Mitigating Software-Instrumentation Cache Effects in Measurement-Based Timing Analysis

Enrique Díaz[1,2], Jaume Abella[2], Enrico Mezzetti[2], Irune Agirre[3], Mikel Azkarate-Askasua[3], Tullio Vardanega[4], and Francisco J. Cazorla[2,5]

1 **Universitat Politècnica de Catalunya, Spain**
2 **Barcelona Supercomputing Center, Spain**
3 **IK4-IKERLAN, Spain**
4 **University of Padova, Italy**
5 **IIIA-CSIC, Spain**

Abstract

Measurement-based timing analysis (MBTA) is often used to determine the timing behaviour of software programs embedded in safety-aware real-time systems deployed in various industrial domains including automotive and railway. MBTA methods rely on some form of instrumentation, either at hardware or software level, of the target program or fragments thereof to collect execution-time measurement data. A known drawback of software-level instrumentation is that instrumentation itself does affect the timing and functional behaviour of a program, resulting in the so-called *probe effect*: leaving the instrumentation code in the final executable can negatively affect average performance and could not be even admissible under stringent industrial qualification and certification standards; removing it before operation jeopardizes the results of timing analysis as the WCET estimates on the instrumented version of the program cannot be valid any more due, for example, to the timing effects incurred by different cache alignments. In this paper, we present a novel approach to mitigate the impact of instrumentation code on cache behaviour by reducing the instrumentation overhead while at the same time preserving and consolidating the results of timing analysis.

1998 ACM Subject Classification [D4.7] Real-time Systems and Embedded Systems

Keywords and phrases WCET, Measurements, Instrumentation overhead

Digital Object Identifier 10.4230/OASIcs.xxx.yyy.p

1 Introduction

Measurement-based timing analysis (MBTA) methods are widely used in application domains such as automotive, railway or space [12]. With MBTA, execution-time measurements of selected fragments of software programs of interest are taken while observing runs of the program on the target processor, performed under stressful conditions. The highest value, usually known as high-water mark time (HWMT), is recorded. The HWMT of all code fragments (the smallest of which is a basic block) in the program are then combined to determine the worst-case execution time (WCET) of the corresponding task. It is worth noting that MBTA works on the premise that the testing conditions are representative of real system operation, so that the HWMT approximates the real WCET.

The places in the code where the required execution-time observations are made are usually referred to as *instrumentation points* (ipoints). MBTA generates a run-time trace that logs which ipoints are traversed at what time during the observation run. Two differing

Conference/workshop/symposium title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–10

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

approaches exist to generate time readings at ipoints: the generation of time traces via hardware methods has been made possible with the advent of processors with advanced debug capabilities that do not affect program timing behaviour. Hardware instrumentation ideally provides transparent generation of timing traces – assuming that collected traces can be output in a fast-enough and equally transparent way so that no trace data are lost because of overfull buffers and no explicit program action is to be taken. However, this support is not present in all processors candidate for use in real-time systems. As a result, in the general case, software-level solutions are adopted where some form of instrumentation code is required in the program to generate timing information.

In contrast with its hardware-based counterpart software instrumentation is highly intrusive, especially on the temporal behaviour. In the presence of caches, instrumentation code can generate unwanted timing effects far beyond the intrinsic timing penalty of the additional instrumentation instructions, that would not have been observed in the un-instrumented (original) program. The user thus faces the dilemma of whether to remove the instrumentation code from or leave it in the final program.

- Removing the instrumentation code from the final executable raises the question of how the execution-time observations taken with the instrumented code correlate with the timing behaviour of the un-instrumented program. In fact, both functional and timing verification would have been conducted on a different software artifact and strong additional argument must be provided for the analysis result to hold.
- Leaving instrumentation code in the final executable spares the burden (for cost and complexity) of demonstrating equivalent functionality as WCET estimation is performed on the executable that will be deployed in the operational system. However, certification and qualification practices may simply not accept the presence of this instrumenter-added code in the executable. As an immediate effect, leaving instrumentation code in a program is likely to worsen memory footprint and average performance. Further, some memory-mapped I/O space – where execution-time readings might be kept – may be unnecessarily wasted.

In fact, both leaving and removing instrumentation code may have disruptive effects on the certification process. While there have been several methods to minimise the number of instrumentation points, without losing too much precision [19], in this paper we show that even a single instrumentation point can lead to significantly different timing behaviour between the original (un-instrumented) program and the instrumented version.

