UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Master in Innovation and Research in Informatics
High Performance Computing

Software Support to Strengthen Measurement-Based Timing Analysis

Author Enrique Díaz Roque[⋆]
Advisors Francisco J. Cazorla Almeida[⋆†]
Jaume Abella Ferrer[⋆]
Tutor Mateo Valero Cortés[⋆‡]

⋆ Barcelona Supercomputing Center (BSC)
† Spanish National Research Council (IIIA-CSIC)
‡ Computer Architecture Department (DAC-UPC)

July 2017

Acknowledgements

I would like to express my gratitude to my advisors, Fran and Jaume, for their guidance, their experience, and the opportunity given. I would also like to thank Mikel, Carles, Pedro, Gabriel, Enrico and Leonidas for the help given. Furthermore, I would like to thank all the CAOS research group at the Barcelona Supercomputing Center for their help and support through these two years.

I would also like to express my gratitude to my parents for their help and to my girlfriend María for her always kind support. Furthermore, I would like to thank Adrián and Calvin for the experiences we lived during the Master.

The research leading to these results has received funding from the European Community's FP7 [FP7/2007-2013] under the PROXIMA Project (http://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.

Abstract

In critical domains, the advent of high-performance (complex) hardware, used to provide the rising levels of guaranteed performance, complicates providing evidence of reliable software execution specially on aspects related to the timing dimension. Caches and multicores are two of the hardware features that have the potential to significantly reduce worst-case execution time (WCET) estimates, yet they pose new challenges on current-practice measurement-based timing analysis (MBTA) approaches.

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. Instrumentation does affect the timing and functional behavior 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.

Measurement-Based Probabilistic Timing Analysis (MBPTA) is a variant of MBTA that aims at increasing the confidence on WCET estimates. MBPTA aims at relieving the user from controlling hardware sources of jitter. MBPTA implicitly controls the impact of jittery resources on measurements captured at analysis. Some hardware resources are randomized so that their execution times at analysis vary according to a probabilistic execution time distribution that can be used to upperbound the latencies during operation and give a WCET prediction with a certain probability.

In this Master Thesis we present our approach to mitigate the impact of instrumentation code on cache behavior by reducing the instrumentation overhead while at the same time preserving and consolidating the results of timing analysis. We further propose a technique for multilevel-cache multicores that combines deterministic and probabilistic jitter-bounding approaches to reliably handle variability in execution time generated by caches and the contention in accessing shared hardware resources.

Contents

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Contributions 3
  - 1.2.1 Contribution 1 3
  - 1.2.2 Contribution 2 4
- 1.3 Structure of the Thesis 5

2 Software-Instrumentation Cache Effects **6**
- 2.1 Introduction 6
- 2.2 Background 8
- 2.3 Problem statement 9
  - 2.3.1 Probe effect 10
- 2.4 Illustrative example 11
- 2.5 Proposal 12
  - 2.5.1 Functionally-neutral program (fnprog) 13
    - 2.5.1.1 Impact argument 14
  - 2.5.2 Instrumented program (iprog) 15
    - 2.5.2.1 WCET upper bounding 15
- 2.6 Experimental results 16
  - 2.6.1 Experiment setup 16
  - 2.6.2 EEMBC 17
  - 2.6.3 Case study 18
- 2.7 Related work 19
- 2.8 Conclusions and future work 19

3 Multicore and Caches **21**
- 3.1 Introduction 21
- 3.2 Background 23
  - 3.2.1 MBPTA 23
  - 3.2.2 Software Randomization 25
- 3.3 Platform 25
- 3.4 Handling multicore contention and cache jitter 26
  - 3.4.1 Goals and challenges 27

3.4.2 Overall process . . . 27
3.4.3 Detailed explanation . . . 28
3.4.4 Request Characteristics . . . 29
3.5 fTC contention model . . . 30
3.6 pTC contention model . . . 30
3.6.1 Bounding contention . . . 32
3.6.2 Other considerations . . . 33
3.7 Experimental results . . . 34
3.7.1 Hardware Setup . . . 34
3.7.2 MBPTA Setup . . . 34
3.7.3 Synthetic application . . . 34
3.7.4 EEMBC . . . 37
3.8 Related work . . . 38
3.9 Conclusions . . . 39

4 Summary . . . 40

Bibliography . . . 41

List of Figures

2.1 Memory layout impact 12
2.2 Execution-time impact 13
2.3 Schematics of the proposed approach 13
2.4 Histogram and MBPTA projection for `a2time` 18

3.1 pWCET example 24
3.2 MBPTA steps 24
3.3 Reference architecture 26
3.4 Schematic view of the proposed pTC contention model 28
3.5 Events and PMCs 31
3.6 Pairing steps 31
3.7 Results of TASA and MBPTA on a single-core setup. 35
3.8 Execution time when $c(\tau_a)$ create clean/dirty misses and fTC 36
3.9 Result of the pTC under different load scenarios generated by the contenders 36
3.10 Results of EEMBCs against 3 copies of themselves 38

List of Tables

2.1 pWCET estimates for 10^{-12} for fnprog and iprog normalized to oprog 17
2.2 Average increase in code size and execution time of fnprog and iprog normalized to oprog 17
2.3 pWCET estimates for 10^{-12} for *fnprog* and *iprog* w.r.t to that for *oprog* 18

3.1 Request types and their latency in our reference board 29
3.2 PMCs available in the reference processor 29

1. Introduction

This chapter introduces the work presented in this thesis. We start in Section 1.1 with the motivation behind this work to the presented problems. We then continue in Section 1.2 by presenting the contributions of this work. At the end of this chapter, Section 1.3, we describe the structure of the rest of the document.

1.1 Motivation

Real-Time Embedded Systems (RTES) are those that have tasks that are expected to be executed within a given time budget in order to guarantee responsiveness. They are said to be embedded as they form part of a larger system (e.g. a car), usually acting as the controller. The RTES industry nowadays represents an important part of the computer market and it is expected to drive the global chip demand in the next years [1].

RTES programs have a set of deadlines to fulfill. These deadlines are time limits of the application running on the embedded system. If the duration of a software activity (usually implemented as a task) exceeds its timing budget, we say that it has missed its deadline. Depending on how critical is missing a deadline, we can grossly classify real-time systems into two categories:

- Soft Real-Time Embedded Systems. While missing a deadline is not desirable, the system can continue with its operation as long as the deadline miss rate is below a threshold. An example of this type of systems is a video decoder. If the processing of a frame is not completed on time – it did not meet the deadline – the result is not catastrophic, furthermore, it may pass unnoticed by the end user.
- Hard Real-Time Embedded Systems. Also known as Critical Real-Time

Embedded Systems (CRTES). Missing deadlines may affect system's safety. Examples of this type of systems include those on-board in cars, airplanes, or satellites that may cause fatalities if the system is unresponsive beyond the time stipulated during the system design phase.

Since recently industrial solutions have been based on adding more than one embedded processor in order to provide increased performance. For instance, a modern car may contain up to 70 Electronic Control Units (ECUs) and this trend is predicted to grow rapidly as automation expands in the automotive industry [2]. As industry keep implementing new features, the number of embedded devices on-board increases. The number of devices to maintain and test in the future will be unsustainable, furthermore, it will complicate design itself. The introduction of high-performance processors in the embedded market is then a need. Deep memory hierarchies and multicore processors will make the difference and will help to reduce money spent in design, maintenance and test. However, the introduction of multicores and high-performance caches in the RTES market is not trivial. We have to pay close attention on how timing budgets are analyzed.

Timing analysis in real-time systems is needed in order to obtain system's execution time profiles so that an upper bound of its execution time can be derived. At task level, this upper bound ideally covers what we know as the worst-case execution time (WCET). WCET estimates for each task are used in conjunction with system's specifications (schedule) so we know ahead if the programs are going to fulfill their assigned time budget. It is worth noting that the WCET estimate derived from timing analysis techniques needs to be (1) reliable, so it trustworthily upper bounds all possible execution times and (2) tight, so it should be close to the actual WCET to avoid over estimation, which leads to a waste of resources [3]. There are two main timing analysis approaches in the literature:

- Static Timing Analysis [3]. STA is based on mathematical models of the processor architecture. The model must take into account low-level details of the processor, such as pipeline stages, latencies,... STA is becoming increasingly difficult as new powerful (and complex) processors are released in the market. It is also noted that manufacturers do not provide many implementation details, especially those related to the timing of the processor. All these problems translate into pessimistic assumptions to cover the unknown, resulting in a pessimistic WCET estimate [4].
- Measurement-Based Timing Analysis. This methods estimates the WCET based on execution time observations. Several runs are required, varying the possible inputs of the program in order to cover the timing space domain. Time

observations are used to feed statistical methods to derive a WCET estimate. The conditions of the system at analysis time must hold at operation time. All events that can cause significant impact in the execution time at operation must be captured during the analysis [4].

