Speeding up Static Probabilistic Timing Analysis

Suzana Milutinovic[1,2], Jaume Abella[2], Damien Hardy[3], Eduardo Quiñones[2], Isabelle Puaut[3], and Francisco J. Cazorla[2,4]

[1] Universitat Politecnica de Catalunya (UPC), Spain.
[2] Barcelona Supercomputing Center (BSC-CNS), Spain
[3] IRISA, France
[4] Spanish National Research Council (IIIA-CSIC), Spain

Abstract. Probabilistic Timing Analysis (PTA) has emerged recently to derive trustworthy and tight WCET estimates. Computational costs due to the use of the mathematical operator called *convolution* used by SPTA – the static variant of PTA – and also deployed in many domains including signal and image processing, jeopardize the scalability of SPTA to real-size programs. We evaluate, qualitatively and quantitatively, optimizations to reduce convlution's computational costs when it is applied to SPTA. We showthat SPTA specific optimizations provide the largest execution time reductions, at the cost of a small loss of precision.

1 Introduction

Probabilistic Timing Analysis (PTA) [2, 3, 5, 7, 9, 16] has emerged recently as a powerful family of techniques to estimate the worst-case execution time (WCET) of programs. Recent PTA techniques advocate for hardware and software designs that either have fixed latency or randomized timing behavior [5, 7, 11, 10], to produce WCET estimates that can be exceeded with a given – arbitrarily low – *probability*, which are typically referred to as probabilistic WCET (pWCET) estimates. Using those hardware and software designs increases coverage (and so usefulness) of pWCET estimates [6]. Examples of time-randomized hardware elements are caches with random placement and/or replacement [13, 11, 5].

The static variant of PTA, called SPTA, has recently been object of intense study [5, 8, 2, 14]. In this paper we contribute to SPTA development by identifying and mitigating one of the major bottlenecks for SPTA to scale to industrial-size programs: its execution time requirements.

Under SPTA, each instruction has a probabilistic timing behavior represented with an Execution Time Profile (ETP). An ETP is expressed by a timing vector that enumerates all the possible latencies that the instruction may incur, and a probability vector, which for each latency in the timing vector, lists the associated probability of occurrence. Hence, for an instruction $\mathcal{I}_i$ we have $ETP(\mathcal{I}_i) = <\vec{t_i}, \vec{p_i}>$ where $\vec{t_i} = (t_i^1, t_i^2, ..., t_i^{N_i})$ and $\vec{p_i} = (p_i^1, p_i^2, ..., p_i^{N_i})$, with $\sum_{j=1}^{N_i} p_i^j = 1$. The *convolution* function, $\otimes$, is used to combine ETPs, such that a new ETP is obtained representing the execution time distribution of the execution of all the instructions convolved.

With real-time programs growing in size, the need to carry out a convolution operation for every instruction in the object code may incur high computation time requirements. Hence, efficient ways to perform convolutions in the context of SPTA are needed. In this paper we analyze a number of optimizations of the convolution operation. Some optimizations keep precision, whereas some others sacrifice some precision to reduce computational cost, while preserving WCET trustworthiness.

- Among precision-preserving optimizations we consider convolution parallelization, as largely studied previously in the literature [15, 17], in 2 forms: (1) *inter-convolution parallelization*, where ETPs to be convolved are split into several groups that are convolved in parallel and (2) *intra-convolution parallelization* where one (or both) of the ETPs to be convolved is split into sub-ETPs so that each sub-ETP is convolved with the other ETP in parallel.
- Among optimizations that sacrifice some precision to reduce convolution cost, we consider (3) *discretization*, such that few different forms of ETPs exist and convolutions across identical ETPs need not be carried out too often. We also consider (4) *sampling* where several elements in the ETP are collapsed into one [12], thus reducing the length of the ETPs to be convolved and so the number of operations.

Our results show that discretization and sampling – the SPTA specific optimizations – lead to the highest reductions in execution time, whereas the combination of intra- and inter-convolution parallelization provides second order reductions in execution time. In particular, discretization and sampling reduce execution time by a factor of 10 whereas precision-preserving optimizations reduce it by a factor of 2. This execution time reduction comes at the expense of a pWCET increase around 3%.

