in the input cache state $s$, otherwise $s[-e]$ is equal to $s$ and $\mathcal{D}(t, s) = \mathcal{D}(t, s[-e])$. We prove the theorem by induction.

Base case $t = [b]$: There are two cases to distinguish when the trace comprises a single access. If $b \neq e$, then the execution time of $[b]$ from $s$ or $s[-e]$ is the same since the eviction of $e$ does not affect $[b]$. If $b = e$, the access $[b]$ may either be a hit or a miss starting from $s$, but is guaranteed to be a miss from $s[-b]$. Hence, $\mathcal{D}([b], s) \leq \mathcal{D}([b], s[-e])$.

Inductive case $t' = [[b], t]$: Suppose the theorem holds for trace $t$ and any input cache state $s$, $\mathcal{D}(t, s) \leq \mathcal{D}(t, s[-e])$. The cases to consider are:

$b \neq e$ The behaviour of the first access $[b]$ is the same from both $s$ and $s[-e]$. If $b$ is in cache, the access is a hit, the cache state is left unchanged, and the property holds thanks to the induction hypothesis. Otherwise, this results in $N$ different outcomes corresponding to the eviction of any one of the $N$ different lines $l_i$, $s[-e][l_i = b]$ where $e$ is first evicted from $s$ then $b$ replaces $l_i$. If $l_i$ is the line which held $e$, $s[-e][l_i = b] = s[l_i = b]$; the behaviour of trace $t$ is the same from either state. If one of the $N - 1$ other lines is selected, then the eviction of $e$ from the cache could take place after or before the replacement by $b$ without any impact on the resulting cache contents, $s[-e][l_i = b] = s[l_i = b][-e]$. As defined by the induction hypothesis, the $N$ resulting cache states from $s[-e]$

cannot decrease the execution time distributions over their counterparts for $s$ thus:

$$\mathcal{D}(t, s[l_i = b]) \leq \mathcal{D}(t, s[-e][l_i = b]) \tag{65}$$

Hence, the sum of these distributions, in (21), cannot result in a decrease of the execution time distributions from $s[-e]$ over $s$:

$$\mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot (\mathcal{D}(t, s[l_i = b])) \leq \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot (\mathcal{D}(t, s[-e][l_i = b])) \tag{66}$$

$b = e$ The first access $[b]$ is a hit in $s$ and $t$ executes from input state $s$. From $s[-e] = s[-b]$, the first access $[b]$ is a miss. As in the previous case, the resulting cache states may be the same should the lines selected for eviction and replacement match, $s[-b][l_j = b] = s$. Alternatively, another block is evicted from the cache to insert $b$ again and the resulting state $s[-b][l_j = b]$ holds the same contents as $s[-l_j]$. From the induction hypothesis, we know that:

$$\mathcal{D}(t, s) \leq \mathcal{D}(t, s[-l_j]) \tag{67}$$

As a consequence, for any $j$, $s[-b][l_j = b]$ holds the same contents as either $s$ or $s[-l_j]$ and we have:

$$\mathcal{D}(t, s) \leq \mathcal{D}(t, s[-b][l_j = b]) \tag{68}$$

This can be extended to the execution time distribution of $t' = [[b], t]$. Since the property holds for any $j$, we expand the equation to a weighted sum across values of $j$:

$$\sum_{j \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t, s) \leq \sum_{j \in [1,N]} \frac{1}{N} \cdot \big(\mathcal{D}(t, s[-b][l_j = b])\big) \tag{69}$$

Since the selection of $j$ has no impact on the left-hand term, we have:

$$\mathcal{D}(t, s) \leq \sum_{j \in [1,N]} \frac{1}{N} \cdot \big(\mathcal{D}(t, s[-b][l_j = b])\big) \tag{70}$$

Because of the ordering between the hit and miss latencies, we can expand the equation by adding a hit and miss latencies respectively on the left and right hand-sides:

$$\mathcal{H} + \mathcal{D}(t, s) \leq \mathcal{M} + \sum_{j \in [1,N]} \frac{1}{N} \cdot \big(\mathcal{D}(t, s[-b][l_j = b])\big) \tag{71}$$

$\mathcal{H} + \mathcal{D}(t, s)$ corresponds to the execution time of trace $t$ after an initial hit in cache state $s$ as per (21). Similarly the right hand term corresponds to an initial

fist miss from the state $s[-b]$ before the execution of $t$. An access to $b$ fits these behaviour and can be incorporated into both terms:

$$\mathcal{D}([[b], t], s) \leq \mathcal{D}([[b], t], s[-b]) \tag{72}$$

Hence, $\forall t, \forall s, \mathcal{D}(t, s) \leq \mathcal{D}(t, s[-e])$ □

**Theorem 2** *The replacement of a random block in cache triggers at most one additional hit.*

*The distribution for any trace t from any cache state s is upper-bounded by the distribution for trace t after the replacement of a random block in s and assuming a single hit turns into a miss.*

$$\mathcal{H} + \mathcal{D}(t, s) \leq \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t, s[l_i = e]) \tag{23}$$

*Proof* The property trivially holds if $e$ is already present in the cache as the replacement then has no impact on the cache state, $s[l_i = e] = s$. We only consider input states $s$ where $e$ is absent and prove this property by induction.

- Base case $t = [b], b \neq e$: If $l_i \neq b$, input caches $s$ and $s[l_i = e]$ result in the same miss latency $\mathcal{M}$. If $e$ replaces $b$ in the cache, i.e. $l_i = b$, then the execution of $t$ is a miss from $s[l_i = e]$ and a hit from $s$. The property trivially holds.
- Base case $t = [e]$: The replacement of line $l_i$ by $e$, absent from the cache, implies that the execution of $t$ is a hit from $s[l_i = e]$ and a miss from $s$:

$$\mathcal{H} + \mathcal{M} = \mathcal{M} + \mathcal{H} \tag{73}$$

$$\mathcal{H} + \mathcal{D}(t, s) = \mathcal{M} + \mathcal{D}(t, s[l_i = e]) \tag{74}$$

The property holds for any $i$ and can be extended to the weighted sum over $i$:

$$\mathcal{H} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t, s) = \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t, s[l_i = e]) \tag{75}$$

The same distribution is weighted and summed $N$ times on the left-hand term, it can be simplified as such:

$$\mathcal{H} + \mathcal{D}(t, s) = \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t, s[l_i = e]) \tag{76}$$

- Inductive case $t = [[b], t']$: Assume the property holds for any trace $t'$ and any $x$:

$$\mathcal{H} + \mathcal{D}(t', s) \leq \mathcal{M} + \sum_{j \in [1,N]} \mathcal{D}(t', s[l_j = x]) \tag{77}$$

The execution time distribution of $t = [[b], t']$ from either $s$ or one of the $s[l_i = e]$ depends first on the presence or absence of the first accessed block $b$ in the cache. We consider all alternatives and expand the execution time distribution of the trace as per (21), $b = e$, $b \neq e \wedge b \in s$, and $b \neq e \wedge b \notin s$.

– $b = e$: The block is absent from the input cache state $s$ and results in a miss and the eviction of a line $l_j$:

$$\mathcal{H} + \mathcal{D}([[e], t'], s) = \mathcal{H} + \mathcal{M} + \sum_{j \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_j = e]) \tag{78}$$

When $e$ is randomly inserted in the cache before the execution of the same sequence, its presence in $s[l_i = e]$ results in a guaranteed hit from any of the $N$ possible states. The resulting cache states are left unchanged:

$$\mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}([[e], t'], s[l_i = e]) = \mathcal{M} + \mathcal{H} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_i = e]) \tag{79}$$

The two expanded distributions (78) and (79) obviously have the same behaviour. The additional miss and hit latencies respectively balance the guaranteed hit and miss, while the resulting cache states are the same. Hence:

$$\mathcal{H} + \mathcal{D}([[e], t'], s) = \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}([[e], t'], s[l_i = e]) \tag{80}$$

– $b \neq e$, $b \in s$: $b$ is present in the input cache state $s$. The corresponding execution time distribution from $s$ can simply be expressed as:

$$\mathcal{D}(t, s) = \mathcal{H} + \mathcal{D}(t', s) \tag{81}$$

From the input state $s[l_i = e]$, different cases have to be considered depending on whether $e$ replaced $b$ or not, that is respectively a guaranteed miss or a hit. Upon a hit in particular, the cache state is left unchanged and replacement of line $l_i$ can occur after or before the access without incidence:

$$\mathcal{D}(t, s[l_i = e]) = \begin{cases} \mathcal{M} + \sum_{j \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[b = e][l_j = b]) & \text{if } l_i = b \\ \mathcal{H} + \mathcal{D}(t', s[l_i = e]) & \text{otherwise} \end{cases} \tag{82}$$

