On the Correctness, Optimality and Precision of Static Probabilistic Timing Analysis

Sebastian Altmeyer
University of Amsterdam, Netherlands
altmeyer@uva.nl

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

***Abstract*—In this paper, we investigate Static Probabilistic Timing Analysis (SPTA) for single processor systems that use a cache with an evict-on-miss random replacement policy. We show that previously published formulae for the probability of a cache hit can produce results that are optimistic and unsound when used to compute probabilistic Worst-Case Execution Time (pWCET) distributions.**

We investigate the correctness, optimality, and precision of different approaches to SPTA. We prove that one of the previously published formulae for the probability of a cache hit is optimal with respect to the limited information that it uses. We improve upon this formulation by using extra information about cache contention. To investigate the precision of various approaches to SPTA, we introduce a simple exhaustive method that computes a precise pWCET distribution, albeit at the cost of exponential complexity. Further, we integrate this precise approach, applied to small numbers of frequently accessed memory blocks, with imprecise analysis of other memory blocks, to form a combined approach that improves precision, without significantly increasing its complexity. The performance of the various approaches are compared on benchmark programs.

Technical Report

This technical report supplements [1] published in DATE 2014. The technical report:

- Corrects an omission in the formula for cache contention given in section III, with a proof of correctness in Appendix A 1.
- Provides an extended evaluation covering additional benchmarks in Appendix A 2 and larger traces in Appendix A 3.

I. Introduction

Real-time systems such as those deployed in space, aerospace, automotive and railway applications require guarantees that the probability of the system failing to meet its timing constraints is below an acceptable threshold (e.g. a failure rate of less than 10^{-9} per hour). Advances in hardware technology and the large gap between processor and memory speeds, bridged by the use of cache, make it difficult to provide such guarantees without significant over-provision of hardware resources. The use of deterministic cache replacement policies means that pathological worst-case behaviours need to be accounted for, even when in practice they may have a vanishingly small probability of actually occurring. Further, the quality of deterministic worst-case execution time estimates for such systems can be highly sensitive to missing information, making them overly pessimistic [3]. The use of cache with random replacement policies can negate the effects of these pathological worst-case behaviours while still achieving efficient average-case performance, hence providing a way of increasing guaranteed performance in hard real-time systems.

978-3-9815370-2-4/DATE14/©2014 EDAA

The timing behaviour of programs running on a processor with a random cache 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 exceedence function (1 - cumulative distribution function (CDF)). This exceedence 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 not exceed that budget on any single run. (See [7] for examples of pWCET distributions, and [4] for a detailed discussion of what is meant by a pWCET distribution and the important difference between that and a probabilistic Execution Time (pET) distribution).

SPTA comprises two main steps [7]: First, a probability function is required that can be used to compute an estimate of the probability of a cache hit for each memory access. This probability function is *valid* if it provides a lower bound on the probability of a cache hit. Typical probability functions used in SPTA are a function of the cache associativity and the *reuse distance*, defined as the number of intervening memory accesses that could cause an eviction, since the memory block was last accessed. The probability function is used to obtain a pWCET distribution for each instruction. Second, SPTA computes the pWCET distribution for a sequence of instructions by convolving the distributions obtained for individual instructions. For convolution to give correct results, the pWCET distributions obtained for the different instructions must be *independent*. By *independent*, we mean that the estimate of the probability of a cache hit for a given memory access remains a valid lower bound irrespective of the behaviour of other memory accesses. We note that the precise probability of a cache hit for a given memory access is rarely independent, it typically depends strongly on the history of previous accesses (i.e. whether or not they were cache hits). Thus care needs to be taken in the derivation of a suitable probability function to ensure that independence is obtained.

SPTA has been developed for single processor systems assuming evict-on-miss [10], [9], and evict-on-access [5], [3] random replacement policies. This initial work assumed single path programs and no pre-emption. Subsequently, Davis et al. [7] provided analysis for both evict-on-miss and evict-on-access policies for single and multi-path programs, along with a method of accounting for cache related pre-emption delays. As the evict-on-miss policy dominates evict-on-access we focus on the former in this paper.

Despite the intensive research in this area over the past few years, it remains an open problem [6] how to accurately and

efficiently compute the pWCET distributions for individual instructions and sequences of them. In particular, prior approaches gave little information about the correctness and precision of the pWCET distributions obtained.

In this paper, we re-visit the probability functions that form the fundamental building blocks of SPTA. We show that convolution of the probability functions given in [9] and [10] is unsound (optimistic), as these probability functions do not provide lower bounds on the probability of a cache hit that are *independent* of the behaviour of previous memory accesses. By contrast, we show that the probability function derived in [7] provides a valid lower bound that is independent of the behaviour of previous memory accesses, enabling the calculation of sound pWCET distributions via convolution. We prove that this probability function is *optimal* with respect to the limited information (reuse distances and the cache associativity) that it employs, in the sense that no further improvement is possible without considering additional information.

As well as correctness and optimality, we also investigate the precision of SPTA. Despite claims to the contrary in the conclusions of [5], SPTA does not provide a precise pWCET distribution for a sequence of instructions when based on the convolution of simple pWCET distributions for each instruction. Instead, precise analysis requires that the probabilities of all possible sequences of cache hits and cache misses are considered, leading to exponential complexity. Previous work [3, 5, 9, 10] provides little indication of the precision of existing SPTA techniques, while [7] provides some comparisons with simulation.

In this paper, we improve the precision of SPTA in two ways. First, we refine the probability function given in [7] using the concept of *cache contention*. Second, we describe a simple approach that exhaustively enumerates all cache states that may occur for a given sequence of memory accesses. This provides precise analysis at the cost of complexity that is exponential in the number of pairwise distinct memory blocks. Nevertheless, this approach enables us to quantify the precision of various approaches to SPTA for small programs. We also introduce a combined approach which integrates precise analysis of the most important memory accesses (those made most frequently), with imprecise analysis of the remaining memory accesses, using a simple probability function. We show that this combined technique is effective in improving the accuracy of SPTA while avoiding the exponential increase in complexity that exhaustive analysis brings.

A. Random Cache Replacement

A cache with the evict-on-miss random replacement policy operates as follows: whenever a memory block is requested and is not found in the cache, then a randomly chosen cache line is evicted and the requested block is loaded into the evicted location. We assume an N-way associative cache, and so the probability of any cache line being evicted on a miss is $1/N$.

Prior work considered mostly instruction cache only and no data cache; however, this restriction is unnecessary for the theoretical foundations of our analysis. We therefore define traces as sequences of memory blocks directly (independent of the content of the memory blocks) instead of sequences of instructions as was done in [7]. Further, the restriction to a fully-associative cache can be easily lifted, as a set-associative cache with s cache sets can be analysed as s parallel and independent fully-associative caches.