Another approach to speed-up convolutions is to use Fourier Transformation, and in particular its discrete fast version (DFT). This approach needs first to convert the distribution from the time domain to the frequency domain using DFT. Then, according to the convolution theorem, a point-wise multiplication is applied, which is equivalent to the convolution in the time domain. Finally, inverse DFT is performed to obtain the distribution in the time domain. Evaluating DFT to speed up convolutions is left for future work.

The rest of the paper is organized as follows. Section 2 provides background on PTA and convolutions. Section 3 presents issues challenging SPTA scalability and optimizations to reduce its computational cost. Optimizations are evaluated in Section 4. Finally, Section 5 concludes the paper.

2 Background: PTA and convolutions

Along a given path, assuming that the probabilities for the execution times of each instruction are independent, SPTA is performed by deploying the discrete convolution ($\otimes$) of the ETPs that describe the execution time for each instruction along that path. The final outcome is a probability distribution representing the

Algorithm 1 Convolution canonical implementation

```
1: c ← 1
2: for i = 1 to N do
3:    for j = 1 to N do
4:       etpr.lat[c] ← etp1.lat[i] + etp2.lat[j]
5:       etpr.prob[c] ← etp1.prob[i] ∗ etp2.prob[j]
6:       c ← c + 1
7:    end for
8: end for
```

timing behavior of the entire execution path. For the sake of clarity we keep the discussion at the level of a single execution path.

More formally, if $\mathcal{X}$ and $\mathcal{Y}$ denote the random variables that describe the execution time of two instructions x and y, the convolution $\mathcal{Z} = \mathcal{X} \otimes \mathcal{Y}$ is defined as follows: $P\{\mathcal{Z} = z\} = \sum_{k=0}^{k=+\infty} P\{\mathcal{X} = k\}P\{\mathcal{Y} = z - k\}$. For instance if an instruction x is known to execute in 1 cycle with a probability of 0.9 and to execute in 10 cycles with a probability of 0.1 and an instruction y has an equal probability of 0.5 to execute in 2 or 10 cycles, we have:

$$\begin{aligned}\mathcal{Z} = \mathcal{X} \otimes \mathcal{Y} &= (\{1, 10\}, \{0.9, 0.1\}) \otimes (\{2, 10\}, \{0.5, 0.5\}) \\ &= (\{3, 11, 12, 20\}, \{0.45, 0.45, 0.05, 0.05\})\end{aligned}$$

For every static instruction, i.e. instruction in the executable of the program, SPTA requires that their ETPs are not affected by the execution of previous instructions. When time-randomized caches are used, there is an intrinsic dependence among the hit probability of an access (P_{hit}) and the outcome of previous cache accesses [5, 8]. Existing techniques to break this dependence create a lower bound function to P_{hit} (so an upper bound to P_{miss}) of every instruction to make it independent – for WCET estimation purposes – from previous accesses [5, 8, 2]. Given that those methods are orthogonal to the cost of convolutions, we omit details and refer the interested reader to the original works.

3 SPTA: performance issues and optimizations

When implementing ETP convolution it is convenient to operate normalized ETPs (ETPs whose latencies are sorted from lowest to highest). Canonical convolution of normalized ETPs then consists of three steps: *convolution, sorting* and *normalization*. *Convolution* per se, shown in Algorithm 1, consists of multiplying each pair of probabilities from both ETPs and adding their latencies. After convolution, latencies in the result ETP are not sorted anymore, which is corrected by the *sorting step*. *Normalization*, shown in Algorithm 2, then removes repeated latencies in ETPs; it combines consecutive repeated latencies by adding up their probabilities.