In this work we build on MBPTA [5], a probabilistic variant of measurement-based timing analysis. MBPTA targets to alleviate the problem of capturing events that can impact the execution time at operation. Extreme Value Theory is applied in MBPTA, a statistical method that builds upon the complementary cumulative distribution function (CCDF) of the observed execution times to project a probability distribution. Given a threshold, e.g. target probability of 10^{-12} in the EVT projection, we can then infer a probabilistic WCET (pWCET) that can only be exceeded with a probability below such threshold. In Sections 2.2 and 3.2 we review the concept and explain further details.

1.2 Contributions

In this section we describe the main contributions of this work.

1.2.1 Contribution 1

MBTA in general – and MBPTA in particular – requires instrumenting the program in the analysis phase to extract information (traces) from its execution on the target board. This information includes execution times and other events such as cache miss counts.

The first contribution focuses on the negative effect that software-instrumentation can introduce on measurement-based timing analysis. In particular, we show how instrumented code can change program execution time behavior due to the *probe-effect*. We also show how we can proceed in order to obtain a reliable timing bound.

- We introduce and analyze the challenge of instrumenting a program in order to generate valid timing analysis.

- We illustrate the problem with an example where software-instrumentation interferes optimistically leading to potentially unreliable worst-case execution time bounds.
- We propose a solution for the probe effect caused by software-instrumentation. We implemented our solution and collected information about its impact in representative benchmarks and an industrial case study.

Our results show how our proposal to mitigate the impact of instrumentation code generates valid analysis to infer worst-case execution time estimates while incurring low overhead in terms of execution time.

As a result of this work we have published the following paper: *Enrique Díaz, Jaume Abella, Enrico Mezzetti, Irune Agirre, Mikel Azkarate-Askasua, Tullio Vardanega, and Francisco J. Cazorla, "Mitigating Software-Instrumentation Cache Effects in Measurement-Based Timing Analysis", in 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016), Toulouse (France), July 2016.*

1.2.2 Contribution 2

We propose a method to model contention on shared hardware resources, called MC2 (multicore and cache).

- We present two contention models to address the increase in execution time due to multicore environments and to address variability introduced by caches. We first introduce a contention model (fTC) that captures all possible contention without knowledge on the contender being executed. We also present a contention model (pTC) that tightens execution time estimates by adding additional knowledge of the contention the task under analysis is going to suffer.
- We applied our methodology to a LEON3 based multicore platform implemented on a FPGA and evaluated our model with synthetic applications resembling an aggressive contender and a set of representative benchmarks.

The results presented show how our models tightly and reliably capture contention of other applications running in our multicore platform.

As a result of this work we have published the following paper: *Enrique Díaz, Mikel Fernández, Leonidas Kosmidis, Enrico Mezzetti, Carles Hernandez, Jaume Abella,*

and Francisco J. Cazorla, "MC2: Multicore and Cache Analysis via Deterministic and Probabilistic Jitter Bounding", in 22nd International Conference on Reliable Software Technologies (Ada-Europe 2017), Vienna (Austria), June 2017.

1.3 Structure of the Thesis

The rest of this document is structured as follows:

- Chapter 2 presents our approach to tackle software-instrumentation cache effects, also called *probe effect*. We make a quick review on measurement-based timing analysis and how software-instrumentation can interfere with the analysis. We also propose a solution to this problem and show results of our approach on a well-known set of benchmarks and an industrial case-study.
- Chapter 3 presents the challenge that multicore embedded systems represent for current measurement-based timing analysis techniques. To be more precise, we explore and model contention on shared hardware resources to enable reliable timing analysis. We also present our results on a real hardware platform using a well-known set of benchmarks.
- Chapter 4 closes the Master Thesis by summarizing our contributions to strengthen measurement-based timing analysis.

2. Software-Instrumentation Cache Effects

In this chapter we address the problematics of software instrumentation and its implication in the cache layout.

2.1 Introduction

Measurement-based timing analysis (MBTA) methods are widely used in application domains such as automotive, railway or space [3]. 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 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 behavior. 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 behavior. 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 minimize the number of instrumentation points, without losing too much precision [6], in this work we show that even a single instrumentation point can lead to significantly different timing behavior 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 behavior 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) [7]. 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.

2.2 Background

MBTA differentiates between the *analysis phase*, when verification of timing behavior 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 [7] 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 [5]. Cache memories, whose behavior cannot be treated that way, are time randomized instead [8], 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 randomized hardware resources both individually and collectively.

2.3 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 [10]. 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. Specialized hardware to process the trace at very high speed has been proposed [11] and exists commercially [12], 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 behavior. 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.

2.3.1 Probe effect

We categorize and analyze the effects of unwanted noise introduced by the instrumented version of the original program. We can classify its impact 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 – timestamp –, ($\Delta_{icode}^{core-exec}$) 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 behavior. 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 (ET_{oprog}) is affected in a direct manner by icode in the instrumented program (ET_{iprog}) as shown in the second addend of Equation 2.1.

$$ET_{iprog} = ET_{oprog} + \left(\Delta_{icode}^{core-exec} + \Delta_{icode}^{chip-exec} + \Delta_{icode}^{collect}\right) + \Delta_{icode}^{malign} \tag{2.1}$$

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 $\Delta_{icode}^{core-exec} < \Delta_{icode}^{malign}$.

2.4 Illustrative example

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

Let us consider the code fragment shown in the left part of Figure 2.1. 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 2.2 (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

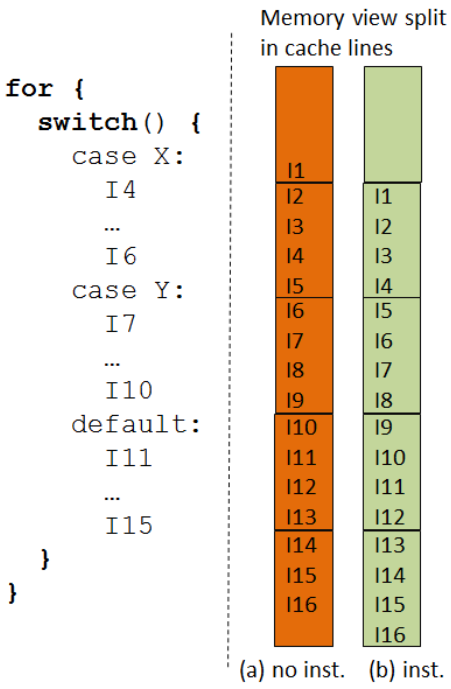


Figure 2.1: Memory layout impact

estimates shown in Figure 2.2 (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.

2.5 Proposal

The proposal presented 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 deployed during operation. As a secondary goal, we want to reduce $\Delta_{icode}^{core-exec}$ to produce tighter WCET estimates.

In our approach we define three versions of the program of interest (see Figure 2.3):

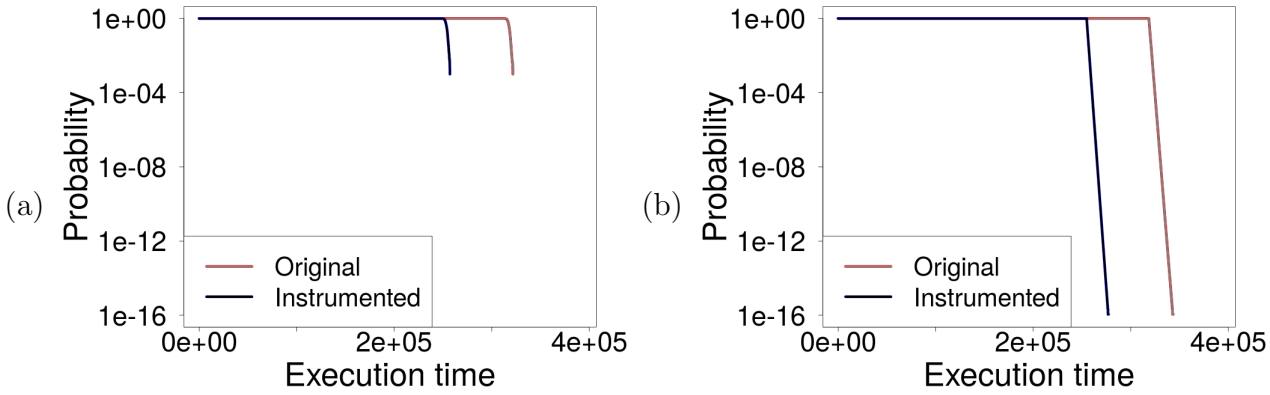


Figure 2.2: Execution-time impact

1. oprog, the original program.
2. fnprog, augmented functionally-neutral version of the original program, which is used in operation.
3. iprog, the instrumented version of the original program, which is used for analysis.

