Measurement-Based Timing Analysis of the AURIX Caches[*]

Leonidas Kosmidis[1], Davide Compagnin[2], David Morales[3], Enrico Mezzetti[4], Eduardo Quinones[5], Jaume Abella[6], Tullio Vardanega[7], and Francisco J. Cazorla[8]

1 **Universitat Politècnica de Catalunya, Barcelona, Spain; and Barcelona Supercomputing Center, Spain**
2 **University of Padova, Padova, Italy**
3 **Barcelona Supercomputing Center, Barcelona, Spain**
4 **Barcelona Supercomputing Center, Barcelona, Spain**
5 **Barcelona Supercomputing Center, Barcelona, Spain**
6 **Barcelona Supercomputing Center, Barcelona, Spain**
7 **University of Padova, Padova, Italy**
8 **Barcelona Supercomputing Center, Barcelona, Spain; and IIIA-CSIC, Barcelona, Spain**

Abstract

Cache memories are one of the hardware resources with higher potential to reduce worst-case execution time (WCET) costs for software programs with tight real-time constraints. Yet, the complexity of cache analysis has caused a large fraction of real-time systems industry to avoid using them, especially in the automotive sector. For measurement-based timing analysis (MBTA) – the dominant technique in domains such as automotive – cache challenges the definition of test scenarios stressful enough to produce (cache) layouts that causing high contention. In this paper, we present our experience in enabling the use of caches for a real automotive application running on an AURIX multiprocessor, using software randomization and measurement-based probabilistic timing analysis (MBPTA). Our results show that software randomization successfully exposes – in the experiments performed for timing analysis – cache related variability, in a manner that can be effectively captured by MBPTA.

1998 ACM Subject Classification D.4.7 Real-time Systems and Embedded Systems

Keywords and phrases WCET, caches, AURIX, Automotive

Digital Object Identifier 10.4230/OASIcs.WCET.2016.9

1 Introduction

Despite the complexity added by caches to timing analysis [11], their potential benefits in the reduction of WCET estimates have motivated their analysis for decades [27]. In terms of

[*] 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. Jaume Abella has been partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal fellowship RYC-2013-14717. This work was conducted as part of a collaboration with the CONCERTO project (ARTEMIS-JU grant nr. 333053), which provided the automotive application and the build automation. Authors thank Benjamin Lesage for his support in execution time collection infrastructure on the AURIX board.

performance, it is well known that the program memory layout, i.e. the addresses in which the program's code and data are located, determines the program's *cache layout*, i.e. the cache sets to which the program's data and code are mapped. For a given program, the execution time variability under different cache layouts can be significant, ultimately affecting processor performance [22][25][23]. This occurs because both even small variations in the order in which the object files are linked together [25] and in other elements of the memory layout (e.g. environmental variables) may significantly affect execution time. This variability in execution time complicates timing analysis significantly. In confirmation of the difficulties that caches bring into timing analysis, the state of the art chips for the automotive market typically include large scratchpads (32KB for code and 120KB for data in our target platform) and relatively smaller caches (16KB for code and 8KB for data in our target platform), possibly disabled by default. However, in the general case, transparently-managed caches offer a more flexible and efficient hardware acceleration mechanism than user-controlled scratchpad. The computational power required to support the user demands for increasingly complex functionality suggests that in the long term, scratchpads and caches will coexist in embedded real-time systems. It therefore stands to reason that cache-related variability needs be characterized and analysed.

Measurement-based timing analysis (MBTA) methods are widely used in application domains such as automotive [27][1]. MBTA application comprises the *analysis phase* in which verification of the timing behaviour is performed and the *operation phase* in which the system is deployed in the actual operational environment. MBTA aims at estimating a WCET estimate that holds during operation with execution-time measurements taken at analysis time. In the context of MBTA, the challenge of using caches lies on providing evidence that, in the measurements runs, the cache layouts that lead to high execution times are properly factored in when computing the WCET estimates. However, in general, it is hard for the user to design experiments in which bad (worst) cache layouts are enforced. This reduces the confidence on the WCET estimates obtained with MBTA required for timing verification according to the domain-specific safety standards, resulting in the cache not being fully embraced by the real-time industry.