Given two normalized ETPs of N elements each, convolution per se, has a complexity of $\mathcal{O}(N^2)$, and the resulting ETP contains N^2 elements. The complexity of sorting the N^2 elements is $\mathcal{O}(N^2 \log N^2)$. However, the resulting ETP

Algorithm 2 Normalizing function

```
1: c ← 0
2: etp_out.lat[0] ← etp_in.lat[0]
3: for i = 1 to N do
4:    if etp_in.lat[i] = etp_in.lat[i − 1] then
5:       etp_out.prob[c] ← etp_out.prob[c] + etp_in.prob[i]
6:    else
7:       c ← c + 1
8:       etp_out.lat[c] ← etp_in.lat[i]
9:       etp_out.prob[c] ← etp_in.prob[i]
10:   end if
11: end for
```

contains N blocks of N elements, each block sorted internally, which reduces computational cost in practice down to $\mathcal{O}(N^2)$. The cost of normalization is linear with the number of elements in the ETP.

Starting from the canonical convolution, we survey optimizations related to (i) the cost of each individual operation, (ii) parallelization, (iii) sampling and (iv) discretization. Experimental results are shown in Section 4.

3.1 Cost of each operation

The main particularity when convolution is applied to SPTA is that SPTA works with very small probabilities (e.g. 10^{-30}) due to the fact that multiplication of probabilities during convolution leads to lower values for probabilities with an increased number of decimal digits. Operating with such low values makes IEEE 754 standard floating-point (FP) representations inaccurate. For instance, 64-bit double precision FP IEEE 754 numbers use 52 binary digits for the fraction, which allows representing up to 15 decimal digits approximately. To avoid issues with precision, arbitrary-precision FP (*apfp*) numbers can be used. *apfp* precision is not limited by fixed-precision arithmetic implemented in hardware. This increase in precision is provided at the cost of significant longer latency to carry out each operation, as each operation may require dozens of assembly instructions. The impact of the *apfp* precision on the execution time of convolutions will be studied in Section 4.

3.2 Parallelization

Parallelization can be applied across different convolution operations on different ETPs (inter-convolution parallelism) or in the convolution of a pair of ETPs (intra-convolution parallelism).

Intra-convolution parallelism. Given two ETPs with N and M elements respectively, convolution requires adding the latencies and multiplying the probabilities for the $N \times M$ different pairs of elements from both ETPs. Dividing such work into T parts to be performed in parallel can be done in many different ways. In our case, we divide the N-point ETP_1 (or ETP_2) into T subETPs

of N/T points each, ETP_1^{part} where T is the number of cores/processors used. Each such ETP_1^{part} can be convolved with ETP_2 in parallel. The result of this step are T different ETPs. Those have to be concatenated and normalized to become the final outcome of the convolution of ETP_1 and ETP_2.

Inter-convolution parallelism. In the case of SPTA, typically each instruction has its own ETP. Programs may have easily thousands if not hundreds of thousands of instructions. Hence, convolutions can be performed in parallel. Given a list of M ETPs to be convolved, our approach consists in splitting the list into T chunks of $Mc = M/T$ ETPs each. Each chunk to be convolved is assigned to a different core or processor. Two approaches can be followed to convolve the ETPs in each chunk:

Sequential order within a chunk. The first two ETPs (e.g., of N elements each) are convolved, which requires N^2 operations if sorting and normalization of the resulting ETP are omitted, and generates an ETP with up to N^2 elements, which in a following step is convolved with the third ETP requiring up to N^3 operations. Equation 1 shows the maximum number of operations carried out with this approach.

$$OpCount_{seq}^{Mc} = \sum_{i=2}^{Mc} \left(N^i\right) \tag{1}$$

Tree reduction within a chunk. In a first step, the Mc ETPs (each of N elements) are convolved in pairs, so each convolution requires N^2 operations. In a second step, the resulting $Mc/2$ ETPs, each of up to N^2 elements, are convolved in pairs requiring up to N^4 operations each and resulting in $Mc/4$ ETPs. Equation 2 shows in the general case the maximum number of operations carried out with this approach.

$$OpCount_{tree}^{Mc} = \sum_{i=1}^{\lceil \log_2 Mc \rceil} \left(\frac{Mc}{2^i} \times N^{2^i} \right) \tag{2}$$

If the number of ETPs is not a power-of-two, the tree reduction approach requires an adjustment phase. Given M ETPs, we convolve as many pairs as needed so that we obtain M' ETPs where M' is a power-of-two.