B. Traces and Reuse Distance

A trace T of size n is an ordered sequence of n memory blocks $[e_1, \ldots, e_n]$. The set of all traces is denoted by $\mathbb{T}$, and $\mathbb{E}$ denotes the set of all elements. The reuse distance $\mathrm{rd}(e)$ of an element e is the maximum number of evictions since the last access to the same element, with reuse distance ∞ in the case that there is no prior access to that memory block.

$$rd\colon \mathbb{E} \times \mathbb{T} \to \mathbb{N} \cup \{\infty\}$$
$$rd\,(e_l, [e_1, \ldots e_{l-1}]) = \begin{cases} l-j-1 & \text{if } \exists e_j\colon e_j = e_l \wedge \forall_{j<i<l}\colon e_l \neq e_i \\ \infty & \text{otherwise} \end{cases} \tag{1}$$

We typically represent the reuse distance k using a superscript and omit all infinite reuse distances. For example,

$$a, b, a^1, c, d, b^3, c^2, f, a^5, c^5$$

We denote the event of a cache hit at memory block e_i as e_i^{hit} and $P(e_i^{hit})$ the corresponding probability, with e_i^{miss} and $P(e_i^{miss})$ being the equivalent for a cache miss. Further, we use $\hat{P}$ to denote the approximations to distinguish them from the actual values.

C. Review of prior Approaches

In this section, we present the different approaches that have been proposed to compute the probability of cache hits and misses and thus the pWCET distribution.

Zhou [11] proposed using the reuse-distance to compute the probability $P(e^{hit})$ of a cache hit at access e with reuse distance k:

$$\hat{P}^Z(k) = \left(\frac{N-1}{N}\right)^k \tag{2}$$

where N is the associativity of the cache. The rationale behind (2) is that the second access to e can only be a hit, if all intermediate cache misses evict cache lines other than the one element e occupies. Equation (2) is not precise, but a lower bound on the *individual* probability of a cache hit. Remember that the reuse distance was defined as the *maximum* number of evictions, and not the actual number. Therefore $\hat{P}^Z(\mathrm{rd}(e)) < P(e^{hit})$ holds for some access sequences.

In 2009, Quinones et al. [10] proposed to derive the pWCET distribution of a single path program via convolution of the pWCET distributions of individual accesses obtained from (2). However, (2) is only valid if considered in isolation and the convolution for independent events cannot be used due to a dependency stemming from the finite size of the cache (see [7]). To correct (2), Davis et al. [7] proposed an independent lower bound on the probability of a cache hit:

$$\hat{P}^D(k) = \begin{cases} \left(\frac{N-1}{N}\right)^k & N > k \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

The drawback of (3) is that all accesses with reuse distance higher than the associativity are considered to be cache misses.

In 2013, Kosmidis et al. [9] proposed the following formula

$$\hat{P}^K(e^{hit}) = \left(\frac{N-1}{N}\right)^{\sum P\left(e_j^{\mathrm{miss}}\right)} \tag{4}$$

where the summation in the exponent is over the probabilities of misses of the intervening memory accesses. (We note that a similar formula was also given by Zhou [11] as an approximation). Equation (4) may over-estimate the actual probability of a cache hit as noted in [6], and thus lead to a pWCET distribution that is optimistic.

Using one of the above estimates of the probability of a cache hit, the Probability Mass Function (PMF) $\mathcal{I}_i$ of element

e_i is defined as follows:

$$\mathcal{I}_i = \begin{pmatrix} \text{hit-delay} & \text{miss-delay} \\ P(e^{hit}) & P(e^{miss}) \end{pmatrix} \tag{5}$$

with $P(e^{miss}) = 1 - P(e^{hit})$ and *hit-delay* (*miss-delay*) denoting the execution time for a cache hit (cache miss). The $pWCET$ distribution is then derived by computing the convolution $\otimes$ of the probability mass function of each memory access e_i:

$$pWCET = \bigotimes \mathcal{I}_i \tag{6}$$

Under the assumption of constant hit- and miss-delays, computing the distribution of cache-hits and cache-misses is sufficient to derive the pWCET distribution. We will therefore concentrate only on the former, the latter can be obtained by multiplying the constant delays by the number of hits and misses.

II. Correctness Conditions and Optimality

As stated in [7], the probability that a single access is a cache hit/miss is not independent of prior events. This means that in general, the convolution for independent events cannot be soundly applied, as it is only valid for independent events. What can be done instead, however, is to provide a lower bound approximation $\hat{P}$ to the actual probability of a cache hit for which we can soundly apply the basic convolution for independent events. Sound in this context means that for any sequence of cache accesses $[e_1, \ldots, e_n]$, the approximation $\hat{P}$ (i) does not over-estimate the probability of a cache hit, and (ii) the value obtained from convolution of the approximated probabilities for any subset of a trace T describing the probability that all elements in the subset are a hit, is at most the precise probability of such an event occurring:

(i) $\forall e \in [e_1, \ldots, e_n]\colon P(e^{hit}) \geq \hat{P}(e^{hit})$,

(ii) $\forall E \subseteq [e_1, \ldots, e_n]\colon P\left(\bigwedge_{e \in E} e^{hit}\right) \geq \prod_{e \in E} \hat{P}(e^{hit})$.

A. Counterexamples to Equation (2) and Equation (4)

Of the different approaches presented in Section I, only (3) provides a valid and sound lower-bound. Equation (2) does not fulfil (ii) and (4) fulfils neither (i) nor (ii) as shown below.

To show that (4) over-estimates (i) the probability of an individual cache hit, we use the access sequence

$$a, b, c, d, a^3, b^3$$

and a cache with associativity 2.

Equation (4) computes the probability of a cache hit for the second access to a of $(1/2)^3 = 0.125$, which is correct and precise. For the second access to b, however, (4) results in $(1/2)^{2+0.875} \geq 0.1363$ which is optimistic as the correct probability of a cache hit is $(1/2)^3 = 0.125$ in this case. (This can be seen by enumerating the possible cache states that could exist after the second access to a; out of 16 possibilities each with probability 0.0625, only two contain b).

To show that (ii) does not hold using (4) we use an example derived from that given in [6], we assume the access sequence

$$a, b, a^1, b^1$$

and a cache with associativity of 100. Further, we assume that the latency of a cache hit is 1 and the latency of a cache miss is 10. The first two accesses are certain misses, so using (4), the probability distributions for the first three instructions are as follows:

$$\begin{pmatrix} 1 & 10 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 10 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 10 \\ 0.99 & 0.01 \end{pmatrix} \tag{7}$$

According to (4), the probability of the 4th access being a cache hit is then: $\hat{P}^K(e^{hit}) = 0.99^{0.01} = 0.9998995$. (Note this value is rounded down slightly, which is a safe assumption). So the overall $pWCET$ is

$$\begin{pmatrix} 10 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 10 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 10 \\ 0.99 & 0.01 \end{pmatrix} \otimes \begin{pmatrix} 1 & 10 \\ 0.9998995 & 0.0001005 \end{pmatrix} = \begin{pmatrix} 22 & 31 & 40 \\ 0.989900505 & 0.01009849 & 1.005e^{-6} \end{pmatrix} \tag{8}$$

However, the correct $pWCET$ is

$$\begin{pmatrix} 22 & 31 & 40 \\ 0.99 & 0.0099 & 0.0001 \end{pmatrix} \tag{9}$$

This is easily seen by considering the scenarios that result in a total of two and four cache misses respectively. (Recall that the first accesses to a and b are certain to be cache misses). If the first access to b does not evict a (probability 0.99), then the second access to a can only be a cache hit, which in itself does not evict b, and so in this scenario, which has a probability of occurring of 0.99, there are two cache misses in total. Alternatively, if the first access to b evicts a (probability 0.01), then the second access to a is *certain* to be a cache miss, which in turn has a probability of 0.01 of evicting b and so making the second access to b a cache miss. Hence the only scenario with four cache misses in total has a probability of occurring of 0.0001.