We present a novel technique that strikes an optimal balance between the two approaches, basing on the concept of *functionally-neutral* program, that is used at system operation. From the original program (oprog), fnprog is generated by inserting *nop* instructions at desired instrumentation points. The neutral nature of nop instructions and the fact that they can neither generate interrupts nor have input or output dependences simplifies certification/qualification argumentation. For the purposes of timing analysis, nop instructions are replaced by actual instrumentation operations, resulting in an instrumented program (iprog). The number of nops inserted per ipoint in fnprog is carefully selected so that the cache alignment of code in fnprog and iprog stays unchanged. This prevents any unwanted impact on timing behaviour that may stem from variations in cache alignment. The increase in terms of memory footprint and execution time of fnprog as compared to oprog, instead, depends on the program and the number of ipoints inserted.

We have applied this method within the scope of measurement-based probabilistic timing analysis (MBPTA) [2]. Besides its evident benefits on the fronts of qualification and certification alike, our approach significantly reduces the impact of software-level instrumentation.

In quantitative terms, assuming basic block level instrumentation with two instructions per ipoint, the average degradation we observed in the computed execution-time bounds was 9.3% for EEMBC benchmark programs and 8.7% for a railway case-study application.

2 Background and Problem Statement

MBTA differentiates between the *analysis phase*, when verification of timing behaviour takes place, and the *operation phase*, when the system is deployed into operation. MBTA computes WCET estimates with execution-time measurements taken at analysis time, on the condition that the corresponding bound holds at operation. This requires the user to define test scenarios that trigger worst-case conditions that can occur during system operation. (We intentionally omit discussing here how difficult, if at all possible, that is for the user.)

MBPTA [2] is a variant of MBTA that derives probabilistic WCET (pWCET) estimates – an execution-time distribution expressing the maximum probability that upper bounds the residual risk that *one instance* of the program may exceed a given execution-time threshold. MBPTA handles the sources of execution-time variability caused by hardware and software effects by ensuring that the jitter they cause at analysis matches or upper bounds that which occurs during operation [8]. Cache memories, whose behaviour cannot be treated that way, are time randomised instead [7], so that their impact on execution time can be studied probabilistically for both analysis and operation conditions and the former can be made to upper bound the latter. MBPTA employs extreme value theory [9] (EVT) to model the distribution of extreme (worst-case) execution times (pWCET), considering the timing impact of randomised hardware resources both individually and collectively.

2.1 Problem Statement

MB(P)TA generates a trace that records which and when *ipoints* are traversed during execution. Every such trace is a sequence of <*ipointid*, *timestamp*> pairs. Two main steps take place in the generation of ipoint traces.

- **Generation**. Modern hardware comprises advanced debug interfaces that trigger specific actions when certain opcodes are executed. For example, debug hardware can be used, on every branch instruction taken by program execution, to collect a <branch (instruction) address, execution cycle> pair of trace data. The type of instruction (event) to trace and the action to perform when such an instruction is hit can be programmed through a provided interface, e.g. Nexus or GRMON for the LEON processor family [16]. Debug hardware of that kind is not present in all processors used in real-time systems. In general, therefore, some form of software instrumentation is needed. To that end, specific instrumentation instructions are inserted at the desired granularity of information in the execution context of the program of interest. For instance, these instructions may read the time-base register and output its contents to a specific I/O address.
- **Collection**. Once instrumentation (in either hardware or software) is in place, the execution of the unit of analysis on the target processor results in a set of timestamps and events. This list is output outside of the processor and dispatched to some off-line analysis tool. The capture and offloading process can be the source of further interference. On-chip debug hardware usually includes off-band buses so that the transfer of <timestamp, event> pairs does not affect the execution of the unit of analysis. Depending on the speed of the CPU and the quantity of ipoints, the volume of timestamped data can be high. This can be managed by outputting the data to high-speed ports. Specialised hardware to

process the trace at very high speed has been proposed [13] and exists commercially [15], which prevents loss of information or stalls of execution.