3.3 Sampling

When two ETPs of N elements are convolved the resulting ETP may have up to N^2 elements. Hence, there is an exponential increase in the number of elements in the result ETP as the number of convolutions increases. In order to limit the number of elements in the ETP, sampling techniques can be used [12].

The principle of sampling, largely used in the literature, is reducing the number of points in the ETPs. In a real-time context, an additional requirement is to ensure that the new ETP is an upper-bound of the original one, so that pWCETs are never underestimated. This is done by collapsing probabilities to

the right [12]. For instance, $ETP1 =< (1,2,3,4),(0.2,0.1,0.5,0.2) >$ could be sampled as $ETP1' =< (2,4),(0.3,0.7) >$ or $ETP1' =< (3,4),(0.8,0.2) >$.

There are several ways of sampling an ETP such that, while ensuring it is a safe upper-bound of the original one, the pessimism introduced is kept low [12]. As shown in [12], sampling makes convolution cost to flatten asymptotically so that it does not grow exponentially.

3.4 Discretization of probabilities

In order to introduce discretization we use an example. Let us assume an architecture in which each instruction can take exactly two latencies (e.g., cache hit and cache miss [11]). Discretization consists in rounding probabilities such that the probability of the highest latency is rounded up and the one of the lowest latency is rounded down. For instance, given $ETP =< (1,20),(0.24,0.76) >$, if we round to a given fraction, e.g. 0.1, this would result in $ETP_{rounded} =< (1,20),(0.2,0.8) >$. Overall, rounding consists in adding ϵ to the probability of the high latency (and subtracting ϵ from the probability of low latency) such that it becomes a multiple of a given rounding value rv, where $rv \leq 1$ and $1 \ mod \ rv = 0$, so that $(p_{high\ lat} + \epsilon) \ mod \ rv = 0$.

Rounding has two effects. On the one hand, the resulting ETP can have only $1/rv + 1$ different forms. On the other hand, the probability of high latencies is increased, thus inducing higher pessimism. Similarly to sampling, discretization reduces precision. However, those optimizations sacrifice precision in a controlled and trustworthy way from a WCET estimation perspective (the resulting ETP always upper-bounds the exact one).

In the presence of an M-element vector of ETPs, in a first pass all the probabilities of the ETPs are rounded as explained resulting in g different forms of ETP, with $g = 1/rv + 1$. The convolution of N copies of the same ETP can be done much faster than the normal convolution. This is explained later in this section.

After the first step, there are up to g ETPs to convolve, with g being typically a relatively low value (e.g., $g = 101$ if $rv = 0.01$). Those ETPs can be convolved in parallel applying any of the techniques explained before.

Convolution of E copies of the same ETP. Convolving E times an ETP consists, in essence, of applying the power operation. In order to reduce the execution time of the power operation of convolutions we need to decompose E into an addition of power-of-two values. For instance, $E = 7$ can be decomposed into 4, 2 and 1. In this case we convolve $ETP_1^{pow(2)} = ETP_1 \otimes ETP_1$. In a second step we convolve $ETP_1^{pow(4)} = ETP_1^{pow(2)} \otimes ETP_1^{pow(2)}$. The final ETP can be obtained by convolving at most all those ETPs as shown in Equation 3.

$$ETP_1^{pow(7)} = ETP_1^{pow(4)} \otimes ETP_1^{pow(2)} \otimes ETP_1^{pow(1)} \tag{3}$$

In general, generating the power-of-two ETPs requires performing $\lceil log_2 E \rceil - 1$ convolutions. Then, at most each such ETP (including the original one, ETP_1) needs to be convolved once, thus requiring up to $\lceil log_2 E \rceil - 1$ extra convolutions.

Overall, with this approach the power of a given ETP can be carried out with at most $2 \times (\lceil log_2 E \rceil - 1)$ convolutions, whereas the sequential approach requires $E - 1$ convolutions.