Performing an exhaustive exploration of the space of potential cache layouts is not only practically difficult (assuming full control of hardware and software states) but also computationally infeasible in the general case. Cache randomization techniques [14][18] may help in this respect by ensuring that a new random cache layout is exercised in every program run. As a result, the search space of cache layouts is transparently (and randomly) explored by performing additional runs of the program at analysis time. This, in turn, allows deriving a probabilistic characterization of the impact of changes in the cache layout on the execution time of a program. Furthermore, such probabilistic characterization facilitates the application of Measurement-based Probabilistic Timing Analysis (MBPTA) [9, 1] in that it probabilistically guarantees that those cache layouts leading to higher execution times have been captured at analysis time.

Two main flavours of cache randomization exist: hardware randomization employs time-randomized caches [13], featuring random placement and replacement policies; software randomization [18] instead randomizes the memory layout of the program's data and code

[1] While the scientific literature extends on trade-offs between static and measurement-based approaches [3], industrial practitioners take a pragmatic view to when to use either, which considers cost-effectiveness in achieving required evidence. This paper contributes how to increase confidence on measurement-based approaches – massively used in automotive – in the presence of caches.

across runs. In this paper we present our experience with applying MBPTA and software-based cache randomization to probabilistically characterize and analyse the impact of code and data layout of an Automotive Cruise Control System (ACCS) running on an AURIX TC277 processor board [26]. In this work is we do not provide an analysis method for the TC277, which has been designed to be with determinism in mind. Nor we make a comparison between MBPTA and static timing analysis techniques, which has been already done [3]. Instead, in this paper we demonstrate that the impact of cache layout on execution time variability is relevant even on deterministic architectures, showing how this variability can be effectively analysed and characterized with randomization and MBPTA.

2 Background and Problem Statement

In the presence of caches, the memory layout of a program impacts the pattern of hits and misses – and hence the timing behaviour – of individual program runs [7]. Even small changes in their memory layout may cause significant jitter in the observed timing behaviour [19]. The memory layout is subject to small – many times transparent-to-the-user – changes (e.g., compiler flags or even small code modifications). As a result, no deterministic timing analysis approach can provide trustworthy timing guarantees unless either the memory layout is guaranteed not to change or the worst-case layout is determined (and observed, in MBTA approaches). However, freezing the memory layout is typically not possible before the very end of the development process and the definition of a global (best-) worst-case memory layout for an application comprising several tasks is a generally intractable problem. Needless to say, changes to the memory layout may always occur even after deployment, as a result of code patches and other modifications.

MBPTA [9, 1] offers a solution to probabilistically characterise the impact of memory layout on the timing behaviour of a program while still exhibiting the appealing cost-benefit ratio of deterministic measurement-based methods [27]. Probabilistic WCET (pWCET) estimates computed with MBPTA differ from standard WCET figures in that they do not consist on a single value but corresponds to a probability distribution of high execution times. The pWCET distribution conservatively models the residual risk for *one instance* of the target program to exceed a given execution time bound. Users are then interested in those execution time bounds whose exceedance probability is considered acceptable in relation to the integrity level of the functionality being analysed, which in turns depends on the corresponding safety standard.

MBPTA builds on extreme value theory [17] (EVT) to model the pWCET distribution. EVT requires that the observed execution times of the program under analysis must be modellable with *independent and identically distributed* (i.i.d.) random variables. This requirement has been shown to be achievable with the adoption of MBPTA-compliant hardware [15]. In the absence of time-randomised hardware, by exploiting specific software techniques. In particular, software randomisation (SWRand) approaches have been shown to enable the analysis of deterministic caches with MBPTA [14][18]. Different software randomization variants are presented in Section 5.

Interestingly MBPTA and EVT are not the same thing [8]: while EVT can be, in principle, applied to time-deterministic architectures, it would still requires the user to provide evidence that the measurements at analysis time capture the potential variability at operation time. This same requirement, instead, is almost met by construction in MBPTA-compliant platforms where the sources-of-variability are operated almost transparently to the user (or with low user intervention) in a way that guarantees that the *jitter they cause at analysis matches or upperbounds that which may occur during operation* [15].

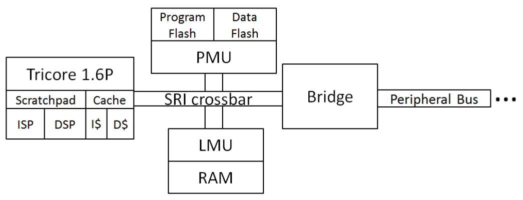


Figure 1 Block Diagram of one core and the LMU/PMU in the AURIXTC277.