We expand the definition (82) of the contribution of $t$ assuming a line $l_i$ was first replaced by $e$ in the cache. We sum the $N$ different terms resulting from the replacement of $b$ or one of the other block, as follows:

$$\sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t, s[l_i = e]) = \frac{1}{N} \cdot \left( \sum_{j \in [1,N]} \frac{1}{N} \cdot \left( \mathcal{M} + \mathcal{D}(t', s[b = e][l_j = b]) \right) \right)$$

$$+ \sum_{i \in [1,N] \setminus b} \frac{1}{N} \cdot \big(\mathcal{H} + \mathcal{D}(t', s[l_i = e])\big) \tag{83}$$

We deduce from the induction hypothesis (77) an upper bound $\mathcal{U}$ on the execution time distribution of $t'$ from $s$:

$$\mathcal{H} + \mathcal{D}(t', s) \leq \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_i = e]) \tag{84}$$

With the addition of a hit latency $\mathcal{H}$ on both sides:

$$\mathcal{H} + \mathcal{H} + \mathcal{D}(t', s) \leq \mathcal{H} + \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_i = e]) \tag{85}$$

$\mathcal{H}+\mathcal{D}(t', s)$ is equivalent to the execution time of $t$ from $s$ as expressed in (81):

$$\mathcal{H} + \mathcal{D}(t, s) \leq \mathcal{H} + \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_i = e]) \tag{86}$$

$$\mathcal{H} + \mathcal{D}(t, s) \leq \mathcal{M} + \mathcal{U} \tag{87}$$

We further distinguish in $\mathcal{U}$ the cases where $e$ specifically replaces $b$ in the cache or any other line:

$$\begin{aligned} \mathcal{U} &= \mathcal{H} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_i = e]) \\ &= \frac{1}{N} \cdot \big(\mathcal{H} + \mathcal{D}(t', s[b = e])\big) + \sum_{i \in [1,N] \setminus b} \frac{1}{N} \cdot \big(\mathcal{H} + \mathcal{D}(t', s[l_i = e])\big) \end{aligned} \tag{88}$$

Thanks to the induction hypothesis (77), we can define an upper-bound on the left-most term, where $e$ replaces $b$ in the cache:

$$\mathcal{H} + \mathcal{D}(t', s[b = e]) \leq \mathcal{M} + \sum_{j \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[b = e][l_j = b]) \tag{89}$$

Multiplying both sides by $\frac{1}{N}$, we have:

$$\frac{1}{N} \cdot \big(\mathcal{H} + \mathcal{D}(t', s[b = e])\big) \leq \frac{1}{N} \cdot \left( \sum_{j \in [1,N]} \frac{1}{N} \cdot \big(\mathcal{M} + \mathcal{D}(t', s[b = e][l_j = b])\big) \right) \tag{90}$$

We can then deduce that $\mathcal{U}$ is a lower bound on the execution time distribution of $t$ when $e$ first replaces a random line $l_i$ in $s$. From (88) and (90):

$$\mathcal{U} \leq \frac{1}{N} \cdot \left( \sum_{j_\in[1,N]} \frac{1}{N} \cdot \left(\mathcal{M} + \mathcal{D}(t', s[b = e][l_j = b])\right) \right) + \sum_{i \in [1,N] \setminus b} \frac{1}{N} \cdot \left(\mathcal{H} + \mathcal{D}(t', s[l_i = e])\right) \tag{91}$$

From (83), it follows that:

$$\mathcal{U} \leq \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t, s[l_i = e]) \tag{92}$$

This can be combined with (87) such that $\mathcal{M} + \mathcal{U}$ is an intermediate bound between (81) and (82):

$$\mathcal{H} + \mathcal{D}(t, s) \leq \mathcal{M} + \mathcal{U} \leq \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t, s[l_i = e]) \tag{93}$$

Hence the property holds:

$$\mathcal{H} + \mathcal{D}(t, s) \leq \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t, s[l_i = e]) \tag{94}$$

– $b \neq e, b \notin s$: We need to consider two subcases depending on whether the random insertion of $b$ or $e$ results in the higher execution time distribution for $t'$, i.e. the comparison between $\mathcal{D}_b$ and $\mathcal{D}_e$:

$$\mathcal{D}_b = \sum_{i \in [1,n]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_i = b]) \tag{95}$$

$$\mathcal{D}_e = \sum_{i \in [1,n]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_i = e]) \tag{96}$$

- $\mathcal{D}_e \leq \mathcal{D}_b$: We prove the existence of an intermediate bound between $\mathcal{D}(t, s)$ and $\mathcal{D}(t, s[l_i = e])$ where $t = [[b], t']$. From the induction hypothesis (77), we deduce that:

$$\mathcal{H} + \mathcal{D}(t', s[l_j = b]) \leq \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_j = b][l_i = e]) \tag{97}$$

Let us define $U_j$ such that:

$$U_j = \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_j = b][l_i = e]) \tag{98}$$

$$\mathcal{H} + \mathcal{D}(t', s[l_j = b]) \leq \mathcal{M} + U_j \tag{99}$$

The property further holds for any $j$ and extends to the weighted sum over $j$ of the terms on each side of the inequality:

$$\mathcal{H} + \sum_{j \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_j = b]) \leq \mathcal{M} + \sum_{j \in [1,N]} \frac{1}{N} \cdot U_j \tag{100}$$

Using the definition of $U_j$ (98), the right-hand term can be expanded, by distinguishing the cases where $j$ and $i$ denote the same line, that is when $e$ replaces the randomly inserted $b$, $s[l_j = b][l_j = e] = s[l_j = e]$:

$$\begin{aligned} \sum_{j \in [1,N]} \frac{1}{N} \cdot U_j = & \sum_{j \in [1,N]} \frac{1}{N} \cdot \frac{1}{N} \cdot \mathcal{D}(t', s[l_j = e]) \\ & + \sum_{j \in [1,N]} \frac{1}{N} \cdot \sum_{i \in [1,N] \setminus j} \frac{1}{N} \cdot \mathcal{D}(t', s[l_j = b][l_i = e]) \end{aligned} \tag{101}$$

By substituting $\mathcal{D}_e$ (96) we have:

$$\sum_{j \in [1,N]} \frac{1}{N} \cdot U_j = \frac{1}{N} \cdot \mathcal{D}_e + \sum_{j \in [1,N]} \frac{1}{N} \cdot \sum_{i \in [1,N] \setminus j} \frac{1}{N} \cdot \mathcal{D}(t', s[l_j = b][l_i = e]) \tag{102}$$

Similarly, we distinguish in $\mathcal{D}([[b], t'], s[l_i = e])$ the same cases where the first access to $b$, a miss, replaces line $l_i$ or not:

$$\begin{aligned} & \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}([[b], t'], s[l_i = e]) \\ & = \mathcal{M} + \sum_{j \in [1,N]} \frac{1}{N} \cdot \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_i = e][l_j = b]) \\ & = \mathcal{M} + \sum_{j \in [1,N]} \frac{1}{N} \cdot \frac{1}{N} \cdot \mathcal{D}(t', s[l_j = b]) \\ & \quad + \sum_{j \in [1,N]} \frac{1}{N} \cdot \sum_{i \in [1,N] \setminus j} \frac{1}{N} \cdot \mathcal{D}(t', s[l_i = e][l_j = b]) \end{aligned} \tag{103}$$

By substituting $\mathcal{D}_b$ (95) in the above equation we get:

$$\sum_{i\in[1,N]} \frac{1}{N} \cdot \mathcal{D}([[b], t'], s[l_i = e]) \\ = \mathcal{M} + \frac{1}{N} \cdot \mathcal{D}_b + \sum_{j\in[1,N]} \frac{1}{N} \cdot \sum_{i\in[1,N]\setminus j} \frac{1}{N} \cdot \mathcal{D}(t', s[l_i = e][l_j = b]) \tag{104}$$