In this example, using (4) results in a pWCET distribution that under-estimates the probability of obtaining four cache misses and hence an execution time of 40, by two orders of magnitude. This is a highly optimistic and unsound result.

To show that (2) also contradicts constraint (ii), we assume an associativity of 4 and the access sequence:

$$a, b, c, d, e, a^4, b^4, c^4, d^4, e^4$$

By construction, all probabilities of a cache hit $\hat{P}^Z(e^{hit})$ for the last five accesses are non-zero. Hence, the combined probability of five hits is also non-zero, which contradicts the fact that at most 4 elements can be stored simultaneously in the cache.

B. Optimality of Equation (3)

We can derive an estimate of the probability of a cache hit that is more precise than (3) as we can simply enumerate all possible cache states and the associated probabilities; however, this solution is computationally intractable, as we will explain in Section IV. If we aim at an estimate using only the reuse distances and on which we can apply the convolution for independent events, then (3) is optimal in the sense that there is no function of k and N that is valid and returns any larger value.

Proof: We assume that $\hat{P}'(k)$ is a probability function such that $\hat{P}'(k)$ is more precise than $\hat{P}^D(k)$. Hence:

$$\exists k\colon \hat{P}'(k) = \hat{P}^D(k) + \epsilon \tag{10}$$

for some $\epsilon > 0$. We assume that the only input to $\hat{P}^D$ and $\hat{P}'$ is the reuse distance k which must be valid for any sequence of accesses $[e_1, \ldots, e_n]$, and the associativity N. We make a case distinction on k:

Case $k < N$: Assume the following ordered sequence with accesses to k pairwise distinct elements

$$[e_x, e_1, e_2, e_3, \ldots e_{k-1}, e_k, e_x]$$

and an initially empty cache. The reuse distance of the second access to e_x is k. Each access to any

of the other elements results in a cache miss, the probability of a cache hit $P(e_x^{hit})$ is exactly $\hat{P}^D(k)$ and the assumption that $\epsilon > 0$ contradicts (i).

Case $k \geq N$: Assume the access pattern

$$[e_1, e_2, e_3, \ldots e_{k-1}, e_k, e_1, e_2, e_3, \ldots e_{k-1}, e_k]$$

for each second access to e_i, the reuse distance is k. Since the cache can store at most N elements and $k > N$, we know that the probability of k hits is 0, i.e, $P\left(\bigwedge_{e_i} e_i^{hit}\right) = 0$. However, since $\epsilon > 0$ and $\hat{P}^D(k) \geq 0$, we can conclude that $\prod_{e \in E} \hat{P}'(e^{hit}) > 0$, which contradicts (ii).

Hence, we can construct for any k and any N, an access sequence where $\hat{P}^D$ is optimal in the sense that it provides the largest valid value of any function relying only on the reuse distance and the associativity. ∎

III. Probability of a cache hit using Cache Contention

This section corrects an omission in the formula for cache contention given in the published paper [1]. Proof of correctness for the revised formula is given in Appendix A 1.

Equation (3) provides a tight lower bound on the probability of a cache hit, but it is imprecise even for simple access sequences. If we consider for instance a random cache with associativity 4 and the following access sequence,

$$a, b, c, d, f, a^4, b^4, c^4, d^4, f^4$$

all accesses are considered cache misses. The reason for this is that for each of the last five accesses, the probability of a cache hit is set to 0 to ensure correctness with respect to constraint (ii), i.e, that the probability of the last five access all being hits is zero. However, this can also be ensured by considering the probability of a cache hit for the preceding accesses. To this end, we define the concept of the *cache contention con* of a memory block e_l which denotes the number of memory accesses within the reuse distance of e_l that potentially contend with e_l for space in the cache. The cache contention models each of the accesses within the reuse distance of e_l that have been assigned non-zero probability of being a hit as requiring its own separate location in the cache. Further, it is also assumed that the first element e_r within the reuse distance of e_l (i.e. where $r = l - rd(e_l, [e_1, \ldots, l-1])$ also requires a separate location in the cache regardless of its assigned probability.

$$con\colon \mathbb{E} \times \mathbb{T} \to \mathbb{N}$$

$$con\left(e_l, [e_1, e_2, \ldots, e_{l-1}]\right) = \begin{cases} \infty & rd(e_l, [e_1, \ldots, l-1]) = \infty \\ \left|conS\right| & \text{otherwise} \end{cases} \tag{11}$$

with

$$conS = \left\{e_i \in [e_1, \ldots, e_{l-1}] \;\middle|\; \left(l - rd(e_l, [e_1, \ldots, l-1]) < i \wedge \hat{P}(e_i^{hit}) \neq 0\right)\right\} \cup \left\{e_r \in [e_1, \ldots, e_{l-1}] \;\middle|\; (l - rd(e_l, [e_1, \ldots, l-1]) = r\right\}$$

We only need to set the probability of a cache hit for an access e to zero when the cache contention is greater than or equal to the associativity N.

$$\hat{P}^N(e^{hit}) = \begin{cases} 0 & con(e_l, T) \geq N \\ \left(\frac{N-1}{N}\right)^k & \text{otherwise} \end{cases} \tag{12}$$

The cache contention *con* and the probability $\hat{P}^N$ are mutually dependent; but the cache contention of an element e_l depends only on the probability of a cache hit for the preceding elements, which enables *con* and $\hat{P}^N$ to be computed efficiently.

Table I presents the probability $\hat{P}^N$ for the elements of the example sequence. In contrast to (3), only one of the five elements with finite reuse distance is assumed to have zero probability of being a cache hit.

	a,	b,	c,	d,	f,	a,	b,	c,	d,	f
rd	∞	∞	∞	∞	∞	4	4	4	4	4
con	∞	∞	∞	∞	∞	1	2	3	4	3
$\hat{P}^D$	0	0	0	0	0	0	0	0	0	0
$\hat{P}^N$	0	0	0	0	0	$(\frac{3}{4})^4$	$(\frac{3}{4})^4$	$(\frac{3}{4})^4$	0	$(\frac{3}{4})^4$

TABLE I. Probabilities $\hat{P}^N$ and $\hat{P}^D$ for the access sequence $a, b, c, d, f, a, b, c, d, f$, with reuse distances ($rd$) and cache contentions (con).

IV. Exhaustive State-Enumeration

We now describe a simple analysis to compute the exact probability distribution of cache hits. This analysis is orthogonal to the approaches presented in previous sections. Here, we exhaustively enumerate all cache states that may occur during the execution of a given trace.

The domain of the analysis is a set of cache states defined as follows: A cache state CS is a triple (E, P, D) consisting of a set of memory blocks $E \subseteq \mathbb{E}$, a probability $P \in \mathbb{R}$ and a distribution of cache misses $D\colon (\mathbb{N} \to \mathbb{R})$. A cache state $CS = (E, P, D)$ has the following meaning: the cache contains exactly the elements E with probability P, and D denotes the distribution of cache misses when the cache is in this state. The set of all cache states is denoted by $\mathbb{CS}$. Note that we model a distribution by the function $\mathbb{N} \to \mathbb{R}$ which assigns each possible number of cache hits a probability. We start with an initially empty cache, i.e. $CS_{init} = (\emptyset, 1, D)$ with

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

Hence, our initial state space is the set containing the initially empty cache: $\{CS_{init}\}$.