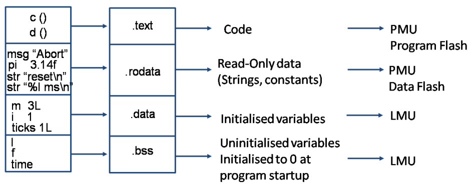


Figure 2 Memory mapping configuration used for the application.

- *Deterministic upperbounding.* Some resources are forced to work on their worst latency so that the analysis time measurements capture the worst timing behaviour that those resources may have during operation [15]. This can be applied to resources with limited impact in the WCET (e.g., some floating point operations with input-dependent latencies).
- *Probabilistic upperbounding.* The previous approach is unaffordable for resources in which jitter is high, since assuming every access to incur the longest latency would cause a significant performance degradation. Instead resource timing randomization and probabilistic reasoning is used. For instance, the reliability of the upperbounds derived by EVT over deterministic caches lies on the user ability to design experiments that exercise conflictive cache layouts causing high execution times. With hardware randomization [13] or software randomization [14], the space of cache layouts is randomly sampled in every new run. Hence, the user only cares about the number of runs to perform instead of having to enforce specific cache layouts.

It is worth noting that MBPTA is exposed to the risk of not capturing the representativeness of events with significant impact on execution time and low probability [2][24]. As discussed in [2][20][21], it is possible to reduce the risk that any such timing event has not being captured in the execution time measurements used by MBPTA. Further exploring this topic is part of our future work.

3 Target AURIX Platform

For this work we use an AURIX TC277 board [26]. Although it is a three-core architecture (TriCore), in this study we focus on a single-core configuration in order to isolate the cache effects from contention-related jitter [29]. That is the application described in Section 4 is executed on one core, while the other two cores remain idle. As part of our future work we aim at combining the solutions to simultaneously handle cache jitter and contention jitter.

The AURIX platform has been designed to have as few jittery resources as possible, following current industrial practice for timing analysis based on determinism. In particular, all the three cores present in the platform are equipped with scratchpad memories to provide deterministic memory accesses. As a platform designed for reliability, the AURIX TC277 is a heterogeneous platform comprising differently implemented cores, which share a common ISA. The first core, which is optimised for low-power execution, has a simpler microarchitecture than the other two high-performance cores, which are equipped with high-performance features such as caches, which create jitter. Our goal is to show how caches in an automotive system like the one presented in this paper can be handled with randomisation and MBPTA. Cores are connected to the 'memory system' through the Shared Resource Interconnect (SRI), see Figure 1. The memory system comprises a SRAM device (LMU - Local Memory Unit) and FLASH device (PMU - Program Memory Unit). The 32 KB SRAM shared memory can

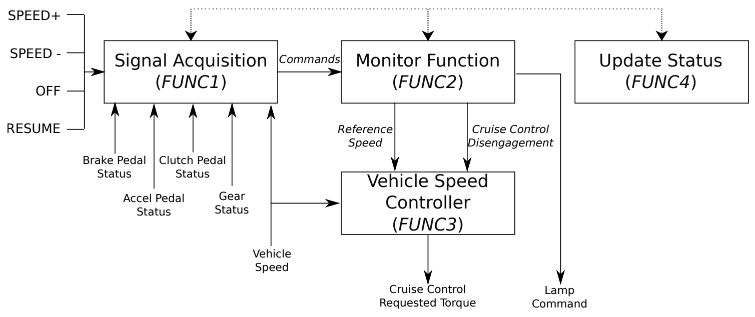


Figure 3 Case Study application overview.

be defined through the LMU as cacheable or uncacheable. Similarly, the PMU flash memory can be also configured as cacheable or not, and is divided in separate units for code and data. The application/RTOS defines statically in a linker script which application software element (function, data etc) is mapped in which hardware resource (LMU, PMU, scratchpad, CPU number etc) and whether accesses to it will be cached or not. Each cacheable memory access can result in a hit or miss in cache, which creates a source of variability. This is handled in the target AURIX platform by MBPTA and static software randomisation (SSR, see Section 5 for further details).