When lines $i$ and $j$ do not match, the ordering of the replacement of $l_i$ and $l_j$ by $e$ and $b$ respectively is irrelevant, $s[l_i = e][l_j = b] = s[l_j = b][l_i = e]$. Hence the difference between (104) and the sum of $U_j$ (102) depends on the ordering between respectively $\mathcal{D}_b$ and $\mathcal{D}_e$. Since $\mathcal{D}_e \leq \mathcal{D}_b$, it follows from (104) and (102) that:

$$\mathcal{M} + \sum_{j\in[1,N]} \frac{1}{N} \cdot U_j \leq \sum_{i\in[1,N]} \frac{1}{N} \cdot \mathcal{D}([[b], t'], s[l_i = e]) \tag{105}$$

As a consequence of (105) and (100), it follows that:

$$\mathcal{H} + \sum_{j\in[1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_j = b]) \leq \mathcal{M} + \sum_{j\in[1,N]} \frac{1}{N} \cdot U_j \\ \leq \sum_{i\in[1,N]} \frac{1}{N} \cdot \mathcal{D}([[b], t'], s[l_i = e]) \tag{106}$$

Adding a miss latency $\mathcal{M}$ on both sides of the inequality:

$$\mathcal{M}+\mathcal{H}+\sum_{j\in[1,N]} \frac{1}{N}\cdot\mathcal{D}(t', s[l_j = b]) \leq \mathcal{M}+\sum_{i\in[1,N]} \frac{1}{N}\cdot\mathcal{D}([[b], t'], s[l_i = e]) \tag{107}$$

The left-hand term collapses to the execution time distribution of $t = [[b], t']$ from $s$ as per (21) as $b$ is absent from the input cache state $s$. Hence, the property holds:

$$\mathcal{H} + \mathcal{D}([[b], t'], s) \leq \mathcal{M} + \sum_{i\in[1,N]} \frac{1}{N} \cdot \mathcal{D}([[b], t'], s[l_i = e]) \tag{108}$$

- $\mathcal{D}_b \leq \mathcal{D}_e$: The induction hypothesis (77) gives us the following relationship:

$$\mathcal{H} + \mathcal{D}(t', s[l_i = e]) \leq \mathcal{M} + \sum_{j\in[1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_i = e][l_j = b]) \tag{109}$$

We can reduce the right-hand term as per (21) given that $b$ is absent from the initial cache state $s$:

$$\mathcal{H} + \mathcal{D}(t', s[l_i = e]) \leq \mathcal{D}([[b], t'], s[l_i = e]) \tag{110}$$

The property, valid for any line $l_i$, holds for summation below:

$$\mathcal{H} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_i = e]) \leq \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}([[b], t'], s[l_i = e]) \tag{111}$$

Substituting $\mathcal{D}_e$ (96), we have:

$$\mathcal{H} + \mathcal{D}_e \leq \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}([[b], t'], s[l_i = e]) \tag{112}$$

Considering the ordering $\mathcal{D}_b \leq \mathcal{D}_e$ between $\mathcal{D}_b$ (95) and $\mathcal{D}_e$ (96), we conclude that:

$$\mathcal{H} + \mathcal{D}_b \leq \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}([[b], t'], s[l_i = e]) \tag{113}$$

Through the expansion of $\mathcal{D}_b$ (95) and the addition of a miss latency $\mathcal{M}$ on both sides of the inequality, we have:

$$\mathcal{M} + \mathcal{H} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}(t', s[l_i = b]) \leq \mathcal{M} + \sum_{i \in [1,N]} \mathcal{D}([[b], t'], s[l_i = e]) \tag{114}$$

From (21) and the absence of $b$ from the input cache state $s$ we know that the left hand term can be expressed as the execution time of $t = [[b], t']$ from $s$:

$$\mathcal{H} + \mathcal{D}([[b], t'], s) \leq \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}([[b], t'], s[l_i = e]) \tag{115}$$

The same distribution might not be the dominant one on the whole input domain; there might be segments where $\mathcal{D}_e$ is greater than $\mathcal{D}_b$ and the converse is true on the rest of the input domain. However, the property holds in either case. Hence the theorem still holds on each segment.

The property holds in all scenarios, whether $b = e$, or block $b$ is absent or present in input cache state $s$. The random replacement of a line $l_i$ by $e$ can trigger an additional hit on the first subsequent access to $e$. The additional miss latency compensates for this potential hit. From the original cache state, this access is a guaranteed miss. The resulting cache states, and the behaviour of the rest of the sequence, match whether this first access to $e$ results in a cache hit (from $s[l_i = e]$) or a miss (from $s$). □

**Lemma 3** *The replacement in input cache state $s$ of a block by another one in trace $t$ has no impact, timing and cache contents-wise, up to the first access to either block. The replacement can occur indifferently before trace $t$ or before the first access to either block.*

$$\begin{aligned}
frd(b,t) &\leq frd(e,t) \leq \infty \wedge t = [t_p, [b], t_s] \wedge b \notin t_p \Rightarrow \\
\mathcal{D}(t, s[b=e]) &= \sum_{(C',P',\mathcal{D}') \in outcomes(t_p, s[b=e])} P' \cdot \left(\mathcal{D}' \otimes \mathcal{D}\left([[b], t_s], C'\right)\right) \\
&= \sum_{(C,P,\mathcal{D}) \in outcomes(t_p, s)} P \cdot (\mathcal{D} \otimes \mathcal{D}([[b], t_s], C[b=e])) \\
\mathcal{D}(t, s[e=b]) &= \sum_{(C',P',\mathcal{D}') \in outcomes(t_p, s[e=b])} P' \cdot \left(\mathcal{D}' \otimes \mathcal{D}\left([[b], t_s], C'\right)\right) \\
&= \sum_{(C,P,\mathcal{D}) \in outcomes(t_p, s)} P \cdot (\mathcal{D} \otimes \mathcal{D}([[b], t_s], C[e=b])) \quad (116)
\end{aligned}$$

*Proof* The property trivially holds if the input cache state $s$ holds both $b$ and $e$ or neither as the replacement are then ineffective. We focus on states which hold either one but not both. $s'$ denotes the input cache where the replacement occurred, $s' = s[b=e]$ or $s' = s[e=b]$

The trace $t$ can be divided as such $t = [t_p, [b], t_s]$ where $[b]$ is the first reference to $b$ in $t$. The subtrace $t_p$ holds no reference to $b$, nor to $e$ as a consequence of the ordering between their forward reuse distances. The execution time distribution of trace $t$ as per (21) is:

$$\mathcal{D}(t,s) = \sum_{(C,P,\mathcal{D}) \in outcomes(t_p, s)} P \cdot (\mathcal{D} \otimes \mathcal{D}([[b], t_s], C)) \quad (117)$$

Accesses in $t_p$ are not impacted by the presence of either $b$ or $e$ in the input cache. The sequence of evictions from $s$ which lead to cache state $C$ with probability $P$ and execution time distribution $\mathcal{D}$ is matched starting from $s'$. It results in cache state $C'$ with the same probability $P$ and execution time distribution $\mathcal{D}$. If the replaced block is absent from $C$, it has been evicted by accesses in $t_p$ and similarly the replacing block has been evicted in $C'$. If the replaced block is still present in $C$, the replacing block is similarly present in $C'$. The other lines hold the same contents since we consider the same fixed sequence of evictions on $t_p$ from $s$ and $s'$. □

**Theorem 3** *The replacement of a block in input cache state $s$ by one which is reused later in trace $t$ cannot result in a decreased execution time distribution: $frd(b,t) \leq frd(e,t) \leq \infty \wedge b \in s \wedge e \notin s \Rightarrow \mathcal{D}(t,s) \leq \mathcal{D}(t, s[b=e])$*

*Proof* If there is no reference to memory block $e$ in the considered trace $t$, the replacement of $b$ by $e$ in input cache state $s$ is equivalent to the eviction of $b$ from the cache, $e \notin t \Rightarrow s[b=e] = s[-b]$. The theorem then holds as per Theorem 1. We therefore focus on the case where $e$ is accessed in $t$, $frd(e,t) \neq \infty$.

We cut the trace $t$ into different segments $t = [t_p, [b], t_m, [e], t_s]$ such that $t_p$ holds no reference to $b$ nor $e$ as a consequence of their forward reuse distances. Similarly, we define $t_m$ such that it holds no reference to $e$. The first reference to $b$ and $e$ in trace $t$ are respectively located after $t_p$ and $t_m$.