The generation and collection of this trace may interfere with the system's timing and perturb its behaviour. This phenomenon is known as the *probe effect*, which causes the instrumented program to have register and cache usage profiles that differ from those of the original (un-instrumented) program. We categorise those effects as follows:

- **Direct impact** stems from the execution latency of fetching and executing instrumentation code (icode). The icode usually involves reading internal processor registers, e.g., the time register, ($\Delta_{icode}^{core-exec}$) and outputting the readout to a specific memory address. If this information is transferred via buses or I/O controllers used by the application under analysis, this action is bound to interfere with application's timing behaviour. Furthermore, if ipoint information is cached, this can also cause significant effects ($\Delta_{icode}^{chip-exec}$). Not all processors are capable of tracing and dumping data implicitly, without using ipoints. Fast debug links and I/O ports (e.g., GPIO, Ethernet) are often available to dump trace data transparently to the application as long as the ipoints are conveniently placed in the program being run. Trace data can be either processed in real-time of stored for off-line processing.
- **Indirect impact** of the icode arises from the change in the layout of program code in memory. When an ipoint is inserted in the program, it shifts the position in memory of the subsequent instructions and hence also their address and possibly its cache set layout. This may make it considerably difficult for the user to provide evidence that the execution-time measurements obtained with the instrumented binary (iprog) are larger or smaller than those obtained with oprog. This in turn causes the pWCET estimates obtained for iprog to not safely upper bound oprog's execution time.

The execution time of the original program ($EToprog$) is affected in a direct manner by icode in the instrumented program ($ETiprog$) as shown in the second addend of Equation 1.

$$ETiprog = EToprog + \left(\Delta_{icode}^{core-exec} + \Delta_{icode}^{chip-exec} + \Delta_{icode}^{collect}\right) + \Delta_{icode}^{malign} \quad (1)$$

Without loss of generality, we assume that both trace-generation impact, other than executing icode at the core level, and trace-collection overhead are null ($\Delta_{icode}^{collect} = \Delta_{icode}^{chip-exec} = 0$), and instrumentation overheads only arise from the core execution of ipoints included in the unit of analysis ($\Delta_{icode}^{core-exec} \neq 0$), which however creates a constant execution time overhead. The actual problem stems from the fact that the insertion of the icode may change the memory layout of the original code in oprog. Unlike the direct impact of icode that is bound to be a positive value, misalignment impact (Δ_{icode}^{malign}) can be either positive or negative. This may cause iprog to run either faster or slower than oprog. This may lead to the situation in which, despite the direct impact caused by the insertion of icode, the execution-time distribution and pWCET estimate for iprog are even smaller than those for obtained for oprog, i.e. $\Delta_{icode}^{core-exec} + \Delta_{icode}^{chip-exec} + \Delta_{icode}^{collect} < \Delta_{icode}^{malign}$ so $\Delta_{icode}^{core-exec} < \Delta_{icode}^{malign}$.

2.2 Illustrative Example

In this section we use a small example to demonstrate how instrumentation code can create unexpected timing behaviour in which the instrumented program yields an execution-time profile that does not upper bound the profile of the un-instrumented program.

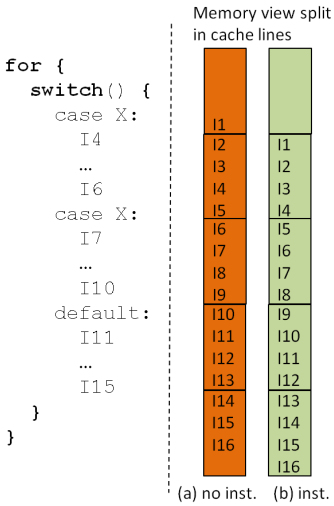


Figure 1 Memory impact

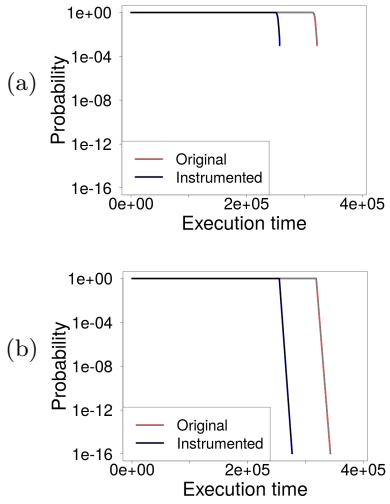


Figure 2 Execution-time impact

Let us consider the code fragment shown in the left part of Figure 1. This code comprises a `for` structure with a nested `switch` structure. In the example, $I1$, $I2$ and $I3$ (not shown) correspond to loop and switch control instructions; $I4 - I6$ are in the first case of the switch; $I7 - I10$ in the second; and $I11 - I15$ in the third. Further assume that before any instrumentation is applied the code (instruction addresses) is laid out in memory as presented in (a), occupying 5 cache lines. Further assume that a single instrumentation instruction is inserted somewhere before this fragment, causing a shift of one instruction and resulting in the memory layout presented in (b). The body of the loop thus occupies 4 lines.