The update function u describes the cache update when accessing element e for a single cache state as follows:

$$u\colon \mathbb{CS} \times \mathbb{E} \to 2^{\mathbb{CS}}$$

$$u\left((E, P, R), e\right) = \begin{cases} \{(E, P, D)\} & \text{if } e \in E \\ miss((E, P, D), e) & \text{otherwise} \end{cases} \tag{13}$$

If the accessed element e is a cache hit, then the cache state remains unchanged, and the output is the set containing the input cache state. However, if the accessed element e is a cache miss, then the update function generates a set of possible cache states as follows:

$$miss((E, P, D), e) = \{(E \setminus e' \cup \{e\}, P \cdot 1/N, D') | e' \in E\} \cup \{(E \cup \{e\}, P \cdot (N - |E|)/N, D') | \text{ if } |E| < N\} \tag{14}$$

with $D'(0) := 0$ and $\forall x > 0\colon D'(x) = D(x-1)$.

Each element e' from the set E may be evicted from the cache with probability $1/N$ and the element e, which is now cached, is added to E. In the case that the set E is smaller than the associativity of the cache, an empty cache line or a cache line with unknown content will be evicted with probability $(N - |E|)/N$. In either case, the miss-distribution will be shifted by one to account for the additional cache miss.

In order to reduce the state space, we merge two cache states if they contain exactly the same memory blocks. We thus define

the join operation for cache states as follows:

$$\sqcup\colon \mathbb{CS} \times \mathbb{CS} \to 2^{\mathbb{CS}}$$
$$(E_1, P_1, D_1) \sqcup (E_2, P_2, D_2) =$$
$$\begin{cases} \left\{(E_1, P_1 + P_2, (\frac{P_1}{P_1+P_2} \cdot D_1) \oplus (\frac{P_2}{P_1+P_2} \cdot D_2))\right\} & \text{if } E_1 = E_2 \\ \{(E_1, P_1, D_1), (E_2, P_2, D_2)\} & \text{otherwise} \end{cases} \tag{15}$$

where $\oplus$ denotes the summation of two distributions (i.e. $D_1 \oplus D_2 := \lambda x.D_1(x) + D_2(x)$) and $p \cdot D$ denotes the multiplication of each element in D by p (i.e. $p \cdot D := \lambda x.p \cdot D(x)$). This step is necessary to weight each distribution by its probability.

We can lift the function u from single cache states to a set of cache states as follows:

$$U\colon 2^{\mathbb{CS}} \times \mathbb{E} \to 2^{\mathbb{CS}}$$
$$U(S, e) = \bigsqcup \{u(CS, e) | CS \in S\} \tag{16}$$

The set of cache states S_{res} generated by a trace $T = [e_1, \ldots, e_n]$ executed on the initial cache state CS_{init} is thus given by the composition of U:

$$S_{res} := U(\ldots(U(U(CS_{init}, e_1), e_2), \ldots, e_n) \tag{17}$$

The final distribution of all cache states in S_{res} is then given by the summation of all individual distributions of each cache state weighted by their probabilities:

$$D_{res} = \bigoplus \{D \cdot P | (E, P, D) \in S_{res}\} \tag{18}$$

See Figure 1 for an example of the exhaustive enumeration of all cache states, for a cache with associativity 4. Here, we assume an initially empty cache. The access to block a leads with probability 1 to the next cache state (where only a is cached). The next access to b evicts memory block a with probability $1/4$ or is stored in a different cache line to a with probability $3/4$, and so on.

Unfortunately, a complete enumeration of all possible cache states is computationally intractable. Due to the behaviour of the random replacement policy, each element that was cached once, may still be cached. Thus the number of different cache states grows exponentially.

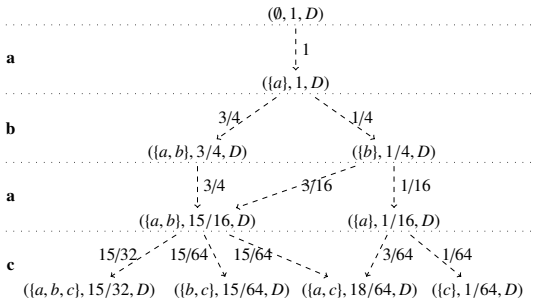

Fig. 1. The first four steps of exhaustive enumeration of all cache states for the access sequence $a, b, a, c, d, b, c, f, a, c$ The dotted arrows show the evolution of the different cache states, annotated with the corresponding probability.

V. Combined Approach

So far, we presented two approaches, one precise but computationally intractable, the other imprecise yet efficient. In this section, we show how these approaches can be combined to form a new approach with scalable precision. The idea is to use the precise approach for a small subset of *relevant* memory blocks, while using the imprecise approach for the remaining blocks. So, instead of enumerating all possible cache states, we abstract the set of cache states and focus only on the m most important memory blocks, where m can be chosen to control both the precision and the runtime of the analysis. In this way, we effectively reduce the complexity of the precise component of the analysis for a trace with l distinct elements from 2^l to 2^m (typically with $m \ll l$). We use the number of occurrences of a memory block e within a trace T as a simple heuristic indicating relevance. We therefore order the memory blocks within a trace T by the number of occurrences and select the m blocks with the highest frequency. Let $R \subseteq \mathbb{E}$ be the set of these m blocks. For the access sequence

$$a, b, a, c, d, b, c, f, a, c$$

and $m = 2$, $R = \{a, c\}$. Thus, the state exploration conceptually computes a precise probability distribution for the sequence

$$a, \_, a, c, \_, \_, c, \_, a, c$$

while the imprecise calculation is used to compute the probability of cache hits for the sequence

$$\_, b, \_, \_, d, b, \_, f, \_, \_$$

We have to change the update function of the analysis (see (13)) such that only elements from the set R are represented explicitly, i.e. $\forall (E, P, D) \in \mathbb{CS}\colon E \subseteq R$. Each access to a memory block e which is not contained in the set R will be considered a cache miss; however, e will not be added to the set E (to ensure that $E \subseteq R$). Further, as we use (12) to compute the probability of a hit for access e, and include the distribution for e via convolution, we do not increase the miss counts of the cache states in respect of e, i.e., we do not update the miss-distributions D of a cache state (E, P, D).

$$u\colon \mathbb{CS} \times \mathbb{E} \to 2^{\mathbb{CS}}$$
$$u((E, P, R), e) = \begin{cases} \{(E, P, D)\} & \text{if } e \in E \\ miss((E, P, D), e) & \text{if } e \notin E \wedge e \in R \\ miss'((E, P, D)) & \text{if } e \notin R \end{cases} \tag{19}$$

with

$$miss'((E, P, D)) = \{(E \setminus e', P \cdot 1/N, D) | e' \in E\}$$
$$\cup \{(E, P \cdot (N - |E|)/N, D) | \text{ if } |E| < N\} \tag{20}$$

The function $miss'$ computes the resulting set of cache states in the case of a miss, without inserting the accessed element e as it is not an element of R. Figure 2 shows the reduced cache state exploration on the example sequence with $R = \{a, c\}$.

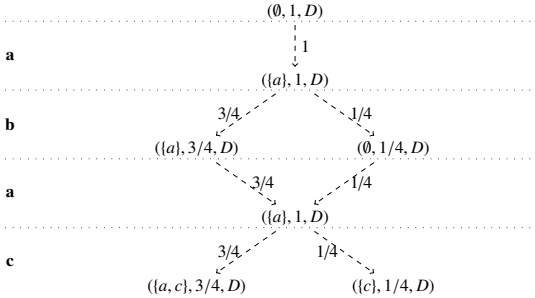

Fig. 2. The first four steps of the reduced cache state enumeration for the access sequence $a, b, a, c, d, b, c, f, a, c$ with $R = \{a, c\}$. The dotted arrows show the evolution of the different cache states, annotated with the corresponding probability. The access to memory block b is considered as cache miss, but this block is not added to the cache states, since $b \notin R$.