Because of Lemma 3, we know that the replacement has no impact on $t_p$ which holds no reference to either the replaced block $b$ or the replacing one $e$. We focus on the execution time distribution of the trace $t' = [[b], t_m, [e], t_s]$ from a state $C_b$, which holds $b$ but not $e$. We further prove by induction that:

$$\mathcal{D}(t', C_b) \leq \mathcal{D}(t', C_b[b = e]) \tag{118}$$

Base case $t_m = t_s = \emptyset, t' = [b, e]$: The property trivially holds as the execution of $t'$ from $C_b$ results in a hit then a miss, $\mathcal{D}(t', C_b) = \mathcal{H} + \mathcal{M}$, whereas it misses then may hit or miss from input $C_b[b = e]$, $\mathcal{D}(t', C_b[b = e]) \geq \mathcal{M} + \mathcal{H}$.

Inductive case $t' = [[b], t_m, [e], t_s]$: Suppose the property $\mathcal{D}(t'', C) \leq \mathcal{D}(t'', C[x = y])$ holds for any trace $t'' = [[x], t''_m, [y], t''_s]$ where $t''_m$ does not access $y$ and any input state $C$ which does not hold $y$. From Lemma 3, this hypothesis applies to arbitrary prefixes $t''_p$ as long as they hold neither $x$ nor $y$:

$$\begin{aligned} x \notin t''_p \wedge y \notin t''_p \wedge y \notin t''_m \wedge y \notin C \Rightarrow \mathcal{D}([t''_p, [x], t''_m, [y], t''_s], C) \\ \leq \mathcal{D}([t''_p, [x], t''_m, [y], t''_s], C[x = y]) \end{aligned} \tag{119}$$

The first access to $b$ in $t'$ is a guaranteed hit from $C_b$ and a miss from $C_b[b = e]$. The resulting execution time distributions can be expressed as per (21):

$$\mathcal{D}(t', C_b) = \mathcal{H} + \mathcal{D}([t_m, [e], t_s], C_b) \tag{120}$$

$$\mathcal{D}(t', C_b[b = e]) = \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}([t_m, [e], t_s], C_b[b = e][l_i = b]) \tag{121}$$

– If $b$ is present in $t_m$, there is an access to $b$ prior to the first access to $e$ in the remaining trace. $[t_m, [e], t_s]$ can be further split into $[t'_m, [b], t''_m, [e], t_s]$ such that $t_m = [t'_m, [b], t''_m]$ and $t'_m$ holds no reference to $b$ nor $e$. There is a reference to $b$ before the next access to $e$. From the induction hypothesis (119), substituting $b$ for $x$ and $e$ for $y$, we have:

$$\mathcal{D}([t_m, [e], t_s], C_b) \leq \mathcal{D}([t_m, [e], t_s], C_b[b = e]) \tag{122}$$

From Theorem 2, we know that:

$$\mathcal{H} + \mathcal{D}([t_m, [e], t_s], C_b[b = e]) \leq \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}([t_m, [e], t_s], C_b[b = e][l_i = b]) \tag{123}$$

Hence from (122) and (123) the property trivally holds when $b$ is in $t_m$ using $\mathcal{D}([t_m, [e], t_s], C_b[b = e])$ as an intermediate bound:

$$\begin{aligned}\mathcal{H} + \mathcal{D}([t_m, [e], t_s], C_b) &\leq \mathcal{H} + \mathcal{D}([t_m, [e], t_s], C_b[b = e]) \\ &\leq \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}([t_m, [e], t_s], C_b[b = e][l_i = b])\end{aligned} \tag{124}$$

The leftmost and rightmost terms can be reduced to the property of interest using respectively (120) and (121):

$$\mathcal{D}(t', C_b) \leq \mathcal{D}(t', C_b[b = e]) \tag{125}$$

– Now consider the case where $b$ is absent from the trace $t_m$ as is $e$. We distinguish the case where the first miss from $C_b[b = e]$ in $t' = [[b], t_m, [e], t_s]$ selects the line that originally held $b$, $C_b[b = e][l_i = b] = C_b$, from the ones where a different line is selected. The latter results in a cache state equivalent to $C_b[l_i = e]$. By separating those cases in (21), we have:

$$\begin{aligned}\mathcal{D}(t', C_b[b = e]) &= \mathcal{M} + \frac{1}{N} \cdot \mathcal{D}([t_m, [e], t_s], C_b) \\ &\quad + \sum_{i \in [1,N] \setminus b} \frac{1}{N} \cdot \mathcal{D}([t_m, [e], t_s], C_b[l_i = e])\end{aligned} \tag{126}$$

This allows the definition of a lower-bound $\mathcal{U}$ of the contribution of the complete trace from $C_b[b = e]$:

$$\mathcal{U} = \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}([t_m, [e], t_s], C_b[l_i = e]) \tag{127}$$

We further distinguish the case where $l_i$ holds $b$ from the others:

$$\mathcal{U} = \frac{1}{N} \cdot \mathcal{D}([t_m, [e], t_s], C_b[b = e]) + \sum_{i \in [1,N] \setminus b} \frac{1}{N} \cdot \mathcal{D}([t_m, [e], t_s], C_b[l_i = e]) \tag{128}$$

Since $t_m$ holds neither $b$ nor $e$, we deduce from the induction hypothesis that replacing $b$ by $e$ in the cache does not degrade the execution time distribution of the trace $[t_m, [e], t_s]$ from $C_b[b = e]$:

$$\mathcal{D}([t_m, [e], t_s], C_b[b = e]) \leq \mathcal{D}([t_m, [e], t_s], C_b[b = e][e = b]) \tag{129}$$

Replacing $b$ by $e$ then $e$ by $b$ has no impact on the cache contents, $C_b[b = e][e = b] = C_b$:

$$\mathcal{D}([t_m, [e], t_s], C_b[b = e]) \leq \mathcal{D}([t_m, [e], t_s], C_b) \tag{130}$$

Dividing both sides by $N$ we get:

$$\frac{1}{N} \cdot (\mathcal{D}([t_m, [e], t_s], C_b[b = e])) \leq \frac{1}{N} \cdot (\mathcal{D}([t_m, [e], t_s], C_b)) \tag{131}$$

We add the same factor, the random replacement of a line other than $b$, on both sides:

$$\begin{aligned} &\frac{1}{N} \cdot \mathcal{D}([t_m, [e], t_s], C_b[b = e]) + \sum_{i \in [1,N] \setminus b} \frac{1}{N} \cdot \mathcal{D}([t_m, [e], t_s], C_b[l_i = e]) \\ &\leq \frac{1}{N} \cdot \mathcal{D}([t_m, [e], t_s], C_b) + \sum_{i \in [1,N] \setminus b} \frac{1}{N} \cdot \mathcal{D}([t_m, [e], t_s], C_b[l_i = e]) \end{aligned} \tag{132}$$

This equation orders the expanded form of $\mathcal{M}+\mathcal{U}$ in (128) and that of $\mathcal{D}(t', C_b[b = e])$ in (126):

$$\mathcal{M} + \mathcal{U} \leq \mathcal{D}(t', C_b[b = e]) \tag{133}$$

From Theorem 2, we can compare the bound $\mathcal{U}$ to the execution time distribution of $t'$ from $C_b$:

$$\mathcal{H} + \mathcal{D}([t_m, [e], t_s], C_b) \leq \mathcal{M} + \sum_{i \in [1,N]} \frac{1}{N} \cdot \mathcal{D}([t_m, [e], t_s], C_b[l_i = b]) \tag{134}$$

The rightmost term collapses to $\mathcal{M} + \mathcal{U}$ from (128):

$$\mathcal{H} + \mathcal{D}([t_m, [e], t_s], C_b) \leq \mathcal{M} + \mathcal{U} \tag{135}$$

Hence, from this equation and (133) the property holds:

$$\mathcal{H} + \mathcal{D}([t_m, [e], t_s], C_b) \leq \mathcal{M} + \mathcal{U} \leq \mathcal{D}(t', C_b[b = e]) \tag{136}$$

$$\mathcal{D}(t', C_b) \leq \mathcal{D}(t', C_b[b = e]) \tag{137}$$

□

**Lemma 1** *The convolution operation preserves the ordering between execution time distributions:*

$$\mathcal{D} \leq \mathcal{D}' \Rightarrow \mathcal{D} \otimes \mathcal{A} \leq \mathcal{D}' \otimes \mathcal{A}$$