4 Experimental Results

In this section we evaluate the execution time reduction and pessimism increase of the techniques presented when applied in isolation and in a combined manner. The number of configurations and results presented is limited due to space constraints. All these optimizations have been integrated into an ETP management library, developed in C++.

4.1 Experimental conditions

Platform and *apfp* library. We use a quad-core AMD Opteron^{TM} processor connected to a 32GB DDR2 667 MHz SDRAM. We run a standard Linux distribution on top of it. For arbitrary-precision FP computations we use the GNU *mpfr* (multiple-precision FP) Library, `http://www.mpfr.org/`.

The precision of the *mpfr* library was selected according to the criticality level of the target applications. Obviously, the higher the precision the longer takes each operation to execute and the higher are the memory requirements of the library. As an example, for commercial airborne systems at the highest integrity level, called DAL-A, the maximum allowed failure rate per hour of operation [1] in a system component is 10^{-9}. Thus, if a task is fired up to 10^2 times per second, it can be run up to 3.6×10^5 times per hour, and so its probability of timing failure per activation, TPF_{act} should not exceed 3.6×10^{-14}. Therefore, an exceedance probability threshold of 10^{-15} ($TPF_{act} \leq 10^{-15}$) suffices to achieve the highest integrity level. Similarly, exceedance probability thresholds can be derived for other domains and safety levels. We have observed empirically that even if millions of multiplications are performed, a precision of 20 decimal digits suffices to keep accurate results for the 15^{th} decimal digit (and beyond). This means that when enforcing the 20^{th} decimal digit to be rounded up or down for trustworthiness reasons, such pessimism does not propagate up to the 15^{th} decimal digit. Thus, we regard 20 decimal digits as enough for our needs, and select this value as a default value in the experiments. The impact of this parameter in terms of computation cost is studied later in this section. A sensitivity study of the impact of this parameter on pessimism has not been performed due to space constraints, but our choice limits such pessimism to much less than 0.01% in practice in all our experiments.

Optimization parameters. When applying inter-convolution parallelism, one has to choose between *tree reduction* and *sequential order* when convolving the ETPs within each parallel chunk. Tree reduction typically requires fewer operations than those required with sequential processing ETPs (up to 50% fewer operations). However, it makes ETP size grow faster until their maximum size,

which is limited by calling the *sampling* function. Hence with tree reduction most operations require working with two ETPs of E elements. Instead, sequential order also make intermediate ETPs grow up to E elements, but keeps convolving them with N-elements ETPs, with $N << E$. Overall, sequential order works faster than tree reduction so it is our default choice in the rest of the paper.

As far as sampling is concerned, many sampling methods have been defined and compared in [12]. Among those, we use *uniform space sampling*, as it provides a good balance among execution time requirements and pessimism introduced. In the experiments, unless otherwise stated, sampling will be systematically applied, and the size of ETPs will be limited to 1,024 elements. If larger ETPs are explicitly used (i.e. 2,048 or 4,096 elements) and sampling is applied, the size of the original ETPs determines the size of the output ETPs.

Test-case generation and metrics. In each experiment we use several ETPs with different number of elements. These input ETPs have been generated randomly. To measure the improvement brought by each optimization, we use the execution time reduction, typically w.r.t. non-optimized execution in a single core. Optimizations studied are orthogonal to other methods for convolution optimization [4] whose analysis is beyond the scope of this paper. Pessimism resulting from some optimizations (sampling and discretization) is also computed w.r.t. to the non-optimized results. Pessimism is measured in terms of weight of the ETP, which is obtained as $W = \sum_{i=1}^{N} p_i \times l_i$ where N is the number of elements in the ETP, and p_i and l_i are the probability and latency at position i respectively [12]. Then, the weight of the ETP after optimizations (W_{optim}) is compared w.r.t. to the ETP without optimizations ($W_{baseline}$).