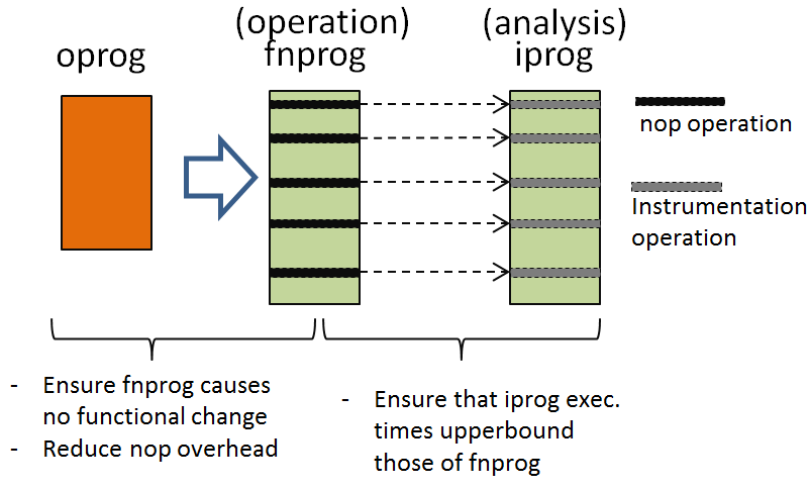


Figure 2.3: Schematics of the proposed approach

2.5.1 Functionally-neutral program (fnprog)

The functionally-neutral program is a version of oprog enlarged with *nop* instructions inserted to bound the indirect impact of the instrumentation code, icode

(Δ_{icode}^{malign}). Of course, this modification results in a program with unaltered functional behavior that can therefore be used in operation.

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.

However, an argument 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.

2.5.1.1 Impact argument

The use of *nops* simplifies providing arguments that fnprog does not change the functional behavior 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.
5. A *nop* has no input and no output (register) dependences.

From all these properties we can state that fnprog cannot change the functional behavior 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.

2.5.2 Instrumented program (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.

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`.

A further argument is needed to show that the execution-time behavior 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.

2.5.2.1 WCET upper bounding

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 [13] in the specific context of MBPTA [7] for time-randomized caches [8] 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 [13] 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 instrumentation code variability – which we attack in this chapter – measurement-based timing analysis needs to handle timing anomalies if they can happen.

2.6 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 focus on providing evidence that the fnprog always exhibits pWCET that tightly upper bounds oprog and is always lower than iprog.

2.6.1 Experiment setup

To perform our experiments we use a cycle-accurate processor trace-driven simulator that implements 4KB L1 instruction and data caches, which comprise 128 sets and 2 ways each. Both caches implement random placement and replacement [8]. 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. This technical specifications were extracted from a real implementation of a LEON3 processor [14].

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, the overhead introduced by our approach will naturally reduce.

2.6.2 EEMBC

We start our evaluation with the well-known EEMBC automotive benchmark suite [15]. 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).

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 [7].

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%

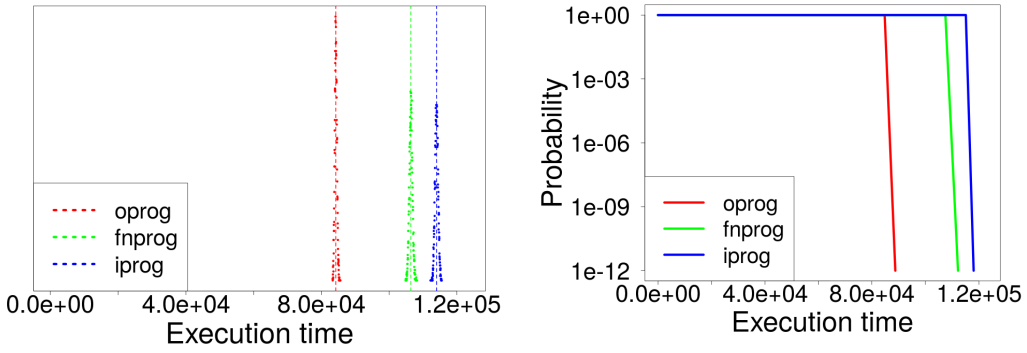
Table 2.1: pWCET estimates for 10^{-12} for fnprog and iprog normalized to oprog

For the other benchmarks, Table 2.1 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 we can observe is A2, which includes many basic blocks, and therefore takes a higher density of nops. In Table 2.2 we show the average code size increase and execution time overhead.

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

Table 2.2: Average increase in code size and execution time of fnprog and iprog normalized to oprog

Figure 2.4 shows the execution-time histogram 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) pWCET estimate

Figure 2.4: Histogram and MBPTA projection for `a2time`

2.6.3 Case study

We also present a railway case study application that is part of the European Railway Traffic Management System (ERTMS) [16] initiative that seeks to define a unique European train signaling standard. To be more specific, our focus is on the on-board unit of the ERTMS, called European Train Control System (ETCS).

We performed simulations on 10 different input sets (`S0` to `S9`). For the iprog we assume 2 instrumentation instructions per ipoint.

Table 2.3 reports analogous results for the railway case study for 2 instrumentation instructions per ipoint. We can observe that 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 average code size increase observed is 12%, which is also less than the average incurred with the EEMBC benchmarks.

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

Overall, our proposed solution provably abates the negative misalignment effects of icode, for a small cost in execution time in general.

2.7 Related work

The literature on measurement-based timing analysis is abundant [3, 6, 17, 18, 19].

In [17], 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 [19,20]. The segments (partitions) are chosen to facilitate the derivation of a WCET by composing the WCET of each segment, facilitating measurements [6,19] or reducing the number of ipoints. Some of these techniques also work on an automatic generation of input data [6, 20, 21]. In [20], 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 [22], 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 [11]. Other approaches propose on-line aggregation of timing data removing the need for collecting and post-processing long traces [23].

2.8 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.

3. Multicore and Caches

In this chapter we propose the *MC2* (multicore and cache) MBPTA approach for the analysis of a Commercial-Off-The-Shelf (COTS) multicore processor equipped with multilevel-caches. Our approach factors in both cache jitter and multicore contention jitter.

3.1 Introduction

Computing power needs are steadily increasing in the critical real-time embedded domains, fuelled by the complexity and sheer amount of data a modern on-board software is expected to handle [24, 25, 26]. At hardware level, while high-performance features, such as caches and multicore processors, provide the demanded performance, they also bring about hard-to-model *jitter* (variability) in execution time, which complicates timing validation and verification. This has resulted in an increased attention on timing in safety standards (e.g., ISO26262 [27] in automotive) and support documents (e.g., CAST32-A [28] in aerospace).

MBTA is the dominant timing analysis approach in most real-time domains [3]. MBTA aims at deriving a worst-case execution time (WCET) estimate that holds for the program during *system operation* from the execution time measurements captured during the tests executed at various stages in the *analysis phase*. The quality of the derived WCET estimates lies on the user's ability to design stressful test scenarios (conditions) that are presumably close to the worst-case conditions that can arise during system operation. The degree of control available to the user, while adequate on simple processor designs, diminishes with the inclusion of complex hardware that challenges: (i) designing worst-case scenarios, e.g., identifying the memory object allocation (code and data) that results in cache set mappings with high impact on execution time, and the worst contention scenarios that the application can suffer in a multicore; and (ii) designing experiments in which bad (pathological) behavior

for several resources occurs simultaneously. Overall, despite the user may perform thousands of experiments, there is no guarantee on whether the bad behavior in the sources of jitter (*soj*), like the cache, has been sufficiently captured. This reduces the confidence on the MBTA WCET estimates, which in turn can prevent the use of some high-performance hardware features in critical real-time embedded systems.

Measurement-Based Probabilistic Timing Analysis (MBPTA) [7, 29] is a variant of MBTA that aims at increasing the confidence on WCET estimates. MBPTA, which has been successfully evaluated on several case studies (e.g., [29, 30]), aims at relieving the user from controlling hardware *soj*. Instead, MBPTA makes that their impact on the measurements emerges naturally, reducing user's burden to only controlling the number of runs to perform [5]. To that end, MBPTA implicitly controls the impact of jittery resources on measurements captured at analysis. In particular, some resources are forced to work on their worst latency during analysis (upperbounding), hence ensuring measurements conservatively capture their impact. The latency of other resources is instead randomized so that their execution times at analysis vary according to a probabilistic execution time distribution that can be used to upperbound the latencies during operation.

While hardware designs have been proposed [8, 31] for MBPTA compliance, and some of them have hit pre-silicon (RTL) readiness level [31], analyzing MBPTA applicability on COTS multicore processors is fundamental to favor a fast and widespread adoption of MBPTA. MC2 exposes, in a combined MBPTA-compliant manner, the jitter of caches and multicore contention to the execution time measurements taken at analysis. As a result, the WCET estimates MBPTA generates from those measurements upperbound the impact of both resources on program execution time. MC2 combines two techniques that have been classified as MBPTA compliant: software randomization [32] for cache-jitter management, and delay upperbounding for multicore contention management [33]. For the latter, since multicore contention can lead to very pessimistic WCET estimates [34] when contention bounds are provisioned for the worst possible contention, MC2 provides adaptable WCET estimates that depend on contenders' contention.

Our results provides evidence that MC2 effectively captures the impact on execution time - and hence on WCET estimates - of both resources, and provides tight WCET estimates.

3.2 Background

When selecting the timing analysis technique to use, industrial users balance the cost-effectiveness of the technique and the evidence that it can provide to satisfy the level of confidence required by the domain-specific standards [4]. Static timing analysis (STA) methods are sound, though at considerable cost for effort, with the quality of their results depending on whether sufficient and accurate information are available about hardware timing and software flow facts. Measurement based timing analysis (MBTA) techniques are less rigorous than static analysis methods but, in general, are more attractive because of their cost-effectiveness and major affinity with the consolidated industrial practice.