In the integration of SSR in our target AURIX platform we note that AURIX implements a physical memory management unit in which the memory address used in read and write operations determines the device location in which data reside and their cacheability. Concretely, AURIX defines 15 memory segments defined during the linker process, specifying that objects must be located in the LMU, PMU, scratchpads or caches. In order for the timing effects of SSR, as described in Section 5, to be effective, memory objects need to be located in cacheable memory segments: concretely in segments 8 and 9, which allow cached access to PMU and LMU respectively, as shown in Figure 2. For our experiments, the application is executed on one of the high-performance cores, which features caches.

4 Automotive Cruise Control System (ACCS) Case Study

The real-world application analysed in this work implements an Automotive Cruise Control System (ACCS), automatically generated from a Simulink model. The functional code generated by Simulink has been merged to the architectural code obtained by model transformations applied to the CONCERTO representation of the original application model[2]. The application has been adapted to run on top of a customized version of ERIKA[3], a OSEK/VDX compliant Real-time Operating System, which has been modified to exhibit time-composable timing behaviour [4]. The application includes stubbing for IOs to facilitate the analysis and keeps iterating over a discrete and finite set of inputs. While this may limit the realism in the observation of the application behaviour, the resulting execution scenario is acceptable for the intent of our work.

As depicted in Figure 3, the application consists of four main tasks:

- *Signal Acquisition*: it provides signal acknowledgement to the system. It periodically

[2] CONCERTO, ARTEMIS JU, `http://www.concerto-project.org/`.
[3] Erika Enterprise RTOS, `http://erika.tuxfamily.org/drupal/`.

reads and converts the inputs (i.e., the pedals, the cruise user controls and the vehicle speed) and provides them to the Monitor Function task;

- *Monitor Function*: it implements a finite-state machine for the ACCS. It computes the state transition when activated by the Signal Acquisition task;
- *Vehicle Speed Controller*: it performs several interpolations to compute the requested torque according to the system state and the vehicle speed. Its execution is triggered by the Monitor Function task;
- *Update Status*: it updates the inputs according to the current execution time.

The application has been deployed on the AURIX target according to a specific configuration where the global data and the task's private stack are allocated in the cached SRAM. Instructions are stored in the PFlash memory and accessed from the cacheable segment. Input and outputs were stubbed to dispense with the use of actual sensors and actuators.

5 Software Randomization

The main characteristic of cache software-randomization techniques is that they enable the use of MBPTA on COTS, i.e. non-hardware randomized, caches. Software randomization techniques help assessing that the jitter generated by the cache in analysis phase exposes the jitter that the cache can generate at operation phase. To the best of our knowledge, software randomization is the only technique that enables the analysis of COTS caches with MBPTA. There are two main variants of software randomization: dynamic and static.

Dynamic Software Randomisation (DSR) [14][10] performs the randomisation at runtime during the initialisation phase of the program, so that the location of objects in memory is randomised across different executions of the program. DSR combines a compiler pass that modifies appropriately the intermediate representation of the application's code and a run-time system, based on self-modifying code, that is in charge of performing the relocation of objects in memory. The memory objects whose location is randomised are code, stack frames and global data. However, it has been shown [18] that DSR generated code makes intensive use of pointers and dynamic objects which complicates its use in automotive, where the ISO26262 [12] standard requires a limited use of pointers and no use of dynamic objects (among others). Moreover, most automotive processors and microcontrollers, including AURIX, impose practical limitations which prevent the use of self-modifying code, such as fetching code and read-only data directly from read-only flash devices, featuring small dynamic memories (LMU) and strong memory protection units.

With *Static Software Randomisation* (SSR) [18] instead, the program code, stack and global data are allocated to random memory positions across images, achieving the same effect as DSR in an entirely static manner. SSR builds on the fact that memory objects in the executable defines their placement in main memory, and therefore the cache layout. SSR carries out all relocation operations statically at compile time. SSR generates a number of different binaries of the same program each with different random allocation of memory objects in the executable. By randomly selecting an executable from the pool of the generated ones, the timing properties required for the pWCET estimates are preserved. Similarly to DSR, SSR randomises code (functions), stack frames and global data. Interestingly SSR is certification aware since it does not use pointers [18]. For these reasons, and because of the practical limitations of applying DSR on AURIX, this paper focus on SSR.