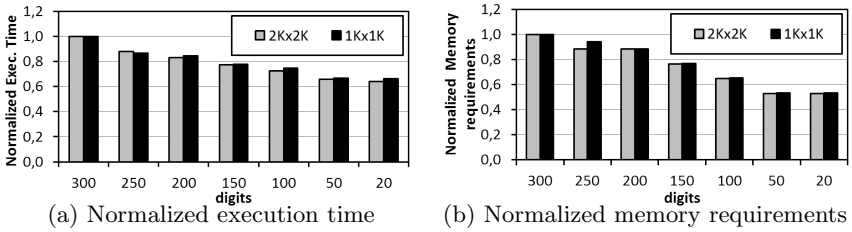
4.2 Impact of *apfp* and *mpfr* precision on the cost of each operation

Fig. 1. Cost of each operation normalized to native ISA FP add operation

ISA					*apfp*				
≥	=	+	*	/	≥	=	+	*	/
1	1	1	2	3	5	22	17	36	75

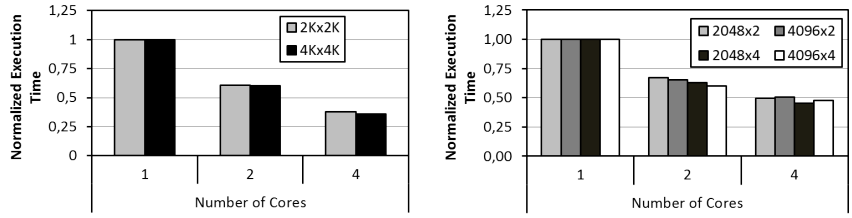
To evaluate the price to pay for having sufficient precision in the ETPs, we first evaluate the execution time of each basic operation used by convolutions (comparison, assignment, addition, multiplication, division). All values are normalized to the execution time of the native FP addition operation, i.e. the operation to add FP numbers in the ISA. Results have been obtained by running on our processor a set of micro-benchmarks that exercise the same number of operations of each type.

The results are given in Fig. 1, with the precision of the *apfp* library set to a high value, 300 digits. We observe that the impact of the *apfp* library is significant. The *apfp* operation with lower overhead, the comparison, has an execution time 5x higher than an ISA regular FP comparison. We attribute this to the fact that it is often completed after comparing only a subset of the digits. Addition and assignment have a similar slowdown around 20x while multiplication and division have a latency 36x and 75x higher than the ISA addition respectively. This represents an increment of more than 22x and 26x w.r.t. their ISA counterparts.



(a) Normalized execution time (b) Normalized memory requirements

Fig. 2. Execution time and memory requirements for different *mpfr* library precisions



(a) E.T. of intra-ETP parallelization (b) E.T. of inter-ETP parallelization

Fig. 3. Impact of parallelization on execution time

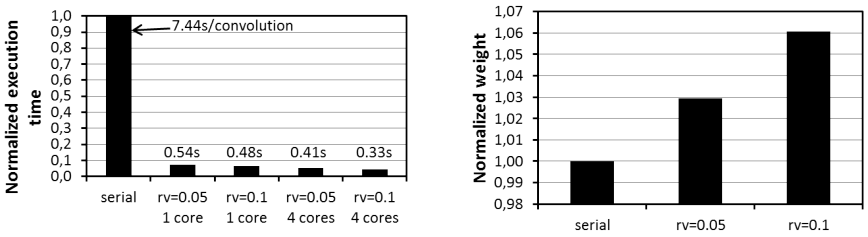
To further evaluate the impact of the *apfp* library precision, we run a single-threaded version of the convolution varying the precision of *mpfr* from 300 digits down to 20, which is considered reasonable for SPTA as explained earlier. Fig. 2(a) and Fig. 2(b) respectively show the reduction in execution time and memory requirements as the number of digits decreases from 300 to 20 when convolving two ETPs. Two ETP sizes are evaluated: 2,048 (i.e *2K*) and 1,024 (i.e. *1K*), and sampling is applied. We observe significant reductions of more than 35% and 45% in execution time and memory respectively when moving from 300 to 20 digits, for both ETP sizes.

4.3 Parallelization

Intra-ETP parallelization. In this experiment we carry out the convolution of 2 ETPs in parallel, with sorting, sampling and normalization turned off. Only the first step of canonical convolution (see Section 3) is executed in parallel and measured. In this way, we obtain an upper-bound of the execution time reduction (scalability) of intra-convolution parallelism. Two different ETP sizes are evaluated: 2,048 (2K) and 4,096 (4K).