The quality of MBTA's derived WCET estimates relates to the evidence on their coverage of the worst-case conditions. When evidence obtained is sufficient, MBTA can be used for high-integrity software, e.g., DAL-A functions in avionics [35]. In practice, all techniques require user-provided inputs, e.g., worst-case scenarios for measurements for MBTA and hardware timing models for static timing analysis (with hardware documentation potentially being inaccurate or incomplete [4], thus eventually resorting to measurements to reverse engineering the timing model [36]). This makes complex argue about the quality of a WCET figure. In this chapter, we focus on MBTA with the intent to increase the confidence that can be placed on the provided WCET estimates.

3.2.1 MBPTA

MBPTA applies Extreme Value Theory [37] (EVT) on execution time observations from the analysis phase to derive the probabilistic WCET (pWCET) distribution that upperbounds program's execution time during operation. MBPTA requires guaranteeing that the observations obtained at analysis capture those events that can impact execution time at operation, and so pWCET estimates [4]. MBPTA, by deploying EVT (see Figure 3.1), is able to derive the probability that bad behavior of several of the *soj* (whose impact has been captured in the analysis-time runs) are triggered in the same run. Hence, EVT has to be seen as a method to predict pathological combinations of observed events in the analysis-time measurements. In general, EVT cannot predict the appearance of unobserved events since their impact on execution time can be arbitrarily large. To cover this gap, MBPTA builds an argument on representativeness by means of i) either injecting randomization in the timing behavior of certain hardware resources (e.g., caches and buses) so that it is possible to determine the probability of their worst behavior to be captured in

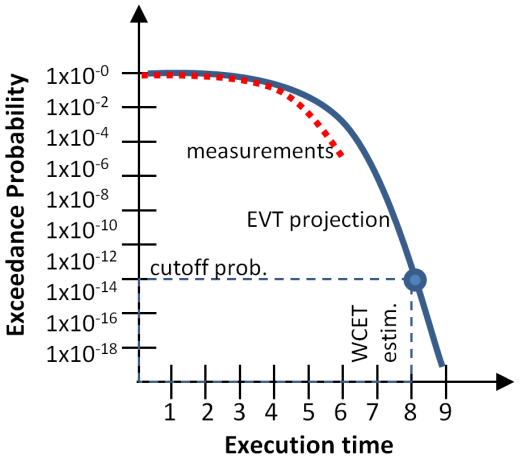


Figure 3.1: pWCET example

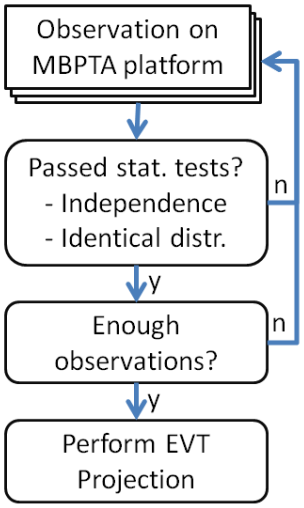


Figure 3.2: MBPTA steps

the analysis-time measurement runs; or ii) making resources to work on their worst latency so the analysis time measurements capture their worst timing behavior.

MBPTA application procedure starts by (1) collecting a set of representative observations, see Figure 3.2. MBPTA then (2) applies some statistical test such as independence and identical distribution tests [7] required for EVT application. Since in a MBPTA-compliant platform these *probabilistic* properties hold by construction, in case *statistical* tests are failed, the user is simply asked for more runs until statistical tests – which are subject to false positives/negatives – are passed. (3) MBPTA checks whether the size of the sample is enough to include all relevant events and ensure certain statistical stability of the results. To that end we use the initial findings in [38] and ask the user for more runs until this condition is satisfied. As final step, (4) MBPTA derives an EVT distribution (pWCET estimate) as shown in Figure 3.1.

Despite time-randomization, programs might exhibit a degenerate distribution of timing, e.g., having a single or very few different execution times. While extremely rare in practice for real-size programs, the lack of jitter would suggest that the maximum observed execution time could be reasonably used as a precise WCET indicator.

3.2.2 Software Randomization

MBPTA handles resources with small jitter (usually in the order of few cycles) by means of upperbounding, i.e., by forcing the resource to operate on its worst latency during analysis time [31]. However, cache resources exhibit high jitter between hit and miss events, especially when these events span across multiple levels of cache. For this reason, timing randomization is used. In particular we use software randomization which, by randomly varying the memory layout between distinct program executions, causes cache events (hits/misses) to have a probabilistic behavior that holds during operation. This allows cache jitter to be properly modeled by MBPTA.

In this work we use our custom implementation of TASA (Toolchain Agnostic Software rAndomization) [39, 40], a static variant of software randomization, applied at source-code level. TASA randomizes the position in memory for any memory object in the software under analysis such as functions, stack frames and global data. Moreover, TASA can randomly affect the internal memory layout of several memory objects such as stack frames and structures.

In general, compilers allocate memory objects in the order they are in the source file. Very few compiler options violate this principle, which can be disabled during compilation with small (if any) impact in the compiler performance [39]. TASA, by randomly rearranging the order of declarations for the corresponding objects in the source file, modifies their relative position in the binary. This, in combination with additional random-sized padding in the form of `nop` instructions or unused data, increases the potential difference among binary layouts. When the binary is loaded to main memory for the program execution, the random binary memory layout translates into random main memory mapping and hence, a random cache layout, i.e., memory objects are allocated in random cache sets.

3.3 Platform

We use a 4-core LEON3 [14] platform implemented on a FPGA. Each LEON3 core implements a 7-stage pipeline: fetch (F), decode (D), register access (RA), execution of non-memory operations (Exe), *dc* access (M), Exceptions (Exc) and write back (WB). Each core comprises first level instruction (*ic*) and data (*dc*) caches, with the *dc* implementing a write-through no write allocate policy, see Figure 3.3. An AMBA AHB bus propagates stores, *dc* misses and *ic* misses to the partitioned L2 cache.

Requests send to the bus are not split. Hence, the bus is locked all the time a

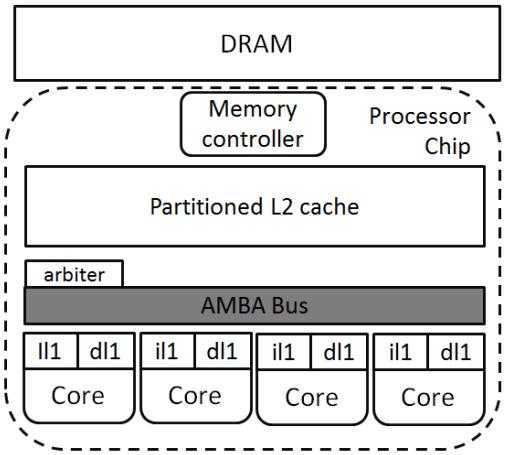


Figure 3.3: Reference architecture

request accesses the L2. If it misses in L2, the bus is locked until the request is solved in main memory and answered back. Requests are arbitrated in the bus using a round-robin arbiter which has been shown to provide time predictability [41]. Hence, our reference architecture comprises two main hardware shared resources, the bus and the memory, with the bus arbiter controlling the contention in both of them.

Our platform also comprises performance monitoring counters (PMCs) from which we track *ic* misses, *dc* misses, store operations and L2 misses, as detailed in Section 3.4.

In our experiments we consider one task under analysis (*tua* or τ_a) and several (up to three) contender tasks, referred to as $c(\tau_a)$ or τ_b, τ_c and τ_d. τ_a is always a time-critical task for which a WCET estimate is to be derived.

3.4 Handling multicore contention and cache jitter

In this section we cover an introduction to our objectives and procedure to measure contention suffered in multicore environments.

3.4.1 Goals and challenges

MC2 aims at reliably capturing the impact that multicore contention (handled by the bus arbiter in our reference architecture) and cache jitter have on pWCET estimates. This requires ensuring that the execution time observations collected at analysis capture the impact of the jitter of both. To ease MC2 adoption, this goal has to be achieved under the following restrictions:

1. MBPTA compliance. The proposed technique must be MBPTA-compliant requiring minimum changes to the single-core MBPTA timing analysis approach, which has already been evaluated with several industrial case studies [29].

2. pWCET estimates should be time composable, so that they are independent of the load contenders put on resources. Time composability enables incremental integration of applications, performing timing analysis of applications mostly in isolation, without the need of regression tests. Time composability also allows updating functionality during system operation without the need of analysing the entire task set, but just those tasks that are updated.

3. The information required by MC2 from the tasks should be obtained via PMCs to facilitate its applicability to real hardware.

4. WCET estimates should be obtained as early as possible in the design process to facilitate incremental software integration [42] (ideally during unit testing) and should hold across integrations for incremental verification purposes.

3.4.2 Overall process

MC2 process starts by running the software-randomized task under analysis (τ_a) in isolation, see Figure 3.4. This exposes the impact of cache jitter to the observed execution time (oet^i) in each run r^i. As a side effect, since the hit/miss pattern of τ_a changes across runs (due to software-randomization), its number of accesses to the bus and the memory also varies. Hence, τ_a has an access distribution to cache/memory rather than a single value (with small variations) as it would be the case if τ_a had not been time randomized.