*Proof* Let us first assume that $\mathcal{D} \leq \mathcal{D}$. This relation implies that $\mathcal{D}'$ is greater than $\mathcal{D}$, more formally:

$$\forall v,\ P(\mathcal{D} \geq v) \leq P(\mathcal{D}' \geq v) \tag{138}$$

This property applies to the sum of probabilities for all values greater than $v$:

$$\forall v, \sum_{x=v}^{+\infty} \mathcal{D}(x) \leq \sum_{x=v}^{+\infty} \mathcal{D}'(x) \tag{139}$$

It can be in particular extended to any value $(v-k)$:

$$\forall v, \forall k, \sum_{x=(v-k)}^{+\infty} \mathcal{D}(x) \leq \sum_{x=(v-k)}^{+\infty} \mathcal{D}'(x) \tag{140}$$

As we are considering the sum to infinity of values $\mathcal{D}(x)$, $k$ can be subtracted indifferently from $x$ or its lower bound $v$:

$$\forall v, \forall k, \sum_{x=(v-k)}^{+\infty} \mathcal{D}(x) = \sum_{x=v}^{+\infty} \mathcal{D}(x-k) \tag{141}$$

From the two previous equations, we have:

$$\forall v, \forall k, \sum_{x=v}^{+\infty} \mathcal{D}(x-k) \leq \sum_{x=v}^{+\infty} \mathcal{D}'(x-k) \tag{142}$$

The occurrence probability of any element $x$ in a distribution $A$ is by definition a positive number, $A(k) \geq 0$. We can factor the same both sides of the inequality with the same values $A(k)$:

$$\forall v, \forall k, \mathcal{A}(k) \cdot \sum_{x=v}^{+\infty} \mathcal{D}(x-k) \leq \mathcal{A}(k) \cdot \sum_{x=v}^{+\infty} \mathcal{D}'(x-k) \tag{143}$$

$$\forall v, \forall k, \sum_{x=v}^{+\infty} \mathcal{A}(k) \cdot \mathcal{D}(x-k) \leq \sum_{x=v}^{+\infty} \mathcal{A}(k) \cdot \mathcal{D}'(x-k) \tag{144}$$

As the inequality holds for any element $k$, it holds for their overall sum over $k$:

$$\forall v, \sum_{k=-\infty}^{+\infty} \sum_{x=v}^{+\infty} \mathcal{A}(k) \cdot \mathcal{D}(x-k) \leq \sum_{k=-\infty}^{+\infty} \sum_{x=v}^{+\infty} \mathcal{A}(k) \cdot \mathcal{D}'(x-k) \tag{145}$$

Thanks to the commutativity of the sum operands, we have:

$$\forall v, \sum_{x=v}^{+\infty} \sum_{k=-\infty}^{+\infty} \mathcal{A}(k) \cdot \mathcal{D}(x-k) \leq \sum_{x=v}^{+\infty} \sum_{k=-\infty}^{+\infty} \mathcal{A}(k) \cdot \mathcal{D}'(x-k) \tag{146}$$

Both terms of the inequality correspond to the convolution of distributions as defined in (20):

$$\forall v, \sum_{x=v}^{+\infty} (\mathcal{A} \otimes \mathcal{D})(x) \leq \sum_{x=v}^{+\infty} (\mathcal{A} \otimes \mathcal{D}')(x) \tag{147}$$

This defines an order between the result of the convolution of $\mathcal{D}$ and $\mathcal{D}'$ with distribution $\mathcal{A}$:

$$\forall v,\ P((\mathcal{A} \otimes \mathcal{D}) \geq v) \leq P((\mathcal{A} \otimes \mathcal{D}') \geq v) \tag{148}$$

$$\mathcal{A} \otimes \mathcal{D} \leq \mathcal{A} \otimes \mathcal{D}' \tag{149}$$

Per commutativity of the convolution operator $\otimes$, we have:

$$\mathcal{D} \leq \mathcal{D}' \Rightarrow \mathcal{D} \otimes \mathcal{A} \leq \mathcal{D}' \otimes \mathcal{A} \tag{150}$$

□

**Lemma 2** *The contributions of merged sets of cache states S and A is the sum of their individual contributions:*

$$\forall t, \mathcal{D}(t, S) + \mathcal{D}(t, A) = \mathcal{D}(t, S \uplus A)$$

*Proof* $S \uplus A$ can be divided into three categories, cache states $C$ that exist only in $S$, only $A$ or in both, denoted respectively $Only_S$, $Only_A$, $Com_{(S,A)}$. The contribution of states in $Only_S$ and $Only_A$ is unchanged by the merge operation. Only states in $Com_{(S,A)}$ are subject to the weighted merge in (6). We focus on proving the equivalence between the contribution of $Com_{(S,A)}$ and that of original states from $A$ and $S$ respectively $Com_A$ and $Com_S$, $Com_{(S,A)} = Com_S \uplus Com_A$.

Each state in $Com_{(S,A)}$ is the combination of corresponding states from $Com_S$ and $Com_A$. Without loss of generality, we assume there is a single matching state in $Com_S$ and $Com_A$ for each merged one in $Com_{(S,A)}$:

$$\forall (C, P, \mathcal{D}) \in Com_{(S,A)}, \exists (C, P_A, \mathcal{D}_A) \in Com_A \wedge \exists (C, P_S, \mathcal{D}_S) \in Com_S \wedge P = P_A + P_S \wedge \mathcal{D} = (\frac{P_A}{P} \cdot \mathcal{D}_A) + (\frac{P_S}{P} \cdot \mathcal{D}_S) \tag{151}$$

We can express the execution time contribution of $Com_{(S,A)}$ as per (37):

$$\mathcal{D}(t, Com_{(S,A)}) = \sum_{(C,P,\mathcal{D}) \in Com_{(S,A)}} P \cdot (\mathcal{D} \otimes \mathcal{D}(t, C)) \tag{152}$$

By replacing each merged distribution $\mathcal{D}$ with the original distributions and probabilities from $S$ and $A$, we have:

$$\mathcal{D}(t, Com_{(S,A)}) = \sum_{(C,P,\mathcal{D}) \in Com_{(S,A)}} P \cdot \left( (\frac{P_A}{P} \cdot \mathcal{D}_A) + (\frac{P_S}{P} \cdot \mathcal{D}_S) \right) \otimes \mathcal{D}(t, C) \tag{153}$$

By definition, the convolution of distributions and the multiplication of a distribution by a constant are associative operations, $P \cdot (\mathcal{D} \otimes \mathcal{D}') = (P \cdot \mathcal{D}) \otimes \mathcal{D}'$. We can therefore factor $P$ inside the merged distributions:

$$\mathcal{D}(t, Com_{(S,A)}) = \sum_{(C,P,\mathcal{D})\in Com_{(S,A)}} \left( P \cdot (\frac{P_A}{P} \cdot \mathcal{D}_A) + P \cdot (\frac{P_S}{P} \cdot \mathcal{D}_S) \right) \otimes \mathcal{D}(t, C) \tag{154}$$

$$\mathcal{D}(t, Com_{(S,A)}) = \sum_{(C,P,\mathcal{D})\in Com_{(S,A)}} ((P_A \cdot \mathcal{D}_A) + (P_S \cdot \mathcal{D}_S)) \otimes \mathcal{D}(t, C) \tag{155}$$

This equation can be refined into the contribution of states in $Com_S$ and $Com_A$ as follows:

$$\mathcal{D}(t, Com_{(S,A)}) = \sum_{(C,P,\mathcal{D})\in Com_{(S,A)}} (P_A \cdot \mathcal{D}_A) \otimes \mathcal{D}(t, C) + (P_S \cdot \mathcal{D}_S) \otimes \mathcal{D}(t, C) \tag{156}$$

$$\mathcal{D}(t, Com_{(S,A)}) = \sum_{(C,P_A,\mathcal{D}_A)\in Com_A} (P_A \cdot \mathcal{D}_A) \otimes \mathcal{D}(t, C) + \sum_{(C,P_S,\mathcal{D}_S)\in Com_S} (P_S \cdot \mathcal{D}_S) \otimes \mathcal{D}(t, C) \tag{157}$$

$$\mathcal{D}(t, Com_{(S,A)}) = \mathcal{D}(t, Com_A) + \mathcal{D}(t, Com_S) \tag{158}$$

□

**Theorem 8** (Renamed path ordering) *Given a path $\pi$ divided into three sub-paths $\pi = [\pi_S, \pi_V, \pi_E]$, where $\pi_V = [e, v_1, \ldots, v_k, e]$. The pWCET of $\pi$ is smaller than or equal to that of the renamed sequence $\pi_r = [\pi_S, \pi_V(e \rightarrow b), \pi_E]$, $\mathcal{D}(\pi) \leq \mathcal{D}(\pi_r)$, if:*

- *there is no access to $b$ in $\pi_V$;*
- *the reuse distance of $e$ before $\pi_V$ is smaller than that of $b$ at this point;*
- *the forward reuse distance of $e$ at the end of $\pi_V$ is smaller than that of $b$ at this point.*

*Proof* We focus on the behaviour of the execution time distribution of the path $\pi$ and its renamed alternative starting from the empty cache state, since is known to result in the worst execution time distribution over any other input state. Any valid pWCET must upper-bound this distribution, hence $\mathcal{D}(\emptyset, \pi)$ is a tight pWCET for path $\pi$.