We also have to update the definition of cache contention (11) to consider all memory blocks of R as potentially

cached:

$$con : \mathbb{E} \times \mathbb{T} \to \mathbb{N}$$

$$con\,(e_l, [e_1, e_2, \ldots, e_{l-1}]) = \begin{cases} \infty & rd(e_l, [e_1, \ldots, l-1]) = \infty \\ \left|conS\right| + \left|R\right| & \text{otherwise} \end{cases} \quad (21)$$

with

$$conS = \left\{ e_i \in [e_1, \ldots, e_{l-1}] \;\middle|\; \left(l - rd(e_l, [e_1, \ldots, l-1]) < i \wedge \hat{P}(e_i^{hit}) \neq 0 \wedge e_i \notin R\right) \right\} \cup \left\{ e_r \in [e_1, \ldots, e_{l-1}] \;\middle|\; (l - rd(e_l, [e_1, \ldots, l-1]) = r \wedge e_r \notin R \right\}$$

Table II presents the probability $\hat{P}^N$ for the elements of the example sequence, assuming $N = 4$.

	_,	b,	_,	_,	d,	b,	_,	f,	_,	_
rd	_	∞	_	_	∞	4	_	∞	_	_
con	_	∞	_	_	∞	2	_	∞	_	_
$\hat{P}^N$	_	0	_	_	0	$(\frac{3}{4})^4$	_	0	_	_

TABLE II. PROBABILITIES $\hat{P}^N$ FOR THE ACCESS SEQUENCE $a, b, a, c, d, b, c, f, a, c$ AND $R = \{a, c\}$, WITH REUSE DISTANCES (rd) AND CACHE CONTENTIONS (con).

In the last step, we convolve the resulting distributions of both approaches to obtain the final distribution of cache misses.

VI. EVALUATION

In this section, we compare the precision of the various approaches presented. Due to the limited space, here we only give results for two benchmarks from the Mälardalen Benchmark Suite [8] (binary search and insertion sort). An extended evaluation covering 6 additional benchmarks, including 4 with much longer traces, and an assessment of the tractability (runtime) of the combined approach can be found in the appendix.

The selected benchmarks are simple but allow us to clearly focus on the code characteristics that impact the precision of the results: (i) the number of distinct memory blocks and (ii) the overall number of memory accesses. We assumed a fully-associative instruction-only cache with an associativity of 16 and a block-size of 8.

To derive an approximation to the actual performance of the random cache, and thus a baseline for our experiments, we performed 10^9 simulations of the cache behaviour for each benchmark (red line). The other lines on the graphs are the imprecise approach using only the reuse distance (3) (light blue line), the cache-contention approach (12) (black line), and the combined approach using 4, 8 and 12 relevant memory blocks (green, dark blue and pink lines respectively).

When the number of distinct memory blocks is smaller than or close to the cache associativity, then the cache-contention approach results in no improvement or only a slight improvement over the imprecise approach using only the reuse distance. This was the case with insertion sort (see Figure 3). Yet, the combined approach with 8 blocks reduces the over-approximation by 50% to 60% and with 12 blocks results in exact or nearly exact results. When the number of distinct memory blocks exceeds the associativity of the cache as with binary search (see Figure 4), the cache-contention approach significantly improves upon the imprecise approach using only the reuse distance, which predicts hardly any cache hits.

The experiments have a short runtime when the number of relevant blocks is small (less than one minute for 8 blocks or less) but the runtime quickly grows to about 15 minutes for 12 blocks (on a 2.3 GHz CPU) and several hours for a complete state-enumeration; even for these simple programs (see the appendix for a detailed evaluation of tractability (runtime)).

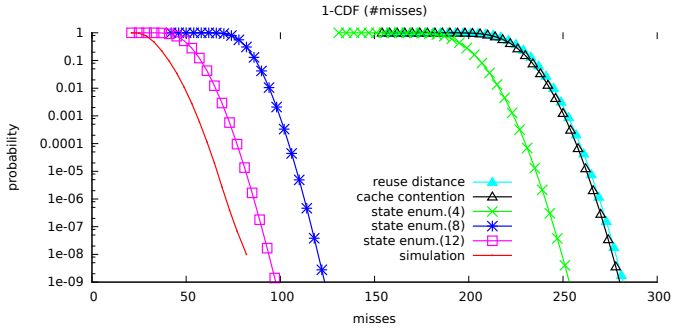

Fig. 3. Insertion Sort. 707 memory accesses in total, 21 distinct memory blocks.

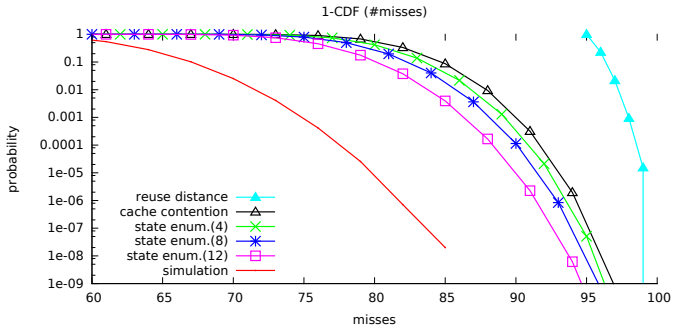

Fig. 4. Binary Search. 132 memory accesses in total, 25 distinct memory blocks.

VII. CONCLUSION AND FUTURE WORK

In this paper, we investigated the correctness, optimality and precision of Static Probabilistic Timing Analysis (SPTA) for systems that use a cache with an evict-on-miss random replacement policy.

The main contributions of this paper are: (i) Showing that the formula for the probability of a cache hit (previously published in DATE 2013 [9]) is not sound for use in SPTA, since it can produce results in the form of probabilistic Worst-Case Execution Time (pWCET) distributions that are optimistic by orders of magnitude and thus unsafe. (ii) Proving the optimality of the probability function given in [7] with respect to the limited information (reuse distance and associativity) that it uses, and deriving an improved probability function that uses information about cache contention. (iii) Introducing an approach with scalable precision, combining precise analysis for frequently used memory blocks with imprecise analysis for those memory blocks that are used less often. Evaluation shows that this technique is effective in reducing pessimism in SPTA without the problems of exponential complexity inherent in an exhaustive cache state exploration.

In future, we intend to investigate how our combined approach may be extended from traces to multi-path programs, improvements that may be obtained by dividing a trace into independent sub-traces (see the appendix for initial work in this area), and explore other heuristics and methods of choosing which memory blocks to select for precise analysis.

Acknowledgments

This work was partially funded by COST Action IC1202: Timing Analysis On Code-Level (TACLe), the UK EPSRC Project MCC (EP/K011626/1), and the EU FP7 Integrated Project PROXIMA (611085).

Revisions

This version of the technical report has been updated to correct an omission in the definition of cache contention, see Section III.