MC2 also factors in the maximum contention delay (mcd) for each request type and the level of contention generated by τ_a's contenders ($c(\tau_a)$), which is refereed as $loc_{c(a)}$. Δ_{cont} in Equation (3.1) captures both mcd and $loc_{c(a)}$. By feeding MBPTA

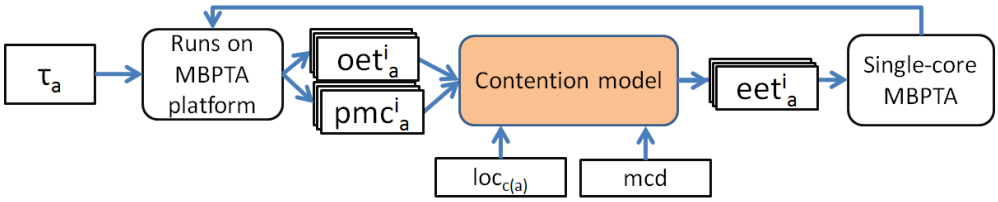


Figure 3.4: Schematic view of the proposed pTC contention model

with enlarged execution times (*eet*), MC2 provides MBPTA with representative information on the impact of cache and multicore contention.

$$eet^i = oet^i + \Delta_{cont} \tag{3.1}$$

3.4.3 Detailed explanation

MC2 builds upon the following assumptions and inputs:

1. τ_a*'s observed execution times* (oet^i_a) in each run r^i_a of τ_a's in isolation.
2. τ_a*'s number of requests* (pmc^i_a) obtained with PMC readings in each run r^i_a of τ_a's in isolation. The particular counters are discussed later.
3. *Worst-case request overlap assumptions*: MC2 assumes that contenders' requests align in the worst possible manner with each τ_a request causing maximum impact on τ_a's execution time. While this assumption is pessimistic, it relieves the end user from modeling the particular cycle when requests occurs, which would be an overly expensive effort, and would only be doable after integration. Instead, assuming *mcd* delay for each contenders' request brings some pessimism but allows MC2 enable WCET estimates during unit testing to favor incremental integration [42]. This is in contrast to the number of requests that can be derived during unit testing and do not change (for our architecture) at integration, i.e., the number of requests of a task to the bus/memory is not affected by its contenders.
4. *User-provided contender's level of contention* ($loc_{c(a)}$): MC2 factors in the contention (i.e., number of requests) of $c(\tau_a)$. To that end, we follow two models. The first one, called fully Time Composable (fTC), assumes each contender task makes as many requests of the longest duration as total number of requests generated by τ_a. The fTC models results in fully time-composable

estimates, but at the cost of over-estimation. To reduce the latter, a second model, called partially Time Composable (pTC), is adjustable to the expected level of contention of the contenders (i.e., its number of requests and their type). The pTC model derives the WCET estimate for τ_a in isolation under a given level of contention of its contenders. At integration time, composability can be assessed by simply checking that the contention level of the particular contenders is smaller than the level assumed at analysis. Both models are detailed in Section 3.5 and Section 3.6 respectively.

3.4.4 Request Characteristics

MC2 requires information about the types of requests to the bus, with emphasis on those having different usage of the bus, and the maximum time each request holds the bus (*mcd*).

From processor manuals, we identify six types of request to the bus (see Table 3.1): load/store requests that hit/miss in L2, and for the case of misses, since the L2 is write-back, request evicting and not evicting dirty data. The former are called dirty misses and the latter clean misses.

The *mcd* for each request, see the second column of Table 3.1, is the time interval (measured in cycles) since a request is granted access to the bus until it relinquishes the bus. Hence, *mcd* is the maximum contention that a request of each type can incur on other requests. We have derived mcd values empirically following the process described in [33]. The general approach consists in generating small benchmarks that generate a single-type of requests (e.g., load hits in L2) and architect experiments so tight bounds to request latencies can be derived.

Table 3.1: Request types and their latency in our reference board

Type	mcd	Description
sh	$l^{sh} = 1$	L2 st hit
lh	$l^{lh} = 8$	L2 ld hit in L2
lmc	$l^{lmc} = 28$	L2 ld clean miss
smc	$l^{smc} = 28$	L2 st clean miss
lmd	$l^{lmd} = 31$	L2 ld dirty miss
smd	$l^{smd} = 31$	L2 st dirty miss

Table 3.2: PMCs available in the reference processor

Name	Description
pmc^{icm}	Bus reads caused by *ic* misses
pmc^{dcm}	Bus reads caused by *dc* misses
pmc^{st}	Writes to L2
pmc^{m}	Misses in the L2

3.5 fTC contention model

fTC derives a WCET estimate that is an upper bound to the slowdown τ_a can suffer regardless of the load its contender tasks put on the shared resources. This requires the model to pessimistically assume that the number of contenders equals $Nc-1$ where Nc is the number of cores – four in our platform. Further, the model assumes that for every τ_a request its contenders have one request of the worst type, i.e., causing the longest contention on it that in our architecture corresponds to lmd and smd (indistinctly referred to as xmd). Hence, fTC assumes that each request of τ_a is delayed 31 $cycles = l^{smd} = l^{lmd}$ by each contenders' request.

Overall, fTC builds a set of enlarged execution time observations as shown in Equation (3.2), where n_a^i is the total number of request that τ_a performs in run r^i.

$$eet_a^i \;=\; oet_a^i + \Delta_{cont}^{i,fTC} = oet_a^i + \left[n_a^i \times (Nc-1) \times l^{xmd}\right] \tag{3.2}$$

3.6 pTC contention model

The fTC model may result in noticeably pessimistic WCET estimates. The partially Time Composable (pTC) model presented in this section trades time composability to tighten WCET estimates. With pTC [41], the user can yet enjoy benefits of incremental integration with small effort to assess time composability.

The pTC model, instead of assuming $Nc-1$ contenders, takes the actual number of running τ_a contenders. Also, unlike fTC, pTC tracks the number of requests of each type. This offers a powerful solution to tighten WCET estimates with a reasonable low impact on time composability. pTC assumes that an upper bound to the number of contenders' request of each type can be derived.

The pTC model derives the impact that the number of requests for each contender task τ_b can cause on τ_a.

Ideally, we would like to have a PMC for the number of requests of each type made by the task. We refer to that ideal counter as n^{xxx} where xxx corresponds to one of the types in Table 3.1. However, in the target platform there is not a specific set of PMCs measuring those values as shown in Table 3.2, which lists the relevant PMCs we used.

We performed an analysis of the relation we derive among the events needed by the pTC model and the PMCs in the architecture (pmc^{yyy}) as shown in Figure 3.5: the number of loads to L2 (n^l) matches the number of misses to the *dc* and the *ic* ($pmc^{dcm} + pmc^{icm}$); the number of stores matches pmc^{st}; and the number of misses pmc^m covers those caused by clean and dirty evictions ($pmc^m = n^{lmc} + n^{smc} + n^{lmd} + n^{smd}$). Further, the number of stores n^{st} matches $pmc^{st} = n^{sh} + n^{smd} + n^{smc}$.

With the existing PMC we approximate the number of requests of each type made by each contender task. In doing so, we take into account the request latency so that the resulting impact that τ_b causes on τ_a derived with PMCs is an upperbound to the actual one we would derive if we had the ideal PMCs. The approach consists in upper bounding high-latency requests first, which in our architecture are dirty misses (*lmd* and *smd*).

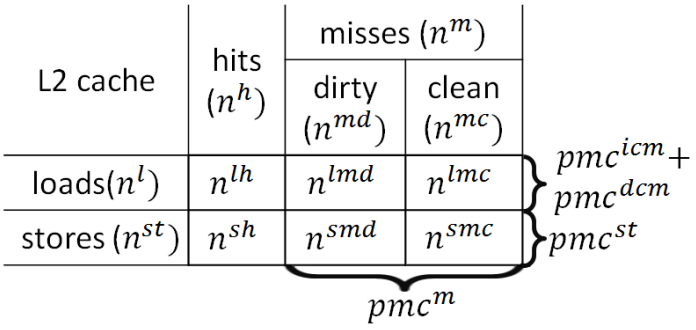


Figure 3.5: Events and PMCs

Bounding Dirty Misses: The number of L2 misses evicting a dirty line is upper bounded by the minimum between the number of stores ($n^{st} = pmc^{st}$) that cause lines to be dirty and the number of L2 misses ($n^m = pmc^m$) that evict cache lines, see left

	τ_b requests conflicting with τ_a requests	τ_a unpaired requests
before pairing		n_a^i
Pairing *md*	$\hat{c}^{md} = \min(n_a, \hat{n}_b^{md})$	$n_a'^i = \max(0, n_a^i, -\hat{c}^{md})$
Pairing *mc*	$\check{c}^{mc} = \min(n_a'^i, \check{n}_b^{mc})$	$n_a''^{i,} = \max(0, n_a'^i - \check{c}^{mc})$
Pairing *lh*	$\hat{c}^{lh} = \min(n_a''^i, \hat{n}_b^{lh})$	$n_a'''^i = \max(0, n_a''^{i,} - \hat{c}^{lh})$
Pairing *sh*	$\check{c}^{sh} = \min(n_a'''^i, \check{n}_b^{sh})$	$n_a''''^i = \max(0, n_a'''^i - \check{c}^{sh})$
after pairing		$n_a''''^i$

Figure 3.6: Pairing steps

part of Equation (3.3).

$$\boxed{\hat{n}^{md} = min(pmc^{m}, pmc^{st})} \quad \rightarrow \quad \boxed{\check{n}^{mc} = pmc^{m} - \hat{n}^{md}} \tag{3.3}$$

Since $n^m = n^{md} + n^{mc}$, the approximation in Equation (3.3) may result in assuming that some misses generate dirty evictions while, in reality, they do not, thus introducing some pessimism. In particular, it results in a lower bound to the number of clean misses ($\check{n}^{mc}$), see the right side of Equation (3.3).