The execution of path $\pi_S$ generates an ensemble $outcomes(\pi_S, \emptyset)$ of cache states $C$. To each is attached an associated execution time distribution $\mathcal{D}$, corresponding to the hit and miss latencies of prior accesses, and an occurrence probability $P$. The renaming does not impact the behaviour of accesses in $\pi_S$, therefore $outcomes(\pi_S, \emptyset)$ is left unchanged.

The execution time distribution of the renamed segment $\pi_V(e \rightarrow b)$ is no greater nor smaller than that of $\pi_V$ from those cache states that hold neither $b$ or $e$, or hold both, $(e \in s \wedge b \in s) \vee (e \notin s \wedge b \notin s) \Rightarrow \mathcal{D}(\pi_V(e \rightarrow b), s) = \mathcal{D}(\pi_V)$. States that hold neither $b$ nor $e$ result in the same hit and miss events on both paths except that $b$ replaces $e$ on the renamed path. This also produces the same cache states but where $b$ replaces $e$ after the renamed segment. As for states that hold both $b$ and $e$, events

which impact the line where $b$ is held on the original path are as likely to impact that of $e$ on the renamed one and vice-versa, e.g. the eviction of $b$ on the original corresponds to that of $e$ on the renamed one. This also results in the same cache states where $b$ replaces $e$ on the renamed path. The outcomes on $\pi_V$ then match the ones on $\pi_V(e \rightarrow b)$ where $b$ replaces $e$ in the cache.

When both blocks $b$ and $e$ are present in the outcomes on $\pi_V$ they match the ones on the renamed path $\pi_V(e \rightarrow b)$. The execution time distribution of the last segment $\pi_S$ is the same in either case. When $e$ is in the cache without $b$ after $\pi_V$, it is matched by a state after $\pi_V(e \rightarrow b)$ where $b$ replaces $e$. From the *Suffix ordering* condition the first access to $e$ in $\pi_S$ is before the first access to $b$, $frd(e, \pi_S) < frd(b, \pi_S)$. Theorem 3 applies; the execution of $\pi_S$ after $\pi_V$, when $e$ is in cache but not $b$, results in an execution time distribution that is no greater than the one after $\pi_V(e \rightarrow b)$ when $b$ replaces $e$ in the cache. Note that $b$ cannot be in cache without $e$ after $\pi_V$ since the last access in $\pi_V$ targets $e$.

We now focus on the contribution of states which hold one of $e$ or $b$ but not both, and prove their contribution to the renamed path outweighs that of the original. $\mathcal{R}_e$ and $\mathcal{R}_b$ respectively distinguish between those states of $outcomes(\pi_S, \emptyset)$ which hold one of $e$ or $b$.

Base case $\pi_V = [e]$: Every state in $\mathcal{R}_b$ shares a common ancestor state with a state in $\mathcal{R}_e$ such that they hold the same contents but $e$ replaces $b$. Indeed because of the condition on *Prefix ordering*, all states in $\mathcal{R}_b$ come from states where $b$ and $e$ were held in the cache simultaneously, after the last access to $e$ in the prefix $\pi_S$. There is then a sequence of events which evicts $e$ from the cache while conserving $b$, hence resulting in a state belonging to $\mathcal{R}_b$. There is a matching sequence of events from this common ancestor which conserves $e$ instead of $b$. Simply assume that evictions on the line of $e$ target that of $b$ and vice-versa. The two sequences of events are exactly as likely to occur as there is no other access to either $b$ or $e$ from their common ancestor to the renamed segment.

Consider the following four scenarios for a state $s$ of $\mathcal{R}_b$:

1. $[e]$ executes from $s$, hence resulting in a miss and $N$ output cache states $s[l_i = e]$.
2. $[e]$ executes from the as likely $s[b = e]$ of $\mathcal{R}_e$, hence resulting in a hit and the output cache state $s[b = e]$.
3. $[b]$, the renamed sequence, executes from $s$, hence resulting in a hit and an output cache state $s$.
4. $[b]$ executes from $s[b = e]$, hence resulting in a miss and $N$ output cache states $s[b = e][l_i = b]$.

Scenarios 3 and 2 balance each other, resulting in a worse behaviour on the renamed sequence. Both suffer from the same execution latency for $\pi_V$. Because of the *Suffix ordering* condition on the ordering between the forward reuse distances of $b$ and $e$ and Theorem 3, the execution time distribution of $\pi_E$ is worse starting from $s$ than from $s[b = e]$.

A similar argument can be made for scenarios 1 and 4. Each line $l_i$ has the same probability to be selected for eviction in each scenario. If $l_i$ is the line that held $b$ in $s$, the output cache states in cases 1 and 4 respectively are $s[b = e]$ and $s[b = e][l_i = b] = s[b = e][e = b] = s$. As per Theorem 3, it results in execution time distributions

that are no lower for the renamed path than for the original one. If $l_i$ is another line, the resulting cache states, $s[l_i = e]$ and $s[b = e][l_i = b]$, hold the same contents, $s[b = e][l_i = b] = s[l_i = e]$, and result in the same execution time distribution for $\pi_E$.

As for the remaining states $C_e$ in $\mathcal{R}_e$, the ones which do not mirror a state in $\mathcal{R}_b$ they respectively result in a hit on the original segment and a miss on the renamed one. On the renamed path, this results in the replacement of a line $l_i$ by $b$. In other words, $\mathcal{D}([\pi_V, \pi_E], C_e) = \mathcal{H} + \mathcal{D}(\pi_E, s)$ and $\mathcal{D}([\pi_V(e \rightarrow b), \pi_E], C_e) = \mathcal{M} + \sum_{i \in [1,N]} \mathcal{D}(\pi_E, C_e[l_i = b])$. The execution time distribution of the original path from $C_e$ is therefore no greater than that of the renamed path according to Theorem 2.

Overall the execution of the renamed path $[b]$ results in execution time distributions that are no lower than those obtained through the execution of the original one $[e]$.

General case $\pi_V = [e, v_1, \ldots, v_k, e]$: The arguments for the basic case can be extended to the general case where $\pi_V$ holds multiple accesses. The key observation is that the renaming has no impact on the reuse distance of accesses within $\pi_V$ except for the first. As in the base case, we focus on the contribution of states which hold one of $b$ or $e$. Consider the same four scenarios for input state $s \in \mathcal{R}_b$ of $\pi_V = [e, v_1, \ldots, v_k, e]$:

1. $[e, v_1, \ldots, v_k, e]$ executes from $s$, hence resulting in a first miss and $N$ cache states $s[l_i = e]$.
2. $[e, v_1, \ldots, v_k, e]$ executes from $s[b = e]$ of $\mathcal{R}_e$, hence resulting in a first hit and cache state $s[b = e]$.
3. $[b, v_1, \ldots, v_k, b]$ executes from $s$, hence resulting in a first hit and cache state $s$.
4. $[b, v_1, \ldots, v_k, b]$ executes from $s[b = e]$, hence resulting in a first miss and $N$ cache states $s[b = e][l_i = b]$.

From scenario 1 to 4, and scenario 2 to 3, $b$ simply replaces $e$ in both cache contents and trace of accesses $\pi_V$. The behaviour of the first access in $\pi_V$ is the same in either the original or the renamed path, and there is no more misses on the original than on the renamed path since they have the same initial contents and trace where $b$ simply replaces $e$. The reuse distance of all accesses but the first is left unchanged between these pairs of scenarios, $\mathcal{D}(\pi_V, s) = \mathcal{D}(\pi_V(e \rightarrow b), s[b = e])$ and $\mathcal{D}(\pi_V, s[b = e] = \mathcal{D}(\pi_V(e \rightarrow b), s)$.