References

[1] S. Altmeyer and R. I. Davis. On the correctness, optimality and precision of static probabilistic timing analysis. In *DATE 2014*, page tbp, 2014. Available from http://www.cs.york.ac.uk/ftpdir/reports/2013/YCS/487/YCS-2013-487.pdf.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[3] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. D. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim. Proartis: Probabilistically analyzable real-time systems. *ACM Trans. Embedded Comput. Syst.*, 12(2s):94, 2013.

[4] L. Cucu-Grosjean. Independence - a misunderstood property of and for probabilistic real-time systems. In N. Audsley and S. Baruah, editors, *In Real-Time Systems: the past, the present and the future*, pages 29–37, 2013.

[5] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiones, and F. J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS 2012*, pages 91–101, 2012.

[6] R. Davis. Improvements to static probabilistic timing analysis for systems with random cache replacement policies. In *RTSOPS 2013*, pages 22–24, 2013.

[7] R. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean. Analysis of probabilistic cache related pre-emption delays. In *ECRTS 2013*, pages 129–138, 2013.

[8] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In *WCET 2010*, pages 137–147, 2010.

[9] L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla. A cache design for probabilistically analysable real-time systems. In *DATE 2013*, pages 513–518, 2013.

[10] E. Quinones, E. D. Berger, G. Bernat, and F. J. Cazorla. Using randomized caches in probabilistic real-time systems. In *ECRTS 2009*, pages 129–138, 2013.

[11] S. Zhou. An efficient simulation algorithm for cache of random replacement policy. In *NPC 2010*, pages 144–154, 2010.

Appendix

This appendix provides a proof of correctness for the cache contention defined in section III. It also provides an extended evaluation covering the four benchmarks from the Mälardalen Benchmark Suite [8] considered in [7], and an investigation into the precision and tractability (runtime performance) of the combined approach to SPTA on larger traces.

A 1. Proof of correctness for Cache Contention

Cache contention is defined in section III as follows. (Note we repeat (11) below as (22) for ease of reference).

$$con\colon \mathbb{E}\times\mathbb{T}\to\mathbb{N}$$
$$con\,(e_l,[e_1,e_2,\ldots,e_{l-1}]) = \begin{cases} \infty & rd(e_l,[e_1,\ldots,l-1]) = \infty \\ \left|conS\right| & \text{otherwise} \end{cases} \tag{22}$$

with

$$conS = \left\{e_i \in [e_1,\ldots,e_{l-1}] \;\middle|\; \left(l - rd(e_l,[e_1,\ldots,l-1]) < i \wedge \hat{P}(e_i^{hit}) \neq 0\right)\right\} \cup \left\{e_r \in [e_1,\ldots,e_{l-1}] \;\middle|\; (l - rd(e_l,[e_1,\ldots,l-1]) = r\right\}$$

To prove that our definition of cache contention is correct, and so eliminates all combinations of cache hits that are infeasible due to the limited associativity of the cache, we must show that there is an evolution of the cache state for the entire trace T that permits *all* accesses assigned a non-zero probability to *all* be hits. (It follows trivially that any subset of these accesses could also all be hits). To do so, we consider the evolution of the set Z which contains the subset of the cache state that is necessary to support cache hits for all of the accesses assigned non-zero probabilities. Let Z_i be this subset of the cache state, immediately following access e_i.

We construct the set Z as follows:

$$Z_1 = \{e_1\}$$

$$Z_i = \begin{cases} (Z_{i-1} \setminus \{e_{i-1}\}) \cup \{e_i\} & \text{if } con(e_{n_{i-1}},T) \geq N \\ (Z_{i-1} \cup \{e_i\}) & \text{if } con(e_{n_{i-1}},T) < N \end{cases} \tag{23}$$

where $e_{n_{i-1}}$ denotes the next access to element e_{i-1}. (Note if there is no such next access, then we assume that $con(e_{n_{i-1}},T) = \infty$). We now prove that the set Z is always a subset of a feasible cache behaviour. Specifically, (i) it always contains the least recently accessed element, (ii) all elements in Z_i have been accessed prior to element e_i and (iii) the set always contains at most N elements:

(i) $\forall i\colon e_i \in Z_i$,
(ii) $\forall i\colon \forall e_j \in Z_i\colon \exists l \leq i\colon e_j = e_l$, and
(iii) $\forall i\colon Z_i \leq N$.

Conditions (i) and (ii) hold by construction. Further, each element $e_j \in Z_i$ where $e_j \neq e_i$ must have a non-zero probability of being a hit at its next reuse and thus a cache contention smaller than N, otherwise it would have been removed from Z_i thus:

$$\forall e_j \in Z_i\colon e_j \neq e_i \Rightarrow con(e_{n_j},T) < N \tag{24}$$

We now prove condition (iii) by contradiction. We assume (for contradiction) that i is the smallest index such that $|Z_i| > N$ holds. It follows that $|Z_{i-1}| = N$ and $|Z_i| = N+1$, since at most one element can be added to Z_{i-1} to form Z_i (see (23)). As $|Z_{i-1}| > |Z_i|$, then from (23), e_{i-1} must be a member of Z_i

with $e_i \neq e_{i-1}$. It must also be the case that $con(e_{n_{i-1}}, T) < N$, otherwise e_{i-1} would have been removed and we would instead have $|Z_{i-1}| = |Z_i|$.

As $con(e_{n_{i-1}}, T) < N$, it follows that Z_{i-1} contains N elements, all of which have a non-zero probability of being a hit at their next reuse. Further, we know that e_i is not one of these elements, since otherwise we would again have $|Z_i| = |Z_{i-1}|$.

Let $e_j \in Z_i$ be the element with the highest index n_j for its next reuse of all those elements in Z_i with a non-zero probability at their next reuse. (Recall that the next access to the same memory block as e_j is denoted by e_{n_j}). Since Z_{i-1} contains N elements all of which have a non-zero probability of being a hit on their next reuse, but does not contain e_i, then we conclude that there must be at least $N-1$ accesses between e_i and e_{n_j} that are assigned non-zero probabilities.

There are two cases to consider:

Case 1: $j = i$. In this case, we know that $e_i \notin Z_{i-1}$ and that all N elements of Z_{i-1} have non-zero probabilities on their next reuse, it must therefore be the case that there are N accesses between e_j and e_{n_j} with non-zero probabilities. Hence the cache contention of e_{n_j} is at least N which contradicts the assumption that e_{n_j} has a non-zero probability of being a cache hit. Hence it cannot be the case that $j = i$.

Case 2: $j < i$. In this case, we know that $e_j \in Z_{i-1}$, $e_j \neq e_i$, and that there must be at least $N-1$ accesses between e_i and e_{n_j} that are assigned non-zero probabilities and so contribute to the cache contention of e_{n_j}. Further, there is also the access e_{j+1} immediately following e_j (where $j+1 \leq i$), which also contributes 1 to the cache contention of e_{n_j} irrespective of the probability assigned to it. Hence the cache contention of e_{n_j} is at least N which contradicts (24).

Both cases lead to contradictions, hence the assumption that $\exists i\colon Z_i = N+1$ is wrong, and thus condition (iii) $\forall i\colon Z_i \leq N$ holds.

The cache contention embodied in (22) is therefore sufficient to ensure that there is a feasible evolution of the cache state, following the construction rules given in (23) that permits all accesses assigned a non-zero probability to be cache hits.

A 2. Extended Evaluation

In this section, we present results for all four Mälardalen benchmarks considered in [7]: binary search, fibonacci calculation, faculty computation and insertion sort. The execution traces for these benchmarks were extracted using the libFirm compiler framework[1] and only contain memory blocks compiled from the source files (i.e. not including any library or operating system calls).