Fig. 3(a) shows the execution time results when running the convolution on 1, 2 and 4 cores. We observe good scalability: execution time reduces by 40% with 2 cores and by 65% with 4 cores. The size of the ETPs has a marginal impact.

Inter-ETP parallelization. In contrast to intra-ETP parallelization, inter-ETP parallelization does not parallelize one convolution, but instead splits a sequence of convolutions into chunks to be processed in parallel. In this experiment, given a vector of M ETPs to convolve, we measure the benefit of dividing



(a) Run time of discretization (b) Pessimism introduced

Fig. 4. Evaluation of the *Discretization* optimization

it into $T \in [1, 4]$ chunks, which are processed in parallel (each chunk in one core). The ETPs in each chunk are processed in sequential order.

Fig. 3(b) shows the execution time reduction of inter-ETP parallelization when convolving vectors of 2,048 and 4,096 ETPs (e.g., 4096x2 in the legend stands for 4,096 ETPs of 2K elements each). Results are also shown across different numbers of elements per ETP, namely, 2 and 4. Results do not reach optimal scaling due to: (i) the intrinsic overhead of parallelization (e.g., spawning and synchronizing threads) and (ii) because eventually the number of ETPs to convolve is lower than the core count, thus leaving some cores idle. As it can be observed in Fig. 3(b), the number and size of the ETPs has marginal impact on execution time.

4.4 Probability discretization

In this experiment, we assess the execution time benefits and impact on pessimism introduced by probability discretization. For this experiment we carry out the convolution of a vector of 4,096 ETPs of 2 elements each[1]. Probabilities of those ETPs are randomly generated, latencies are $l_{hit} = 1$ and $l_{miss} = 60$. We carry out the evaluation for two different rv values: 0.05 and 0.1.

Fig. 4 shows the results, obtained by averaging the ETP weight and execution times on 1,000 runs. When run on one single core (two leftmost bars in Fig. 4(a)), we observe that with $rv = 0.05$, we obtain an execution time reduction of more than 93% (from 7.44s/convolution down to 0.54). With $rv = 0.1$ there is an additional slight reduction in the execution time. However, in terms of pessimism (ETP weight, shown in Fig. 4(b)), $rv = 0.05$ shows to have low pessimism. The increase in pessimism of $rv = 0.1$ does not pay off its additional small reduction in execution time.

Fig. 5 compares the pWCET estimates obtained after convolving 4,096 random ETPs when discretization is not applied, and when it is applied with $rv = 0.05$ and $rv = 0.1$. We observe that with discretization pWCET estimates obtained are more pessimistic than when not using discretization. However, the pessimism introduced is relatively small. For instance, for a cutoff probability of 10^{-12} the overestimation is 3.1% for $rv = 0.05$ and 5.5% for $rv = 0.1$.

[1] A two-point ETP represents an architecture with a single level of cache, e.g. the instruction cache, where each ETP takes the form: $< (l_{hit}, l_{miss}), (p_{hit}, p_{miss}) >$