The resulting cache states after $\pi_V(e \rightarrow b)$ also match the ones after $\pi_V$ with $b$ replacing $e$ in cache. Because of the *Suffix ordering* condition, the first access to $b$ in $\pi_E$ is preceded by an earlier access to $e$. Hence from Theorem 3 we have that the execution of $\pi_E$ after $\pi_V$ results in an execution time distribution that is no greater than the one starting from the matching input state $s'[e = b]$ after the renamed path $\pi_V(e \rightarrow b)$.

For some cache states $C_e$ in $\mathcal{R}_e$, the input states of $\pi_V$ which hold $e$ but not $b$, do not mirror a state in $\mathcal{R}_b$. The first access in $\pi_V(e \rightarrow b)$ is a miss from $C_e$. This intuitively increases the reuse distance of the remaining accesses in the renamed $[\pi_V(e \rightarrow b), \pi_E]$ over the original trace $[\pi_V, \pi_E]$. We prove by induction that:

$$\mathcal{D}([\pi_V, \pi_E], C_e) \leq \mathcal{D}([\pi_V(e \rightarrow b), \pi_E], C_e) \tag{159}$$

The base case, when $\pi_V$ holds a single access to $e$, has already been proved thanks to Theorem 2. Our induction hypothesis is, assuming $\pi'_V$ is a subtrace of $\pi_V$:

$$\mathcal{D}([\pi'_V, \pi_E], C_e) \leq \mathcal{D}([\pi'_V(e \to b), \pi_E], C_e) \tag{160}$$

From Theorem 2, we have:

$$\begin{aligned} &\mathcal{H} + \mathcal{D}([[v_1, \ldots, v_k, e](e \to b), \pi_E], C_e) \\ &\leq \mathcal{M} + \sum_{i \in [1,N]} \mathcal{D}([[v_1, \ldots, v_k, e](e \to b), \pi_E], C_e[l_i = b]) \end{aligned} \tag{161}$$

The left-hand term corresponds to the execution of a trace where the renaming from block $e$ to $b$ occurs on $\pi'_V = [v_1, \ldots, v_k, e]$, after the first access to $e$ in $\pi_V$. The right-hand term simply exhibits the first miss on the execution of the renamed trace $[\pi_V(e \to b), \pi_E]$ from $C_e$ as per (21):

$$\mathcal{H} + \mathcal{D}([[v_1, \ldots, v_k, e](e \to b), \pi_E], C_e) \leq \mathcal{D}([\pi_V(e \to b), \pi_E], C_e) \tag{162}$$

From the induction hypothesis we have:

$$\mathcal{D}([[v_1, \ldots, v_k, e], \pi_E], C_e) \leq \mathcal{D}([[v_1, \ldots, v_k, e](e \to b), \pi_E], C_e) \tag{163}$$

By inserting a hit latency on both sides this equation becomes:

$$\mathcal{H} + \mathcal{D}([[v_1, \ldots, v_k, e], \pi_E], C_e) \leq \mathcal{H} + \mathcal{D}([[v_1, \ldots, v_k, e](e \to b), \pi_E], C_e) \tag{164}$$

The first access in $\pi_V = [e, v_1, \ldots, v_k, e]$ from $C_e$ is a cache hit and leaves the cache state unchanged. The term on the left hand side can be expressed as:

$$\mathcal{D}([\pi_V, \pi_E], C_e) \leq \mathcal{H} + \mathcal{D}([[v_1, \ldots, v_k, e](e \to b), \pi_E], C_e) \tag{165}$$

Hence, $\mathcal{H} + \mathcal{D}([[v_1, \ldots, v_k, e](e \to b), \pi_E], C_e)$ is an intermediate bound between the execution time distributions of $\mathcal{D}([\pi_V, \pi_E], C_e)$ and $\mathcal{D}([\pi_V(e \to b), \pi_E, C_e)$. From (165) and (162), we have:

$$\begin{aligned} \mathcal{D}([\pi_V, \pi_E], C_e) &\leq \mathcal{H} + \mathcal{D}([[v_1, \ldots, v_k, e](e \to b), \pi_E], C_e) \\ &\leq \mathcal{D}([\pi_V(e \to b), \pi_E], C_e) \end{aligned} \tag{166}$$

Each possible input cache state $s'$ to the renamed segment has an as likely match $s$ in the original trace such that the execution time distribution of the renamed segment from $s'$ is no lower than that of the original from $s$. □

## References


Al-Zoubi H, Milenkovic A, Milenkovic M (2004) Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In: Proceedings of the 42nd annual Southeast regional conference. ACM, New York, pp 267–272

Alt M, Ferdinand C, Martin F, Wilhelm R (1996) Cache behavior prediction by abstract interpretation. In: Science of computer programming. Springer, Heidelberg, pp 52–66

Altmeyer S, Davis RI (2014) On the correctness, optimality and precision of static probabilistic timing analysis. In: 17th Conference on Design, Automation and Test in Europe (DATE)

Altmeyer S, Cucu-Grosjean L, Davis RI (2015) Static probabilistic timing analysis for real-time systems using random replacement caches. Real Time Syst 51:77–123

Atanassov P, Puschner P (2001) Impact of DRAM refresh on the execution time of real-time tasks. In: Proceedings of IEEE international workshop on application of reliable computing and communication, pp 29–34

Ballabriga C, Cassé H (2008) Improving the WCET computation time by IPET using control flow graph partitioning. In: 8th International workshop on worst-case execution time analysis (WCET)

Bernat G, Burns A, Newby M (2005) Probabilistic timing analysis: an approach using copulas. J Embed Comput 1(2):179–184

Bernat G, Colin A, Petters S (2002) WCET analysis of probabilistic hard real-time systems. In: 23rd IEEE real-time systems symposium (RTSS), pp 279–288

Bernat G, Colin A, Petters S (2003) pWCET: a tool for probabilistic worst-case execution time analysis of real-time systems. Tech. Report YCS-353-2003, Department of Computer Science, The University of York

Bhat B, Mueller F (2011) Making DRAM refresh predictable. Real Time Syst 47:430–453

Bourgade R, Ballabriga C, Cassé H, Rochange C, Sainrat P (2008) Accurate analysis of memory latencies for WCET estimation. In: 16th Conference on real-time and network systems (RTNS)

Burns A, Edgar S (2000) Predicting computation time for advanced processor architectures. In: Proceedings of the 12th Euromicro conference on real-time systems (Euromicro-RTS'00)

Cazorla F, Quiñones E, Vardanega T, Cucu L, Triquet B, Bernat G, Berger E, Abella J, Wartel F, Houston M, Santinelli L, Kosmidis L, Lo C, Maxim D (2013) Proartis: probabilistically analysable real-time systems. ACM Trans Embed Comput Syst 1(2s):1–26

Chiou D, Chiouy D, Rudolph L, Rudolphy L, Devadas S, Devadasy S, Ang BS, Angz BS (2000) Dynamic cache partitioning via columnization. In: Proceedings of design automation conference

Colin A, Puaut I (2001) A modular and retargetable framework for tree-based WCET analysis. In: 13th Euromicro conference on real-time systems (ECRTS), pp 37–44

Cortex-R4 and Cortex-R4F Technical Reference Manual (2010) http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.cortexr/index.html

Cucu-Grosjean L (2013) Independence—a misunderstood property of and for probabilistic real-time systems. In: Alan Burns 60th anniversary, York

Cucu-Grosjean L, Santinelli L, Houston M, Lo C, Vardanega T, Kosmidis L, Abella J, Mezzetti E, Quiones E, Cazorla FJ (2012) Measurement-based probabilistic timing analysis for multi-path programs. In: 24th Euromicro conference on real-time systems (ECRTS), pp 91–101

David L, Puaut I (2004) Static determination of probabilistic execution times. In: 16th Euromicro conference on real-time systems (ECRTS), pp 223–230, June 2004

Davis RI, Santinelli L, Altmeyer S, Maiza C, Cucu-Grosjean L (2013) Analysis of probabilistic cache related pre-emption delays. In: 25th Euromicro conference on real-time systems (ECRTS)

de Dinechin BD, van Amstel D, Poulhiès M, Lager G (2014) Time-critical computing on a single-chip massively parallel processor. In: Conference on Design, Automation & Test in Europe (DATE)