Our experiments assumed a fully-associative instruction-only cache with an associativity of 16 and a block-size of 8. The results are shown in Figures 5 to 8. To derive an approximation to the actual performance of the cache, and thus a baseline for our experiments, we performed 10^9 simulations of the cache behaviour for each benchmark (red line). The other lines on the graphs are the imprecise approach using only the reuse distance (light blue line), the cache-contention approach (black line), and the combined approach using 4, 8 and 12 relevant memory blocks (green, dark blue and pink lines respectively). When the number of pairwise distinct memory blocks exceeds the associativity of the cache as is the case with binary search (see Figure 5) and faculty computation (see Figure 6), then

[1] http://pp.ipd.kit.edu/firm/Index

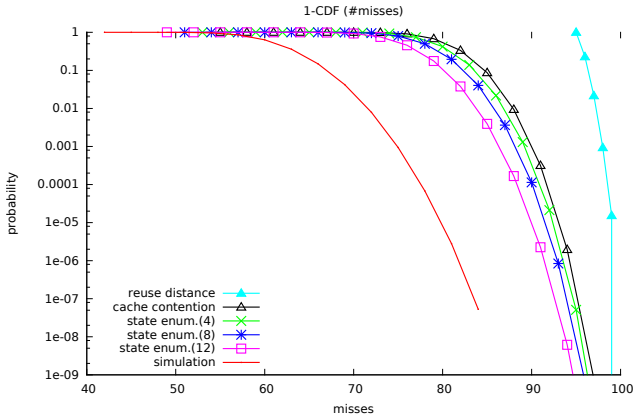

Fig. 5. Binary search. 132 memory accesses in total, 25 distinct memory blocks.

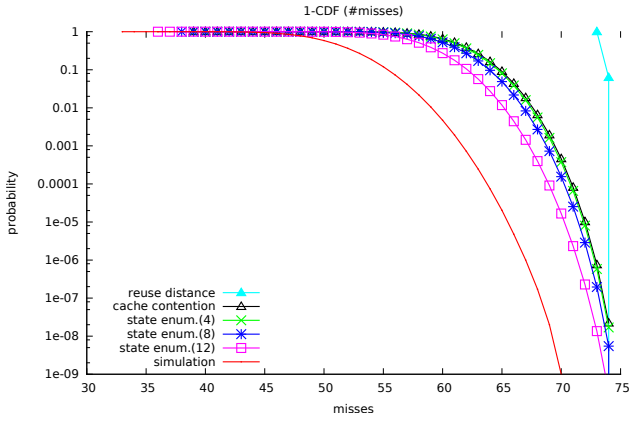

Fig. 6. Faculty computation. 99 memory accesses in total, 25 distinct memory blocks.

the cache-contention approach significantly improves upon the imprecise approach that uses only the reuse distance. Note for these benchmarks, the latter approach predicts hardly any cache hits. For the other two benchmarks: insertion sort (see

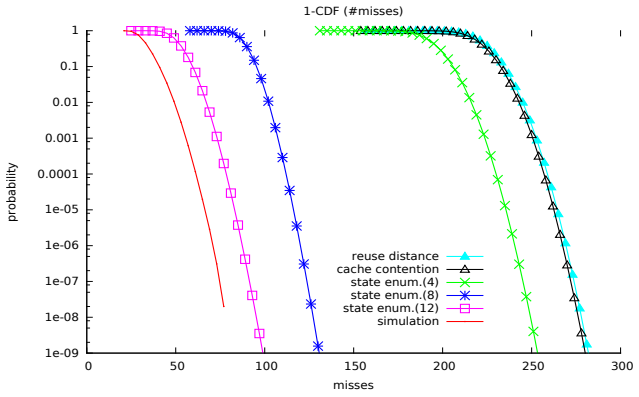

Fig. 7. Insertion sort. 707 memory accesses in total, 21 distinct memory blocks.

Figure 7) and fibonacci calculation (see Figure 8), the number of distinct memory blocks is smaller than or close to the cache associativity. In these cases, the cache-contention approach results in no improvement or only a slight improvement over the imprecise approach that uses only the reuse distance. The

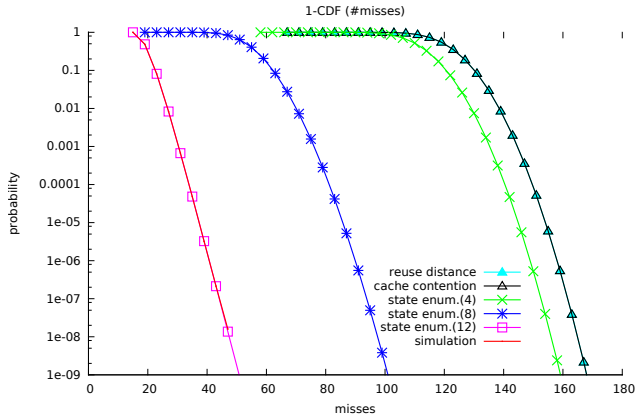

Fig. 8. Fibonacci calculation. 340 memory accesses in total, 15 distinct memory blocks.

combined approach; however, with 8 blocks reduces the over-approximation by 50% to 60%, and with 12 blocks results in exact or nearly exact results.

Figure 9 shows how the runtime of the combined analysis varies with the number of *relevant* cache blocks. (Note the logarithmic scale on the graph). The experiments were run on a 4-core 64-bit 1.2GHz CPU. The graph shows that the runtime of the analysis is exponential in the number of relevant blocks. This indicates that a complete analysis, i.e. where all blocks are considered relevant, is computationally infeasible: Even for the small number of distinct memory blocks and the short access traces, a precise analysis for the binary search and faculty computation benchmarks requires several hours of computation, and so an approximation is necessary.

Observe that the runtime of the combined analysis for the insertion sort benchmark remains constant for 14 or more relevant blocks as the total number of reused memory blocks is 13 in this case. Hence, increasing the number of relevant blocks does not affect the precision or the runtime of the analysis. The same holds for the fibonacci benchmark and 13 blocks. Here, the flattening of the runtime between 8 and 10 relevant blocks is caused by the specific structure of this benchmark. The 9th and 10th most-frequent used memory blocks only occur in the final tenth of the access trace, which means that the exponential growth of the number of states does not fully affect the runtime when increasing the number of relevant blocks from 8 to 10.

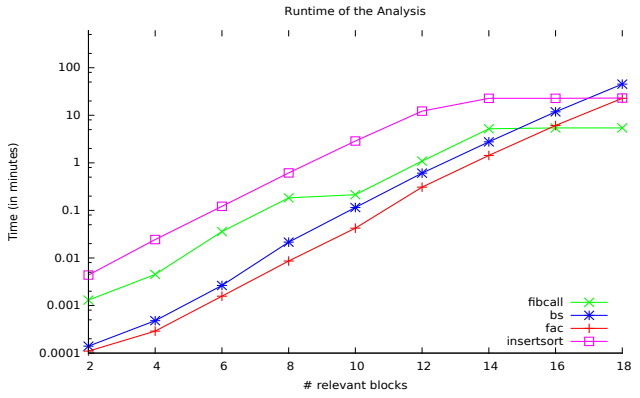

Fig. 9. Runtime evaluation for the benchmarks fibonacci calculation, insertion sort, faculty computation and binary search

A 3. Precision and Performance on Larger Traces

In addition to the benchmarks used in [7], we generated cache access traces for the four largest Mälardalen benchmarks [8] using the gem5 simulator [2] assuming an *ARM7* processor[2]. In contrast to the other benchmarks, the gem5-traces contain library and operating-system calls and are thus significantly larger. We assumed the same cache configuration: a fully-associative instruction-only cache with an associativity of 16 and a block-size of 8. The results are shown in Figures 10 to 13.