We executed this code on a light-weight processor simulator comprising a 2-set 2-way instruction cache. Other than the jitter caused by the instruction cache – 4 cycles in case of hit and 94 cycles in case of miss – instructions have a back-end latency of 6 cycles. We assume an arbitrary input vector that causes a different branch of the `switch` to be triggered at each loop iteration. We assume $\Delta_{icode}^{core-exec} = 2$ for the only added ipoint.

In terms of MBPTA, this yields the empirical complementary cumulative distribution functions (ECCDFs) shown in Figure 2(a) from where we see that the execution profile of the un-instrumented code is higher than that of the instrumented code. If we apply MBPTA to those observed execution times we obtain the pWCET estimates shown in Figure 2(b). We observe that both the empirical distribution and the pWCET for the original code are much higher than those for the instrumented code.

Hence, even a single instrumentation instruction can change the cache layout so that the instrumented program has lower execution time than the un-instrumented one.

3 Proposal

The proposal we present in this section aims to help the user be assured that the version of the program used for WCET analysis leads to an execution-time distribution that reliably upper bounds the execution time of the version of the program used during operation. As a secondary goal, we want to reduce $\Delta_{icode}^{core-exec}$ to produce tighter WCET estimates.

Our approach uses three versions of the program of interest, see Figure 3: oprog, the original program, fnprog, its augmented functionally-neutral version, which is used in operation,

and iprog, the instrumented executable, which is used for analysis.

fnprog. This is a version of oprog augmented with *nop* instructions inserted to bound the indirect impact of the icode (Δ_{icode}^{malign}). This modification results in a program with unaltered functional behaviour that can therefore be used in operation. However, an argument (which we denote A1) is needed to show that fnprog provides the same functional output as oprog. Furthermore, we also need to prove that the average performance of fnprog is near and not unacceptably worse than that of oprog.

The nop instructions in fnprog are inserted at all places where ipoints are needed. The number of nop instructions inserted per ipoint is sufficient to ensure that, when the actual instrumentation instructions are inserted in iprog, no cache-line misalignment occurs. Typically, one or two instrumentation instructions per ipoint are sufficient to collect timing information, depending on the actual hardware support and instruction set. Hence, fnprog includes one or two nops at the place of each ipoint. It is worth noting that oprog may already include some nops. For instance, in an architecture with delayed branches (e.g., SparcV8), delayed slots may not be filled with useful instructions or nops, depending on compiler flags or program semantics. Further, some compilers include options that insert nops to enforce memory alignment at the level of function, branches, jumps and loops (e.g. `-falign-functions=n`, `-falign-labels=n`, `-falign-loops=n`, `-falign-jumps=n` in GCC). Those nops are just fine for placing instrumentation code in iprog.

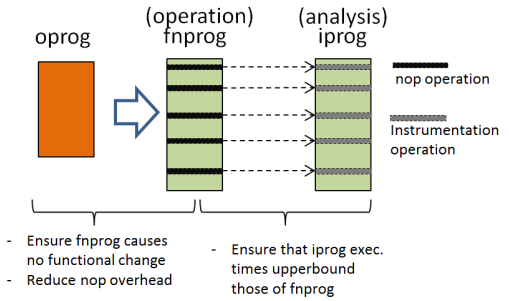


Figure 3 Schematics of the proposed approach

iprog. The execution-time measurement traces needed by MB(P)TA are obtained from a modified version of fnprog. In that version, which we call instrumented binary, iprog, some or even all of the inserted nops are replaced by actual instrumentation instructions impacting execution time. This change is made in a way that causes no code realignment with respect to fnprog which simplifies ensuring that the execution-time traces obtained from iprog can be reliably used to derive a pWCET estimate for fnprog as used in operation. A further argument (A2) is needed to show that the execution-time behaviour of iprog is never less than that of fnprog, so that the execution-time observations taken for iprog can be used to upper bound the WCET of fnprog.