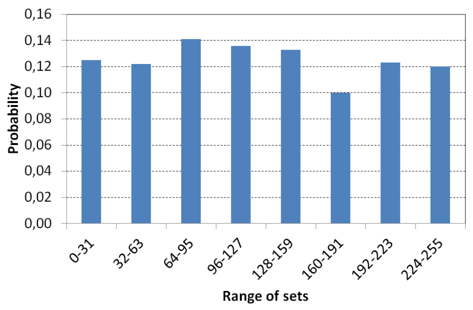


Figure 4 Cache Set distribution of the first instruction of FUNC1 in set groups.

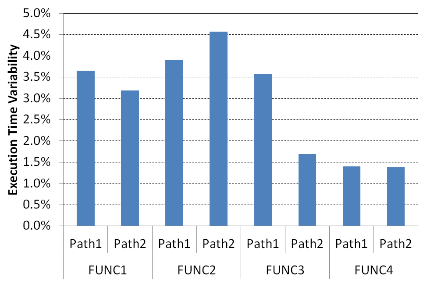


Figure 5 Observed Jitter across runs of the different FUNC for each path.

6 Experimental Results

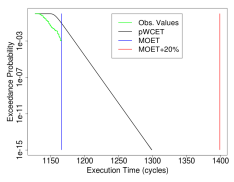
In the AURIX platform, the execution time of ACCS may be influenced by:
(i) the initial processor state when the task starts its execution,
(ii) the interference from the underlying RTOS,
(iii) the time randomisation injected by some processor resources and
(iv) the task input data.
All the above factors are taken into account in our experiments by guaranteeing that the execution conditions enforced at analysis time over-approximate those at deployment. We cover issue (i) by flushing the cache state before each experiment: we start collecting measurements only after the cache warms up and execution times stabilize, which occurs after very few iterations. With respect to factor (ii), we guarantee the OS-induced jitter to be probabilistically analysable [4] by using a time-composable version of the ERIKA [29], as noted in Section 3. The jitter introduced by SSR is also taken into account by MBPTA [9, 1], thus covering issue (iii).

In terms of input-related variability (iv), the MBPTA variant we use [9, 1] can only reason about the paths triggered by the test runs performed at analysis. Solutions to deal with path coverage in MBPTA exist [16][28] but they are not required in this specific context as we rely on a controlled set of input vectors, leading to the different paths. In fact, in our evaluation we focused on two specific paths of the ACCS application, corresponding to the worst-case and best-case observed paths for the used input vectors. We refer to those paths as `Path1` and `Path2` respectively and reason on pWCET estimates obtained separately for each path.

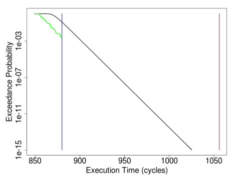
6.1 Randomisation Assessment

We validate that SSR uniformly randomises the cache layout by studying the distribution of the first instruction of the functions under analysis, in the instruction cache sets. Since all the functions under analysis have the same behaviour, in Figure 4 we provide this information for only one of them, FUNC1. Given the initial address of the function, recorded across 1,000 runs, we determine the probability that the first instruction of the function is mapped to any group of 32 sets (0–31, 32–63, . . . , 224–255). As it can be observed SSR achieves a fairly balanced distribution across runs, with similar results obtained for other group sizes. Note that for a higher number of runs, the probability of each group converges to 1/8=0.125, due to the uniform random nature of SSR. This evidences that SSR randomises effectively the cache memory layout.

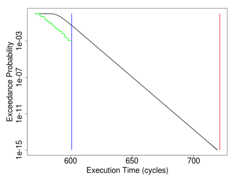
Since the basic principle of SSR is that the memory layout has a direct impact on the execution time, we also assess the effect of cache layout randomisation on the execution time.



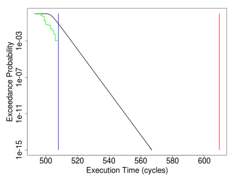
(a) FUNC1-Path 1



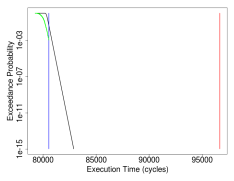
(b) FUNC1-Path 2



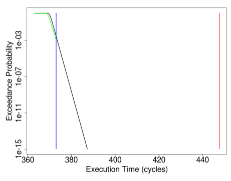
(c) FUNC2-Path 1



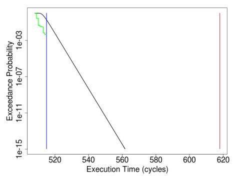
(d) FUNC2-Path 2



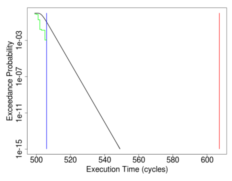
(e) FUNC3-Path 1



(f) FUNC3-Path 2



(g) FUNC4-Path 1



(h) FUNC4-Path 2

Figure 6 pWCET curves derived with MBPTA for the paths of FUNC1-FUNC4.