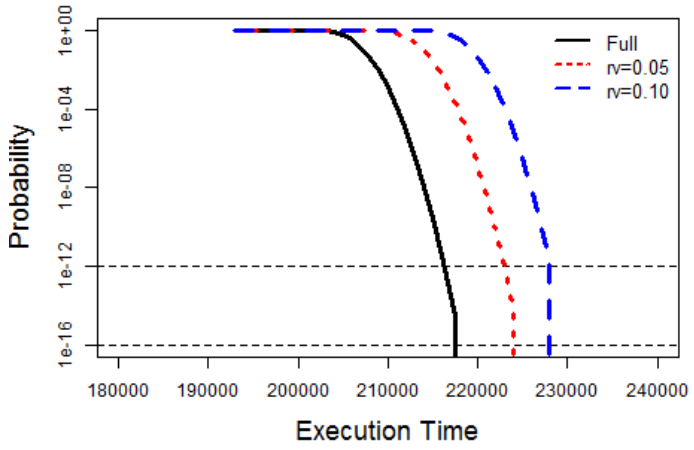


Fig. 5. pWCET estimates with and without discretization

4.5 Combination of techniques

The two rightmost bars in Fig. 4(a) show the result of combining discretization and hybrid parallelization. We observe that the combination of both reduces the cost of convolutions to less than 5% of the cost of the non-optimized convolution method, thus showing that benefits of optimizations increase when combined. In terms of absolute execution time, the cost of one convolution reduces from 7.44s down to 0.33s. Thus, if a program has 100,000 instructions, those optimizations reduce convolution cost from 8.6 days down to 9.2 hours. While such cost is still high, we regard it as affordable and it can be further reduced if other optimizations are applied [4] (e.g., fast-fourier transformation).

5 Conclusions

PTA has been regarded as a powerful approach to obtain trustworthy and tight WCET estimates. The static variant of PTA, SPTA, requires the use of convolutions, whose computational cost is high. In this paper we have identified some features of convolutions that require a large number of computations and provide a set of optimizations to reduce their cost. Those optimizations, integrated into a software library, include precision-preserving optimizations (e.g., parallelization), as well as optimizations that trade off some accuracy for some computational cost reduction while preserving trustworthiness. Among those, discretization shows to be the most effective solution. Our results prove the effectiveness of the different optimizations and a small subset of them show a combined execution time reduction down to less than 5% of that of the non-optimized version.

All in all, SPTA specific optimizations trading off execution time reduction and accuracy show to be the most effective ones and they can be combined straightforwardly with non-specific ones.

Acknowledgments

The research leading to these results has received funding from the European Community's FP7 under the PROXIMA Project, grant agreement no 611085. This work has also been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557, the HiPEAC Network of Excellence, and COST Action IC1202: Timing Analysis On Code-Level (TACLe).

References

1. Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. *ARP4761*, 2001.
2. S. Altmeyer and R.I. Davis. On the correctness, optimality and precision of static probabilistic timing analysis. In *DATE*, 2014.
3. G. Bernat et al. WCET analysis of probabilistic hard real-time systems. In *RTSS*, 2002.
4. A.F. Breitzman. *Automatic Derivation and Implementation of Fast Convolution Algorithms*. PhD thesis, Drexel University, 2003.
5. F.J. Cazorla et al. PROARTIS: Probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems*, 12(2s), 2013.
6. F.J. Cazorla et al. Upper-bounding program execution time with extreme value theory. In *WCET workshop*, 2013.
7. L. Cucu et al. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS*, 2012.
8. R.I. Davis et al. Analysis of probabilistic cache related pre-emption delays. In *ECRTS*, 2013.
9. J. Hansen et al. Statistical-based WCET estimation and validation. In *WCET Workshop*, 2009.
10. Leonidas Kosmidis, Eduardo Quiñones, Jaume Abella, Tullio Vardanega, Ian Broster, and Francisco J. Cazorla. Measurement-based probabilistic timing analysis and its impact on processor architecture. In *17th DSD*, 2014.
11. L. Kosmidis et al. A cache design for probabilistically analysable real-time systems. In *DATE*, 2013.
12. D. Maxim, M. Houston, L. Santinelli, G. Bernat, R.I. Davis, and L. Cucu. Re-sampling for statistical timing analysis of real-time systems. In *RTNS*, 2012.
13. E. Quinones et al. Using Randomized Caches in Probabilistic Real-Time Systems. In *ECRTS*, 2009.
14. J. Reineke. Randomized caches considered harmful in hard real-time systems. *Leibniz Transactions on Embedded Systems*, 1(1), 2014.
15. C.J. Turner et al. Parallel implementations of convolution and moments algorithms on a multi-transputer system. *Microprocessors and Microsystems*, 19(5), 1995.
16. F. Wartel et al. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *SIES*, 2013.
17. H.-M. Yip et al. An efficient parallel algorithm for computing the gaussian convolution of multi-dimensional image data. *J. Supercomput.*, 14(3), 1999.