If our approach is adopted, the required nops could be automatically added by the (qualified) compiler. The number of nops and the level they are added could be easily controlled via compiler parameters, e.g. `-fnopcount=n` and `-fnoplevel=basicblock`.

3.1 Functionally-neutral impact of fnprog (A1)

The use of nops simplifies providing arguments that fnprog does not change the functional behaviour of oprog: most of today's Instruction Set Architectures (ISAs) include *nop*-type instructions, whose function is to perform no operation in the processor other than fetch and, possibly, decode, where the instruction is usually stripped from the execution stream.

The main advantage of using nops is that they are functionally neutral: (1) by definition, a nop performs no operation; (2) its execution does not change status flags or any other control registers; (3) a nop generates neither raises interrupts nor exceptions; (4) a nop uses no architectural (programmer accessible) register, which allows inserting nops anywhere in

the code; and (5) a nop has no input and no output (register) dependences. From all these properties it follows that fnprog cannot change the functional behaviour of oprog.

From the average performance standpoint, whose improvement we defined as our second goal, nops usually take a few cycles to execute. In some architectures, the processor may even strip nops out from the pipeline before they reach the execution stage. This is in contrast with actual instrumentation instructions that usually need to access off-core (or off-chip) resources such as I/O ports or trace buffers, thus incurring longer execution times.

3.2 fnprog's WCET is upper bounded by that of iprog (A2)

The difference between fnprog and iprog is that some nops in the former are changed to actual instrumentation instructions in the latter. The number of nops inserted at the place of each ipoint in fnprog is such that *exactly* the same cache-line alignment occurs in fnprog and in iprog. The net result is an increase in the execution time of iprog in comparison to fnprog since the instrumentation code takes longer to execute than nops. The fact that this overhead has an additive nature ipoint by ipoint – in a processor free of timing anomalies – facilitates making an argument about the fact that the resulting execution-time distribution collected with iprog upper bounds that of fnprog. Notably, the execution time of fnprog might be lower than that of oprog, owing to the instruction cache effects described before. However, this does not create any issue since fnprog is the one that will be deployed to operation.

Recent work [5] in the specific context of MBPTA [2] for time-randomised caches [7] helps us contend that the execution time of iprog does indeed upper bound that of fnprog. Let IS_{org} be a sequence of instructions to which we add a set $\mathcal{O}$ of new operations – both acting within core (e.g. add) and on memory – resulting in the extended instruction sequence IS_{ext}.

The authors of [5] show that the probabilistic execution time (pET) of IS_{ext} is higher than the pET of IS_{org}. We say that $pET(ISi) \geq pET(ISj)$ if, for any cut-off probability, the execution time of ISi is higher than or equal to the execution time of ISj. We contend that this argument can be made for standard MBTA, not just MBPTA, but we leave the corresponding demonstration as future work.

In addition to cache-related icode variability – which we attack in this paper – measurement-based timing analysis needs to handle timing anomalies if they can happen.

4 Experimental Results

So far we have provided arguments in support to the fact that our functionally-neutral approach provides a reasonable solution to the software instrumentation problem from the qualification and certification standpoint as fnprog can be safely deployed instead of the oprog and iprog. In our experimental evaluation we aimed at providing evidence that the fnprog always exhibits pWCET that tightly upper bounds oprog and is always lower than iprog. For our evaluation we use the well-known EEMBC automotive benchmark suite [11]. In particular, we use the following benchmark programs: `a2time` (A2), `aifftr` (AI), `aifirf` (AF), `aiifft` (AT), `bitmnp` (BI),`cacheb` (CB), `canrdr` (CN), `idctrn` (ID), `iirflt` (II), `matrix` (MA). We also use a railway case-study application that is part of the European Railway Traffic Management System (ERTMS) [4] initiative that seeks to define a unique European train signalling standard. Our focus is on the on-board unit of the ERTMS, called European Train Control System (ETCS). We consider 10 different input sets (`S0` to `S9`). For the iprog we assume 1 and 2 instrumentation instructions per ipoint.