Table 1 Independence and identical distribution tests results for FUNC1-4.

	Path1		Path2	
Test	i.d.	ind.	i.d.	ind.
FUNC1	0.81	0.89	0.96	0.13
FUNC2	0.59	0.55	0.85	0.10
FUNC3	0.28	0.92	0.89	0.51
FUNC4	0.85	0.63	0.24	0.51

Table 2 WCET reduction between MBTA and MBPTA for cut-off probability 10^{-12}.

	Cut-off prob. 10^{-12}	
	Path1	Path2
FUNC1	9%	9%
FUNC2	5%	11%
FUNC3	15%	14%
FUNC4	11%	11%

In Figure 5 we observe the variability induced by SSR on the execution time of each observed path of the functions under analysis. This variability (jitter) across executions of different binaries is between 1.5–4.5%. Note that the observed variability is introduced only from the instruction and data cache. Recall that due to the deterministic nature of AURIX executions with the same binary and input do not have any variability. Moreover, since the execution times are collected per path and the same input is used, the variability does not come from the traversal of different paths. Hence, SSR can be used in order to provide insights to the industrial user about the potential impact of memory layout in its final integrated system. Further, from the above results we can conclude that SSR exposes cache jitter impact into execution times, whose probabilistic properties are examined in the next Section.

6.2 Independence and Identical Distribution Tests

We execute the case study 1,000 times to collect execution times, each with a different binary generated by SSR by using different random seeds. We flush caches and reload the executable across executions to have the same clean environment for each execution. This setup complies with MBPTA requirements [15]. The application of EVT – part of MBPTA – requires the execution times (data) provided as input to be statistically i.i.d. We test independence with the Ljung-Box test [6]. For identical distribution we use the two-sample Kolmogorov-Smirnov test [5]. In both cases we use a 5% significance level (a typical value for this type of tests). These means that i.i.d. is rejected only if the value for any of the tests is lower than 0.05. As shown in Table 1 all tests are passed in all setups, hence enabling the application of EVT.

6.3 pWCET Estimates

In Figure 6 we show pWCET estimates for every function and path. In all cases we successfully derived a pWCET curve, whose slope changes from case to case. The Complementary Cumulative Distribution Function (CCDF) of the observed execution times is also reported (shorter scattered green line). The fact that the CCDF is always below the pWCET curve confirms that in all cases our observations are tightly upperbounded by the pWCET.

In addition to the pWCET curves, we also show the maximum observed execution time, or MOET, for each FUNC and path (vertical line on the left), and a WCET estimate computed with MBTA using an engineering margin of 20% (vertical line on the right), commonly used in the real-time industry (e.g. avionics), to account for systems unknowns that could not be fully exercised during analysis such as the memory and cache layouts[4]. Interestingly, in all the cases the pWCET obtained for exceedance probability 10^{-15}, is below the estimates computed with current practice. In particular, considering a cut-off probability of 10^{-12} which has been shown to be an appropriate exceedance probability for hard-real time systems [18], MBPTA provides between 6–17% tighter results than MBTA (see Table 2). Despite the WCET reduction yielded by MBPTA, its most important benefit is that it provides scientific reasoning about the pWCET upper-bound instead of an engineering margin.

6.4 Summary

Cache jitter can be arguably hard to observe with standard MBTA approach. This is so because cache layout is (mostly) implicitly handled by the RTOS and the end user lacks evidence that despite it makes high number of runs, the space of potentially cache layouts that can arise at operation emerge. With software randomization instead, every new run a random cache layout is generated, which makes that as more runs are performed the space of potential cache layout are naturally covered. This not only provides a coverage criteria but also simplifies the analysis of program high execution times with measurement based probabilistic timing analysis. In the ACCS study randomization (i) exposes that variability is reduced; and (ii) enables MBPTA. This results in a proper handling of cache jitter in WCET estimates and further provides evidence for certification.