Bounding Load Hits: The number of loads that hit in cache is upper bounded by the minimum between the number of hits (n^h) and the number of loads (n^l) to the L2, see Equation (3.4). They are respectively computed from PMCs as follows:

The number of loads performed to the L2 cache (n^l) equals the number of misses in the *dc* and *ic*, i.e., $n^l = pmc^{icm} + pmc^{dcm}$. Note that the number of loads to the L2 includes hits and misses (dirty and clean), i.e., $n^l = n^{lh} + n^{lmc} + n^{lmd}$.

The number of hits in L2, n^h is obtained with existing PMCs as $n^h = (pmc^{icm} + pmc^{dcm} + pmc^{st}) - pmc^{m}$, that is, the number of read/write accesses to the L2, which include load misses in the *ic* and the *dc* plus stores (due to the write-through policy of the *dc* cache), minus the number of L2 misses. Note that pmc^m does only count the number of direct misses. More specifically, it does not count the number of memory accesses due to write backs.

$$\boxed{\hat{n}^{lh} = min(n^{h}, n^{l})} \quad \rightarrow \quad \boxed{\check{n}^{sh} = n^{h} - \hat{n}^{lh}} \tag{3.4}$$

Since $n^h = n^{lh} + n^{sh}$, a lower bound to the number of store hits is derived as shown in the right side of Equation (3.4).

3.6.1 Bounding contention

Once bounds to τ_b accesses have been computed, the pTC model assumes that requests from τ_b delay τ_a requests by their respective *mcd*. This is implemented by iteratively "pairing" each request from a run r^i of τ_a with one request of τ_b from worst to best latency, see Figure 3.6.

1. First, the number of requests from task τ_b of type miss dirty ($\hat{n}^{md}$), i.e the

type with highest *mcd*, that contend with requests of τ_a, n_a^i, is given by: $\hat{c}^{md} = min(n_a^i, \hat{n}_b^{md})$. Hence the number of unpaired requests from τ_a is $n_a'^i = max(0, n_a^i - \hat{c}_b^{md})$ requests of τ_a unpaired.

2. Those $n_a'^i$ requests contend with $\check{n}^{mc}$ (second most impacting type) requests of τ_b: $\check{c}^{mc} = min(n_a'^i, \check{n}_b^{mc})$. This results in $n_a'' = max(0, n_a'^i - \check{c}^{mc})$ τ_a's unpaired requests.

3. Those $n_a''^i$ requests contend with $\hat{n}_b^{lh}$ (third most impacting type) requests of τ_b: $\hat{c}^{lh} = min(n_a''^i, \hat{n}_b^{lh})$ with $n_a''' = max(0, n_a''^i - \hat{c}^{lh})$ requests unpaired.

4. Finally, the $n_a'''^i$ remaining τ_a's requests contend with $\check{n}_b^{sh}$ (fourth most impacting type) requests of τ_b: $\check{c}^{sh} = min(n_a'''^i, \check{n}_b^{sh})$. With the remaining τ_a's request $n_a''''^i = max(0, n_a'''^i - \check{c}^{sh})$ not contending with any request of τ_b.

The obtained pTC contention is the result of assuming that each of these contentions among τ_a and its contender τ_b are aligned in the worst way, causing a contention delay as long as each τ_b request (see Equation (3.5)). This process is repeated for the other potential contender tasks τ_c and τ_d. The overall pTC contention bound is given by Equation (3.6).

$$\Delta_{\tau_b \to \tau_a}^{i,pTC} = (\hat{c}^{md} \times l^{md}) + (\check{c}^{mc} \times l^{mc}) + (\hat{c}^{lh} \times l^{lh}) + (\check{c}^{sh} \times l^{sh}) \quad (3.5)$$

$$\Delta_{cont}^{i,pTC} = \Delta_{\tau_b \to \tau_a}^{i,pTC} + \Delta_{\tau_c \to \tau_a}^{i,pTC} + \Delta_{\tau_d \to \tau_a}^{i,pTC} \quad (3.6)$$

3.6.2 Other considerations

The fTC model has the advantage of breaking the dependence between scheduling and WCET. In an exact model, the WCET figure to be used depends on the schedule of tasks, which creates a circular dependence as WCET is also an input for deriving a feasible schedule. This issue has been initially tackled by an iterative approach to simultaneously attack WCET and scheduling [43]. With fTC the WCET estimate is not affected by the load contender tasks put on the shared resources, and hence it does not depend on task scheduling. However, this comes at the cost of inflated WCET estimates.

It is worth noting that the principles of the presented model does not only apply to the studied processor but also to other multicore processors. In order to adapt the model, it is necessary to understand the type of events using the shared resource

and their duration. The quality of the results, however, depends on the PMC support available and the accuracy it guarantees in tracking the desired events, see Figure 3.5. As part of our current work we are extending the model to multicore processors in other domains, e.g., automotive.

3.7 Experimental results

We first demonstrate our combined approach on a synthetic application and then with benchmarks of the EEMBC Automotive suite [15].

3.7.1 Hardware Setup

We used an FPGA implementation of the LEON3 [14] platform, as introduced in Section 3.3. Each core comprises separate 16KB 4-way set-associative L1 caches for instruction and data, with write-through, no write allocate policy. The cache hierarchy is complemented by a shared 128KB 4-way unified L2 cache, with write-back policy. An AMBA AHB bus provides connections among private caches, the L2 and the DRAM memory controller. In our setting, we configured the L2 to be partitioned among cores (contention is still to be suffered on bus accesses), so each core has a 32KB direct-mapped L2.

3.7.2 MBPTA Setup

We applied MBPTA to the target program, considering 10^{-12} as the pWCET exceedance probability threshold of interest. We collected 3,000 runs to meet the representativeness requirements, as determined by the ReVS method [38]. The obtained set of observations successfully passed the statistical independence and identical distribution tests, prerequisites to the application of EVT, and allowed MBPTA to converge on a pWCET distribution.

3.7.3 Synthetic application

Our synthetic application resembles an "aggressive" program uniformly accessing the shared bus for 30% of its execution time. It consists of several functions that

are sequentially accessed within a loop a total of one hundred times. Each function comprises a variable number of instructions, performing a mixture of purely arithmetic and read/write operations.

Our empirical evaluation proceeds through two incremental steps. First, we assess the effectiveness of software randomization in enabling MBPTA to capture intra-core cache jitter. To that end, we execute and analyze our program in isolation (i.e., no contention at all), in a simple single-core setup.

Second, we show how the results in isolation can be complemented with the analysis of inter-core contention jitter. We therefore assess our analytical model, combining cache and contention jitter, against representative execution scenarios where all cores in the system are concurrently enabled.

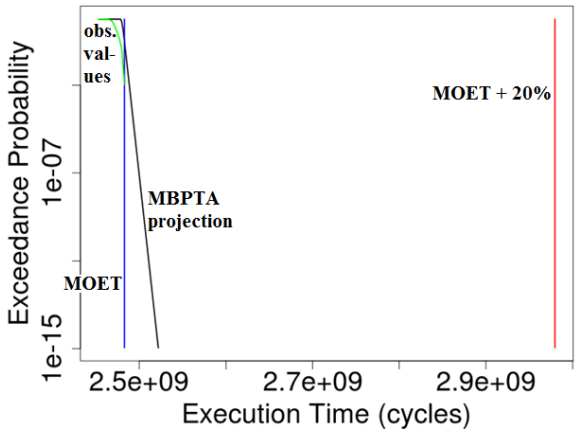


Figure 3.7: Results of TASA and MBPTA on a single-core setup.

Capturing Cache Jitter. We exploited TASA to enable MBPTA to capture the execution time variability incurred by caches. To this extent, we analyzed our application in a single-core setup, guaranteeing complete isolation.

Figure 3.7 reports the pWCET distribution computed by MBPTA for the target program. Observed values are upperbounded by the MBPTA projection and pWCET result is obtained by selecting the value of the projection at the 10^{-12} exceedance threshold. In this case the pWCET distribution is particularly close to the maximum observed execution time (MOET). It is worth noting that, since plain observed values do not provide any worst-case guarantee, it is common (though pretty unscientific) industrial practice to resort to a *fudge factor* to account for unknown factors. This factor is typically in the order of magnitude of 20% of the MOET.