We focus on the pessimistic scenario where ipoints are added at basic block boundary – the smallest granularity of instrumentation in general. While in this case the instrumentation

impact is high, we have seen that a single instrumentation point can cause unwanted cache effects between the instrumented and non-instrumented code. In the presence of techniques reducing the number of ipoints (see Related Work Section), the overhead introduced by our approach will naturally reduce. We use a cycle-accurate processor simulator that implements 4KB L1 instruction- and data-caches, which comprise 128 sets and 2 ways each. Both caches implement random placement and replacement [7]. The access latency to the L1 caches is 1 cycle and that to main memory is 28 cycles. For the instrumentation instructions, we assume they have the cost of 2 cycles.

4.1 Execution Time and Code Size Increase Due to Instrumentation

Table 1 shows the increase in code size and execution time incurred by fnprog and iprog respectively. We can see that fnprog incurs moderate overhead in footprint – between 7% and 14% – when one or two nop operations are added per ipoint per basic block. In terms of execution time, the impact is 5% and 10% when 1 and 2 instructions are added respectively. Meanwhile iprog incurs higher execution time overheads: 9% and 19% respectively.

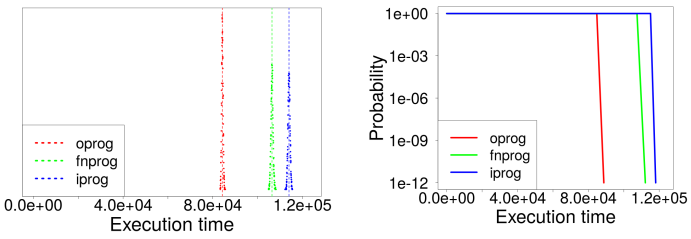
no. instruct.	Code Size	execution time fnprog	execution time iprog
1 inst./2 inst.	6.87%/13.74%	4.35%/9.28%	8.33%/17.54%

Table 1 Average increase in code size and execution time of fnprog and iprog normalized to oprog

Per-benchmark results (not depicted for space constraints), show a clear relation between program's average basic block size the the code size impact of nops. In terms of time, in addition to average basic block size the average duration of each instruction (program's CPI) determines the execution overhead of nops.

4.2 pWCET estimates

For the case of 2 instrumentation operations per ipoint, we compare the pWCET estimates obtained from the execution-time measurements from oprog, fnprog and iprog. We collected 1,000 runs for each version, which have sufficed to pass the MBPTA convergence criteria [2]. Figure 4 shows the execution-time histogram, the EC-CDF and the resulting pWCET estimate for `a2time` benchmark. We observe how fnprog is close to oprog's. Further, changing nops by instrumentation instructions causes iprog's execution-time to upper bound fnprog's.



(a) Histogram (b) ECCDF

Figure 4 Histogram and MBPTA projection for `a2time`

For the other benchmarks, Table 2 summarises the difference observed for the three versions of the program for a cut-off probability of 10^{-12} per run. As shown, fnprog is relatively tight with respect to oprog. The only exception is `A2`, which includes many basic blocks, and therefore takes a higher density of nops.

prog.	A2	AI	AF	AT	BF	BI	CB	CN	ID	II	MA
fnprog	26.4%	3.8%	11.6%	7.1%	11.5%	11.9%	2.1%	14.1%	12.3%	11.9%	6.4%
iprog	33.0%	9.3%	22.1%	12.4%	22.0%	16.8%	4.0%	27.1%	23.3%	21.6%	12.7%

Table 2 pWCET estimates for 10^{-12} for fnprog and iprog normalized to oprog

4.3 Railway case study

Table 3 reports analogous results for the railway case study for 2 instrumentation instructions per ipoint. The results are even tighter on average than those we obtained for the EEMBC benchmarks, with an average increase in pWCET estimates of 8.7% and 11.9% for fnprog and iprog respectively. The code size increase observed is 12%, which is also less than the average incurred with the EEMBC benchmarks. Overall, our proposed solution provably abates the negative misalignment effects of icode, for a small cost in execution time in general.

prog.	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9
fnprog	8.4%	7.6%	9.5%	9.1%	8.1%	9.5%	9.4%	8.3%	8.6%	8.9%
iprog	11.5%	10.4%	12.1%	12.3%	12.2%	13.3%	12.4%	11.9%	12.2%	11.1%

Table 3 pWCET estimates for 10^{-12} for *fnprog* and *iprog* w.r.t to that for *oprog*

5 Related Work

The literature on measurement-based timing analysis is abundant [19][10][18][19][1][12].