For these larger traces, we used a different heuristic to select the *relevant* memory blocks. Instead of defining one fixed set of relevant blocks for the complete trace, we define the set of relevant blocks depending on the position within the trace. We first determine all reused memory blocks in the complete trace, and the points at which they are last used. Then starting from the beginning of the trace, we greedily collect the reused memory blocks until we have a set of k relevant blocks. When we encounter the last reuse of a block, that block is removed from the set of relevant blocks. Further, whenever, the set of relevant blocks contains fewer than k relevant blocks, the next reused block encountered is added to the set. This heuristic accounts for memory blocks with a high locality that occur frequently in one sub-trace and infrequently in others. By construction, at most k memory blocks are relevant at any point in the trace, limiting the number of different cache states to 2^k. (We note that the number of relevant blocks is often significantly lower then k for some sub-traces). In

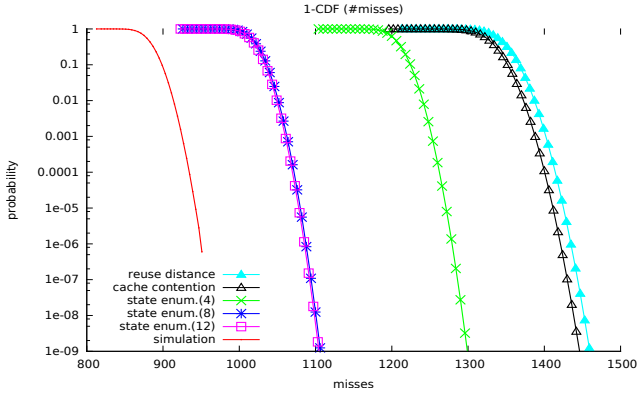

Fig. 10. Finite impulse response filter (fir). 3419 memory accesses in total, 393 distinct memory blocks.

this set of experiments, the cache contention approach always improves upon the imprecise approach that uses only the reuse distance: in the case of the finite impulse filter benchmark (see Figure 10), the over-approximation is reduced by around two percent, whereas in the case of the petri-net simulation (see Figure 12) the improvement is around 13%. In all four cases, the combined approach with 12 relevant blocks provides highly accurate results, with an over-approximation of at most 15% (Figure 11) and a near-optimal estimate for the petri-net simulation (see Figure 12).

Figure 14 shows the runtime of the combined analysis for different numbers of relevant cache blocks for each of the second set of benchmark traces. We note again the exponential growth in runtime with an increasing number of relevant blocks; however, here the runtime of the analysis is significantly higher

[2]http://www.arm.com/products/processors/classic/arm7

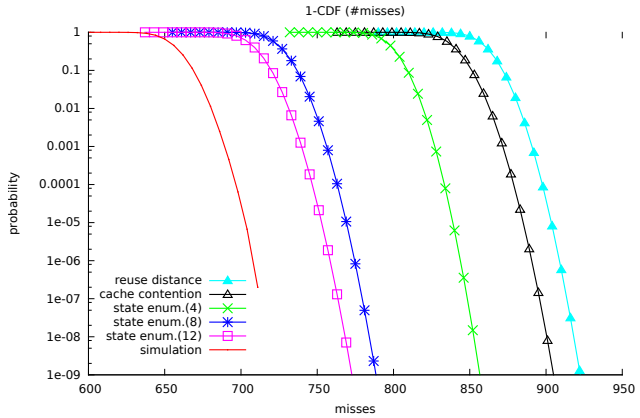

Fig. 11. Discrete-cosine transformation on a 8x8 pixel block (jfdctint). 2185 memory accesses in total, 347 distinct memory blocks.

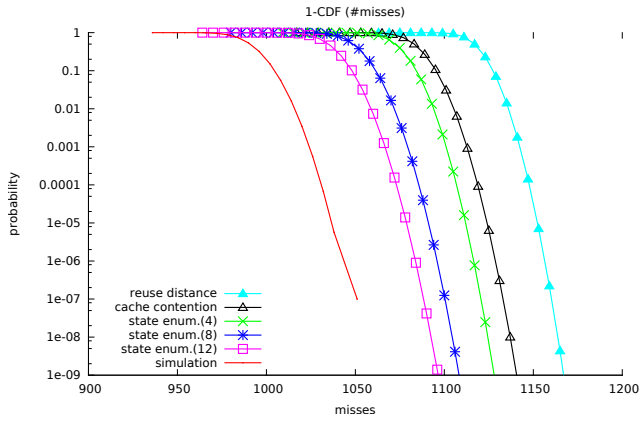

Fig. 12. Automatically generated code (statemate). 1831 memory accesses in total, 394 distinct memory blocks.

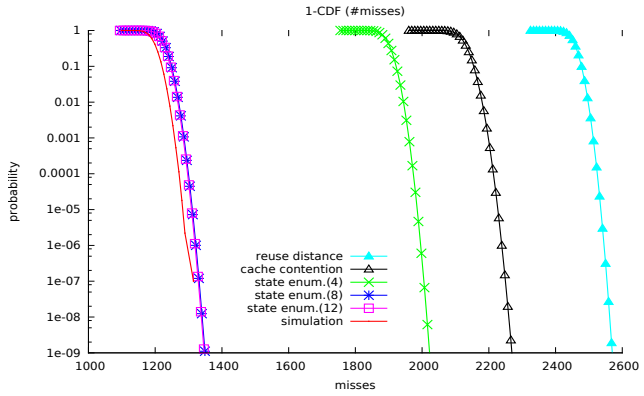

Fig. 13. Simulation of an extended Petri Net (nsichneu). 5202 memory accesses in total, 454 distinct memory blocks.

due to the larger traces. The heuristic conceptually splits the traces into sub-traces enabling different sets of relevant blocks to be defined per sub-trace, while damping the exponential growth in runtime. Since the number of pairwise distinct memory blocks significantly exceeds the number of relevant blocks, we could not reach any saturation, in terms of relevant blocks, for any of these benchmarks within a reasonable time (unlike the insert sort and fibonacci benchmark in Figure 9).

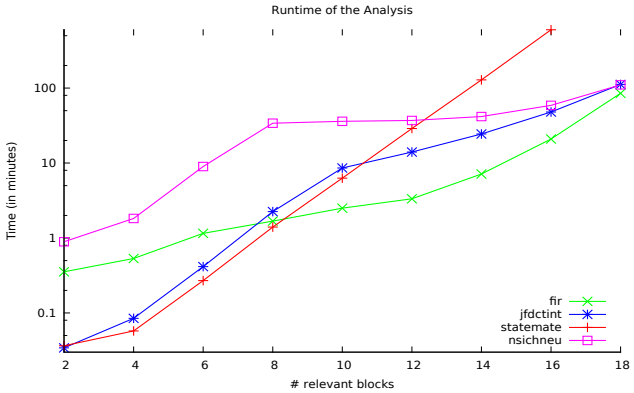

Fig. 14. Runtime evaluation for the benchmarks fir, jfdctint and nsichneu