Notably, the pWCET computed with MBPTA is not only much tighter than the 20% margin, but also comes with scientific reasoning.

Multicore Contention. The MC2 approach extends single-core pWCET estimates by capturing the effect of inter-core contention through a contention model based on PMCs. In order to assess the precision of our analytical model, we performed a set of experiments on representative execution scenarios where all cores in the system are concurrently enabled and compared against the results of our analytical (fTC and pTC) models.

In our setting and platform, the theoretical worst-case inter-core contention suffered by an application corresponds to the fTC scenario where all bus access requests are triggered one cycle after the reserved slot and all other cores already have pending requests, each one incurring the latency of a L2 dirty-miss. While being fully time-composable, this scenario can be extremely pessimistic in practice as it can only occur under extremely bad and rare overlapping of bus requests and cache miss patterns.

fTC. We consider first the fTC contention model as it is used as a reference for the pTC one. Our application, τ_a, is executed under two different scenarios of contention: (1) against three stressing kernels performing loads that miss in L2 (i.e., clean misses) and (2) against three stressing kernels performing stores in L2 overwriting data (i.e., dirty misses). Figure 3.8 shows the execution time of τ_a under the two scenarios of contention and the bound derived with the fTC model. Values are normalized against the MOET from baseline observations (i.e., no contention). As expected the model is accurate when the execution conditions are matching the fTC assumptions (i.e., worst request latencies and alignment). In practice, contenders will not generate those overly-conflictive scenarios. The pTC model cures the pessimism coming from the worst-case latency assumption.

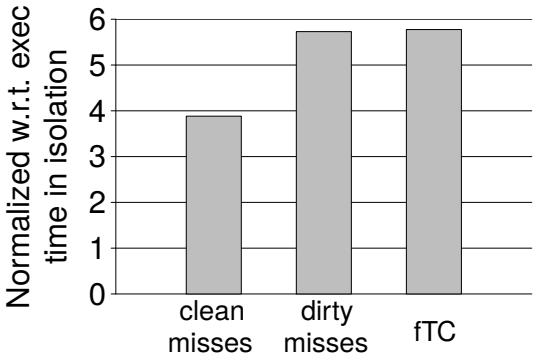


Figure 3.8: Execution time when $c(\tau_a)$ create clean/dirty misses and fTC

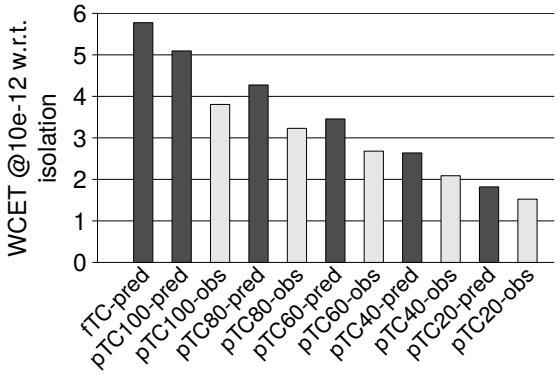


Figure 3.9: Result of the pTC under different load scenarios generated by the contenders

pTC. To compare the accuracy of the pTC model and how it adapts to contenders'

load on the bus, we run our application τ_a against three copies of a benchmark that performs a variable number of bus accesses depending on the configuration, which we express as a percentage of τ_a accesses. Figure 3.9 compares the observed execution times against the predictions of the pTC model. Results from applying the fTC are included as well for the sake of comparison.

As expected, the fTC model yields pessimistic pWCET estimates. Conversely, we observe that the pTC model computes pWCET estimates decrease in parallel with the load put by contenders on the shared bus. Note that the difference among fTC and $pTC-100\%$ is that the former assumes that all requests contribute the worst-case latency (dirty misses), whereas the latter accounts for the actual type of requests of the contenders. For any value of p, e.g., $pTC - 40\%$, the derived $pWCET@10^{-12}$ with pTC tightly upperbounds the actual observed value.

All in all this synthetic evaluation confirms that the MC2 method effectively captures both cache and multicore contention into pWCET estimates that are analytically reliable and tightly upperbounding the observed values.

3.7.4 EEMBC

To further evaluate our approach we applied MC2 on the EEMBC automotive benchmarks [15]. In particular, we analyzed `a2time`, `cacheb`, `idctrn`, `iirflt`, `puwmod`, and `tblook` on the same platform. Figure 3.10 reports, for each benchmark, MOET in both singlecore and multicore scenarios, and the results of the fTC and pTC models. For the pTC model contention was generated by deploying three copies of the benchmark itself. All results are normalized with respect to the multicore MOET.

First we observe that fTC WCET estimate are in general extremely high ($\sim$11x). This is explained by the fact that fTC model assumes not only the worst-case alignment scenario but also the worst-case latencies for each contender access, which is generally unrealistic. The pTC model, instead, provides quite good results, with all values below 1.5x the multicore MOET.

The only pessimism in the pTC model comes from its conservative assumptions on the alignment of requests. pTC estimates provides a good compromise between tightness and flexibility: a further reduction in pessimism cannot be had without exact knowledge on how bus accesses interleave, which is not flexible and typically too difficult to derive.

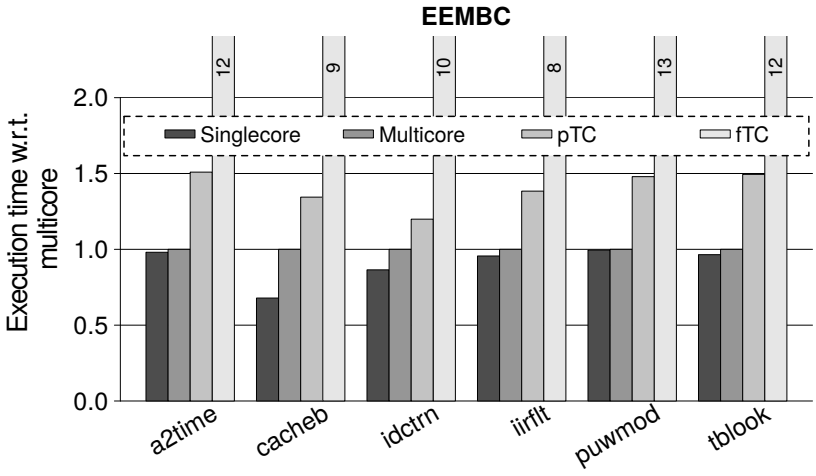


Figure 3.10: Results of EEMBCs against 3 copies of themselves

3.8 Related work

Several approaches have been proposed to account for inter-core contention by computing an upper bound to the delay a task or application may suffer [44]. Some of those approaches require extending classic timing analysis framework to account for the effect of shared resources [45], but they are generally unsustainable owing to the entailed computational complexity. Other approaches suggest a separate (compositional) analysis approach [46, 47, 48]. They propose a separate analysis for contention and, frequently, rely on splitting tasks into sub-tasks or phases so that worst-case alignment in (typically) TDMA-based arbiters can be reasonably computed. Assuming that tasks can be split into phases allows refining the analysis model and reducing the overall pessimism; however, this assumption is quite application-dependent and cannot be generalized.

Moreover, the above approaches typically rely on insightful information on all the applications in the system and a preliminary static analysis step to characterize the pattern of memory accesses. Conversely, the contention analysis approach we rely on limits the pessimism while at the same time making no assumption on how memory accesses are distributed. Our model only requires support for PMCs, which is often available (though at variable extent) in COTS platforms.

Other approaches make use of specific hardware and/or RTOS mechanisms to enforce precomputed bounds to the maximum contention caused/suffered at run time [49, 50]. While interesting, those approaches do rely on domain-specific and custom run-time hardware mechanisms that are not typically available, and yield results that are only valid under the specific task set and system configuration.

Our approach, instead, derives bounds on the inter-core contention that are at the same time realistic and only partially dependent on the co-runners characteristics, as a first step towards enabling incremental development and qualification.

The use of PMCs to model contention and derive an upper bound to multicore contention delays has been originally introduced in [33], where the analytical model for fTC and pTC is tailored to the NGMP platform. In this work, we readapt the same concept to the MBPTA framework and combines the contention model in [33] (adapted to the LEON3) with software randomization to provide holistic pWCET bounds, accounting for both cache jitter and contention effects.

3.9 Conclusions

We have proposed MC2, a technique for COTS multilevel-cache multicores that derives WCET estimates factoring in the jitter generated by caches and multicore contention. To that end, each measurement fed in input to MBPTA systematically accounts for the impact of both resources, effectively enabling MBPTA to factor them in when deriving pWCET estimates. Our results on a COTS platform confirm that MC2 effectively captures the impact of both multi-level cache variability and inter-core contention in realistic WCET estimates, that tightly upperbound observed values.

4. Summary

Measurement-based timing analysis is, and will continue to be, the most used technique to address timing analysis in industry. In this work we advocate for its probabilistic branch to tackle the challenges presented by high performance architectures. Furthermore, measurement-based probabilistic timing analysis covers sources of execution time variability caused by hardware and software that cannot be fully captured by the classic MBTA approach. It is important to remark that MBPTA application is still subject to the inputs used at analysis, i.e. MBPTA cannot predict events that have not been captured during analysis.