Edgar S, Burns A (2001) Statistical analysis of WCET for scheduling. In: 22nd IEEE real-time systems symposium (RTSS '01)

Griffin D, Burns A (2010) Realism in statistical analysis of worst case execution times. In: 10th International workshop on worst-case execution time analysis (WCET'10), July 2010

Griffin D, Lesage B, Burns A, Davis R (2014a) Lossy compression for static probabilistic timing analysis of random replacement caches. In: 22st International conference on real-time networks and systems (RTNS '14)

Griffin D, Lesage B, Burns A, Davis RI (2014b) Lossy compression for worst-case execution time analysis of PLRU caches. In: Proceedings of the 22nd international conference on real-time networks and systems (RTNS '14)
Grund D, Reineke J (2010) Precise and efficient FIFO-replacement analysis based on static phase detection. In: the 22nd Euromicro conference on real-time systems (ECRTS '10), July 2010
Grund D, Reineke J (2010) Toward precise PLRU cache analysis. In: 10th International workshop on worst-case execution time analysis (WCET'10), pp 28–39, July 2010
Gustafsson J, Betts A, Ermedahl A, Lisper B (2010) The Mälardalen WCET benchmarks—past, present and future. In: Proceedings of the 10th international workshop on worst-case execution time analysis (WCET), pp 137–147
Hahn S, Grund D (2012) Relational cache analysis for static timing analysis. In: 2012 24th Euromicro conference on real-time systems, pp 102–111
Hahn S, Reineke J, Wilhelm R (2015) Towards compositionality in execution time analysis: definition and challenges. In: SIGBED Review, vol 12. ACM, New York, pp 28–36
Hennessy JL, Patterson DA (2011) Computer architecture: A quantitative approach, 5th edn. Morgan Kaufmann, Burlington
Holsti N, Lngbacka T, Saarinen S (2000) Using a worst-case execution time tool for real-time verification of the DEBIE software. In: Proceedings of the DASIA 2000 (data systems in aerospace) conference
Huynh BK, Ju L, Roychoudhury A (2011) Scope-aware data cache analysis for WCET estimation. In: 17th Real-time and embedded technology and applications symposium (RTAS)
Kosmidis L, Abella J, Quiñones E, Cazorla FJ (2013) A cache design for probabilistically analysable real-time systems. In: 16th conference on Design, Automation and Test in Europe (DATE), pp 513–518
Kosmidis L, Abella J, Wartel F, Quinones E, Colin A, Cazorla F (2014) PUB: path upper-bounding for measurement-based probabilistic timing analysis. In: 26th Euromicro conference on real-time systems (ECRTS)
Lesage B, Griffin D, Davis R, Altmeyer S (2013) On the application of static probabilistic timing analysis to memory hierarchies. In: Real-time scheduling open problems seminar (RTSOPS)
Lesage B, Griffin D, Altmeyer S, Davis R (2015a) Static probabilistic timing analysis for multi-path programs. In: Real-time systems symposium (RTSS)
Lesage B, Griffin D, Soboczenski F, Bate I, Davis RI (2015b) A framework for the evaluation of measurement-based timing analyses. In: 23rd International conference on real time and networks systems (RTNS)
Li YT, Malik S (1997) Performance analysis of embedded software using implicit path enumeration. Trans Comput Aided Des Integr Circuit Syst 16:1477–1487
Liang Y, Mitra T (2008) Cache modeling in probabilistic execution time analysis. In: Proceedings of the 45th annual design automation conference (DAC), pp 319–324
López J, Díaz J, Entrialgo J, García D (2008) Stochastic analysis of real-time systems underpreemptive priority-driven scheduling. Real Time Syst 40:180–207
Maxim D, Houston M, Santinelli L, Bernat G, Davis RI, Cucu-Grosjean L (2012) Re-sampling for statistical timing analysis of real-time systems. In: 20th International conference on real-time and network systems (RTNS), pp 111–120
MPC8641D Integrated Host Processor Family Reference Manual (2008) http://www.nxp.com/products/microcontrollers-and-processors/power-architecture-processors/integrated-host-processors/high-performance-dual-core-processor:MPC8641D?fpsp=1&tab=Documentation_Tab
Muchnick SS (1997) Advanced compiler design and implementation. Morgan Kaufmann, San Francisco
Nemer F, Cassé H, Sainrat P, Bahsoun J.P, Michiel M D (2006) PapaBench: a free real-time benchmark. In: 6th International workshop on worst-case execution time analysis (WCET'06), vol 4 of OpenAccess Series in Informatics (OASIcs)
Pasdeloup B (2014) Static probabilistic timing analysis of worst-case execution time for random replacement caches. Tech. Report, INRIA, Rennes
Peleska J, Löding H (2008) Static analysis by abstract interpretation. University of Bremen, Centre of Information Technology, Bremen
Puschner P, Koza C (1989) Calculating the maximum, execution time of real-time programs. Real Time Syst 1(2):159–176
Quinones E, Berger ED, Bernat G, Cazorla FJ (2009) Using randomized caches in probabilistic real-time systems. In: 21st Euromicro conference on real-time systems (ECRTS), pp 129–138
Reineke J (2014) Randomized caches considered harmful in hard real-time systems. LITES 1(1):03:1–03:13

Reineke J, Wachter B, Thesing S, Wilhelm R, Polian I, Eisinger J, Becker B (2006) A definition and classification of timing anomalies. In: 6th International workshop on worst-case execution time (WCET) analysis

Spreitzer R, Plos T (2013) Cache-access pattern attack on disaligned AES T-tables. In: Proceedings of the 4th international conference on constructive side-channel analysis and secure design (COSADE'13), pp 200–214

Theiling H, Ferdinand C, Wilhelm R (1999) Fast and precise WCET prediction by separated cache and path analyses. Real Time Syst 18:157–179

Wang Z, Lee RB (2007) New cache designs for thwarting software cache-based side channel attacks. In: Proceedings of the 34th annual international symposium on computer architecture (ISCA '07). ACM, New York, pp 494–505

Wang Z, Lee RB (2008) A novel cache architecture with enhanced performance and security. In: Proceedings of the 41st annual IEEE/ACM international symposium on microarchitecture (MICRO 41), pp 83–93

Wegener S (2012) Computing same block relations for relational cache analysis. In: 12th International workshop on worst-case execution time analysis, pp 25–37

Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, Mueller F, Puaut I, Puschner P, Staschulat J, Stenström P (2008) The worst-case execution-time problem: overview of methods and survey of tools. ACM Trans Embed Comput Syst 7(3):1–53

**Benjamin Lesage** is a Research Associate in the Real-Time Systems Research Group at the University of York, UK. Benjamin received his PhD in Computer Science in 2013 from the University of Rennes, France. He has since been at the University of York as a Research Associate. He is currently working in the context of a Knowledge Transfer Partnership, in collaboration with industrial partners, to put into practice his knowledge of real-time systems' timing analyses.

**David Griffin** is currently a member of the Real Time Systems Group at the University of York, UK. His research has primarily been in the application of non-standard techniques to Real-time problems, utilising techniques from various other fields such as lossy compression, statistics and machine learning.

**Sebastian Altmeyer** is Assistant Professor (Universitair Docent) at the University of Amsterdam. He has received his PhD in Computer Science in 2012 from Saarland University, Germany with a thesis on the analysis of preemptively scheduled hard real-time systems. From 2013 to 2015 he has been a postdoctoral researcher at the University of Amsterdam, and from 2015 to 2016 at the University of Luxembourg. In 2015, he has received an NWO Veni grant on the timing verification of real-time multicore systems, and he is program chair of the Euromicro Conference on Real-Time Systems (ECRTS) 2018. His research targets various aspects of the design, analysis and verification of hard real-time systems, with a particular interest in timing verification and multicore architectures.

**Liliana Cucu-Grosjean** Photograph and Biography not available.

**Robert I. Davis** is a Senior Research Fellow in the Real-Time Systems Research Group at the University of York, UK, and an INRIA International Chair with INRIA, Paris, France. Robert received his PhD in Computer Science from the University of York in 1995. Since then he has founded three start-up companies, all of which have succeeded in transferring real-time systems research into commercial products. Robert's research interests include the following aspects of real-time systems: scheduling algorithms and analysis for single processor, multiprocessor and networked systems; analysis of cache related pre-emption delays, mixed criticality systems, and probabilistic hard real-time systems.