In [10], the authors discuss the pros and cons of several ways to collect execution traces and how the frequency of ipoints results in light-weight or heavy-weight instrumentation.

Measurements can be taken (i.e., ipoints can be placed) at program boundaries. However, hybrid mechanisms exist to identify smaller program parts. The particular segments used are extracted from an analysis of the Control-Flow Graph [6][1]. The segments (partitions) are chosen to facilitate the derivation of a WCET by composing the WCET of each segment, facilitating measurements [19][1] or reducing the number of ipoints. Some of these techniques also work on an automatic generation of input data [6][19][17]. In [6], the authors decompose execution paths into sub-paths and then use formal methods to derive the required test data (in an automatic manner) and measure the execution time of the sub-paths.

In [14], the authors propose the concept of context-sensitive traces to capture the impact of execution history on the precision of measurement-based execution-time estimates. This is done using the concept of "call string" that defines the sequence (of the last k calls) that keeps information similar to the call stack. For trace collection, some research approaches developed an FPGA board to transfer information from the target to the host to increase trace processing capabilities [13]. Other approaches propose on-line aggregation of timing data removing the need for collecting and post-processing long traces [3].

6 Conclusions and Future Work

We have presented a new approach to mitigate the impact of instrumentation code to prevent cache misalignments from occurring between the instrumented and un-instrumented versions

of the program under analysis, while incurring low overhead in terms of execution time. In particular, we build upon the use of *functionally-neutral* operations such as nops to create a program version to be deployed that is functionally equivalent to the original program, and has a provable lower execution time than the instrumented version.

As part of our future work we plan to evaluate the fnprog approach in a real hardware platform and a commercial timing analysis tool. We also plan to extend the argumentation about the timing impact of iprog w.r.t to fnprog to non probabilistic timing analysis.

References

1 A. Betts et al. Hybrid measurement-based WCET analysis at the source level using object-level traces. In *WCET Analysis Workshop*, 2010.
2 L. Cucu-Grosjean et al. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS*, 2012.
3 B. Dreyer et al. Precise continuous non-intrusive measurement-based execution time estimation. In *WCET Analyis Workshop*, 2015.
4 ERA (European Railway Agency). ERTMS - Set of specifications - 2 (ETCS baseline 3 and GSM-R baseline 0), 2014.
5 L. Kosmidis et al. PUB: Path upper-bounding for measurement-based probabilistic timing analysis. In *ECRTS*, 2014.
6 R. Kirner et al. Measurement-based worst-case execution time analysis using automatic test-data generation. In *SEUS Workshop*, 2004.
7 L. Kosmidis et al. A cache design for probabilistically analysable real-time systems. In *DATE*, 2013.
8 L. Kosmidis et al. Probabilistic timing analysis and its impact on processor architecture. In *DSD*, 2014.
9 S. Kotz et al. *Extreme value distributions: theory and applications*. World Scientific, 2000.
10 S.M. Petters. Comparison of trace generation methods for measurement based WCET analysis. In *WCET Analysis Workshop*, 2003.
11 J. Poovey. *Characterization of the EEMBC Benchmark Suite*. NCSU, 2007.
12 R. Wilhelm et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7:1–53, May 2008.
13 B. Rieder et al. Using a runtime measurement device with measurement-based WCET analysis. In *IESS*, 2007.
14 S. Stattelmann and F. Martin. On the use of context information for precise measurement-based execution time estimation. In *WCET Analysis Workshop*, 2010.
15 `https://www.rapitasystems.com/products/rtbx`. *RTBx*. Rapita Systems.
16 `http://www.gaisler.com/index.php/products/debug-tools/grmon`. *Cobham Gaisler*. GRMON.
17 I. Wenzel et al. Automatic timing model generation by CFG partitioning and model checking. In *DATE*, 2005.
18 I. Wenzel et al. Measurement-based worst-case execution time analysis. In *SEUS Workshop*, 2005.
19 I. Wenzel et al. Measurement-based timing analysis. In *ISOLA*, 2008.

[1] The research leading to these results has received funding from the European Community's FP7 [FP7/2007-2013] under the PROXIMA Project (www.proxima-project.eu), grant agreement no 611085. This work has also been partially supported by the Spanish Ministry of Science and Innovation (grant TIN2015-65316-P) and the HiPEAC Network of Excellence. Jaume Abella has been partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal fellowship RYC-2013-14717.