We have shown that software instrumentation, used in MBTA and MBPTA, invalidates the timing analysis if instrumentation instructions are simply removed from the binary deployed. In fact, we will be facing a completely different binary, thus losing those guarantees obtained during the analysis phase. The solution proposed allows keeping valid results obtained with software instrumentation. We show that replacing instrumentation instructions by plain *nops* makes everything simpler with negligible cons.

We have also presented a technique for COTS multilevel-cache multicores to derive reliable WCET estimates taking into account variability coming from the cache hierarchy and variability caused by multicore contention. MBPTA considers both sources of jitter and outputs a trustworthy pWCET estimate. Overall, we have advanced the state of MBPTA contributing to its consolidation for future real-time embedded systems.

Bibliography

[1] "Automotive Industry Drives Chip Demand." `http://www.eetimes.com/document.asp?doc_id=1324718`. Accessed: 2017-06-06.

[2] R. N. Charette, "This car runs on code," *IEEE spectrum*, vol. 46, no. 3, p. 3, 2009.

[3] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, *et al.*, "The worst-case execution-time problem overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, pp. 1–53, May 2008.

[4] J. Abella, C. Hernandez, E. Quinones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega, "WCET analysis methods: Pitfalls and challenges on their trustworthiness," in *Industrial Embedded Systems (SIES), 2015 10th IEEE International Symposium on*, pp. 1–10, IEEE, 2015.

[5] L. Kosmidis, E. Quiñones, J. Abella, T. Vardanega, I. Broster, and F. J. Cazorla, "Measurement-based probabilistic timing analysis and its impact on processor architecture," in *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pp. 401–410, IEEE, 2014.

[6] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based timing analysis," in *ISOLA*, 2008.

[7] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quinones, and F. J. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pp. 91–101, IEEE, 2012.

[8] L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla, "A cache design for probabilistically analysable real-time systems," in *DATE*, 2013.

[9] S. Kotz and S. Nadarajah, *Extreme value distributions: theory and applications*. World Scientific, 2000.

[10] `http://www.gaisler.com/index.php/products/debug-tools/grmon`, *Cobham Gaisler*. GRMON.

[11] B. Rieder, I. Wenzel, K. Steinhammer, and P. Puschner, "Using a runtime measurement device with measurement-based WCET analysis," in *IESS*, 2007.

[12] `https://www.rapitasystems.com/products/rtbx`, *RTBx*. Rapita Systems.

[13] L. Kosmidis, J. Abella, F. Wartel, E. Quiñones, A. Colin, and F. J. Cazorla, "Pub: Path upper-bounding for measurement-based probabilistic timing analysis," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pp. 276–287, IEEE, 2014.

[14] `http://www.gaisler.com/index.php/products/processors/leon3`, *Leon3 Processor*. Cobham Gaisler.

[15] J. Poovey *et al.*, "Characterization of the EEMBC benchmark suite," *North Carolina State University*, 2007.

[16] ERA (European Railway Agency), "ERTMS - Set of specifications - 2 (ETCS baseline 3 and GSM-R baseline 0)," 2014.

[17] S. M. Petters, "Comparison of trace generation methods for measurement based WCET analysis," in *WCET Analysis Workshop*, 2003.

[18] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based worst-case execution time analysis," in *SEUS Workshop*, 2005.

[19] A. Betts, N. Merriam, and G. Bernat, "Hybrid measurement-based WCET analysis at the source level using object-level traces," in *WCET Analysis Workshop*, 2010.

[20] R. Kirner, P. Puschner, I. Wenzel, *et al.*, "Measurement-based worst-case execution time analysis using automatic test-data generation," in *SEUS Workshop*, 2004.

[21] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner, "Automatic timing model generation by CFG partitioning and model checking," in *DATE*, 2005.

[22] S. Stattelmann and F. Martin, "On the use of context information for precise measurement-based execution time estimation," in *WCET Analysis Workshop*, 2010.

[23] B. Dreyer, C. Hochberger, S. Wegener, and A. Weiss, "Precise continuous non-intrusive measurement-based execution time estimation," in *WCET Analyis Workshop*, 2015.

[24] D. Buttle, "Real-time in the prime-time, ETAS GmbH, Germany," in *Keynote talk at 24th Euromicro Conference on Real-Time Systems, Pisa, Italy, July*, 2012.

[25] G. Edelin, "Embedded systems at THALES: the Artemis challenges for an industrial group," in *Lecture at ARTIST Summer School*, (Autrans, France), 2009.

[26] A. West, "NASA Study on Flight Software Complexity. Final Report," tech. rep., NASA Excellence Program, 2009.

[27] International Organization for Standardization, *ISO/DIS 26262. Road Vehicles – Functional Safety.* ISO, Geneva, Switzerland, 2009.

[28] Federal Aviation Administration, Certification Authorities Software Team (CAST), *CAST-32A Multi-core Processors*, 2016.

[29] F. Wartel, L. Kosmidis, A. Gogonel, A. Baldovin, Z. Stephenson, B. Triquet, E. Quiñones, C. Lo, E. Mezzetti, I. Broster, J. Abella, L. Cucu-Grosjean, T. Vardanega, and F. J. Cazorla, "Timing analysis of an avionics case study on complex hardware/software platforms," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pp. 397–402, 2015.

[30] F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quiñones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, T. Vardanega, and F. J. Cazorla, "Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study," in *Int. Symposium on Industrial Embedded Systems*, June 2013.

[31] C. Hernandez, J. Abella, A. Gianarro, J. Andersson, and F. J. Cazorla, "Random modulo: A new processor cache design for real-time critical systems," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2016.

[32] L. Kosmidis, C. Curtsinger, E. Quiñones, J. Abella, E. Berger, and F. J. Cazorla, "Probabilistic timing analysis on conventional cache designs," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, 2013.

[33] J. Jalle, M. Fernandez, J. Abella, J. Andersson, M. Patte, L. Fossati, M. Zulianello, and F. J. Cazorla, "Bounding resource contention interference in the next-generation microprocessor (ngmp)," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[34] M. Fernández, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla, "Assessing the suitability of the ngmp multi-core processor in the space domain," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, (New York, NY, USA), pp. 175–184, ACM, 2012.

[35] S. Law and I. Bate, "Achieving appropriate test coverage for reliable measurement-based timing analysis," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 189–199, July 2016.

[36] J. Nowotsch, M. Paulitsch, A. Henrichsen, W. Pongratz, and A. Schacht, "Monitoring and wcet analysis in cots multi-core-soc-based mixed-criticality systems," in *Conference on Design, Automation & Test in Europe (DATE)*, 2014.

[37] W. Feller, *An Introduction to Probability Theory and Its Applications*. John Willer and Sons, 1968.

[38] S. Milutinovic, J. Abella, and F. J. Cazorla, "Modelling probabilistic cache representativeness in the presence of arbitrary access patterns," in *International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 142–149, May 2016.

[39] L. Kosmidis, R. Vargas, D. Morales, E. Quiñones, J. Abella, and F. J. Cazorla, "Tasa: Toolchain-agnostic static software randomisation for critical real-time systems," in *International Conference on Computer-Aided Design*, pp. 1–8, Nov 2016.

[40] L. Kosmidis, E. Quiñones, J. Abella, G. Farrall, F. Wartel, and F. J. Cazorla, "Containing timing-related certification cost in automotive systems deploying complex hardware," in *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, (New York, NY, USA), pp. 22:1–22:6, ACM, 2014.

[41] G. Fernandez, J. Jalle, J. Abella, E. Quiñones, T. Vardanega, and F. J. Cazorla, "Resource usage templates and signatures for cots multicore processors," in *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, 2015.

[42] E. Mezzetti and T. Vardanega, "A rapid cache-aware procedure positioning optimization to favor incremental development," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 107–116, April 2013.

[43] G. Fernandez, J. Abella, E. Quiñones, L. Fossati, M. Zulianello, T. Vardanega, and F. J. Cazorla, "Seeking time-composable partitions of tasks for cots multicore processors," in *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pp. 208–217, April 2015.

[44] G. Fernandez, J. Abella, E. Quiñones, C. Rochange, T. Vardanega, and F. J. Cazorla, "Contention in multicore hardware shared resources: Understanding of the state of the art," in *OASIcs-OpenAccess Series in Informatics*, vol. 39, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[45] S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, "A unified wcet analysis framework for multi-core platforms," in *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pp. 99–108, April 2012.

[46] S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst, "Reliable performance analysis of a multicore multithreaded system-on-chip," in *6th international conference on Hardware/Software codesign and system synthesis*, CODES '08, 2008.

[47] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Timing analysis for tdma arbitration in resource sharing systems," in *16th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS, 2010.

[48] D. Dasari, V. Nelis, and B. Akesson, "A framework for memory contention analysis in multi-core platforms," *Real-Time Systems*, vol. 52, no. 3, pp. 272–322, 2016.

[49] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware support forWCET analysis of hard real-time multicore systems," in *36th Annual International Symposium on Computer Architecture*, ISCA, 2009.

[50] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement," in *Euromicro Conference on Real-Time Systems*, July 2014.