7 Conclusions

We presented our experience in using a software-based cache randomization techniques – that randomizes across image runs the position in memory where program's data and code are located, and MBPTA for an automotive cruise-control systems on the AURIX TC277 board. We have shown that in such deterministic architecture like the AURIX, caches can cause execution time variability. We further show that cache jitter can be handled with randomization – that makes it naturally emerge in the runs performed at analysis time – and probabilistic timing analysis that models the probability of high execution times.

References

1 Jaume Abella. Improving mbpta with the coefficient of variation. Technical Report UPC-DAC-RR-CAP-2015-11, UPC, July 2015.

[4] The unknowns also cover other factors such as the jitter due to different paths, that with MBPTA can be factored in with solutions like [16][28].

2 J. Abella et al. Heart of Gold: Making the improbable happen to extend coverage in probabilistic timing analysis. In *ECRTS*, 2014.

3 J. Abella et al. On the comparison of deterministic and probabilistic WCET estimation techniques. In *ECRTS*, 2014.

4 A. Baldovin et al. A time-composable operating system. In *WCET Analysis Workshop*, 2012.

5 S. Boslaugh and P.A. Watters. *Statistics in a nutshell.* O'Reilly Media, Inc., 2008.

6 G.E.P. Box and D.A. Pierce. Distribution of residual autocorrelations in autoregressive-integrated moving average time series models. *J. of the American Statistical Assoc.*, 1970.

7 L.P. Bradford and R. Quong. An empirical study on how program layout affects cache miss rates. *SIGMETRICS Perform. Eval.*, 3(27):28–42, 1999.

8 F.J. Cazorla et al. Upper-bounding program execution time with extreme value theory. In *WCET Workshop*, 2013.

9 L. Cucu-Grosjean et al. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS*, 2012.

10 C. Curtsinger and E.D. Berger. STABILIZER: statistically sound performance evaluation. In *ASPLOS*, 2013.

11 E. Mezzetti et al. Cache Optimisations for LEON Analyses (COLA). Technical report, ESA/ESTEC, 2011.

12 International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

13 L. Kosmidis et al. A cache design for probabilistically analysable real-time systems. In *DATE*, 2013.

14 L. Kosmidis et al. Probabilistic timing analysis on conventional cache designs. In *DATE*, 2013.

15 L. Kosmidis et al. Probabilistic timing analysis and its impact on processor architecture. In *DSD*, 2014.

16 L. Kosmidis et al. PUB: Path upper-bounding for measurement-based probabilistic timing analysis. In *ECRTS*, 2014.

17 S. Kotz et al. *Extreme value distributions: theory and applications.* World Scientific, 2000.

18 L. Kosmidis et al. Containing timing-related certification cost in automotive systems deploying complex hardware. *Best Paper Award, DAC*, 2014.

19 E. Mezzetti and T. Vardanega. A rapid cache-aware procedure positioning optimization to favor incremental development. In *RTAS*, 2013.

20 E. Mezzetti et al. Randomized caches can be pretty useful to hard real-time systems. *LITES*, 2(1), 2015.

21 S. Milutinovic et al. Modelling probabilistic cache representativeness in the presence of arbitrary access patterns. In *ISORC*, 2016.

22 N. C. Gloy et al. Procedure placement using temporal ordering information. In *MICRO*, 1997.

23 Eduardo Quinones, Emery D. Berger, Guillem Bernat, and Francisco J. Cazorla. Using Randomized Caches in Probabilistic Real-Time Systems. In *22nd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 129–138, 2009.

24 J. Reineke. Randomized caches considered harmful in hard real-time systems. *LITES*, 1(1), 2014.

25 T. Mytkowicz et al. Producing wrong data without doing anything obviously wrong! In *ASPLOS*, 2009.

26 `http://www.ehitex.de/application-kits/infineon/2531/aurix-application-kit-tc277-tft`. *AURIX Application Kit TC277 TFT.* hitex.

27 R. Wilhelm et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7:1–53, May 2008.

28 M. Ziccardi et al. EPC: extended path coverage for measurement-based probabilistic timing analysis. In *RTSS*, 2015.

29 M. Ziccardi et al. Software-enforced Interconnect Arbitration for COTS Multicores. In *WCET Analysis Workshop*, 2015.