Dynamic Software Randomisation: Lessons Learned From an Aerospace Case Study

Fabrice Cros[1], Leonidas Kosmidis[2,3], Franck Wartel[1], David Morales[2]
Jaume Abella[2], Ian Broster[4], Francisco J. Cazorla[2,5]
[1]Airbus Defence and Space, France [2]Barcelona Supercomputing Center (BSC), Spain
[3]Universitat Politècnica de Catalunya, Spain [4]Rapita Systems, UK [5]Spanish National Research Council (IIIA-CSIC), Spain

***Abstract*—Timing Validation and Verification (V&V) is an important step in real-time system design, in which a system's timing behaviour is assessed via Worst Case Execution Time (WCET) estimation and scheduling analysis. For WCET estimation, measurement-based timing analysis (MBTA) techniques are widely-used and well-established in industrial environments. However, the advent of complex processors makes it more difficult for the user to provide evidence that the software is tested under stress conditions *representative* of those at system operation. Measurement-Based Probabilistic Timing Analysis (MBPTA) is a variant of MBTA followed by the PROXIMA European Project that facilitates formulating this representativeness argument. MBPTA requires certain properties to be applicable, which can be obtained by selectively injecting randomisation in platform's timing behaviour via hardware or software means.**

In this paper, we assess the effectiveness of the PROXIMA's dynamic software randomisation (DSR) with a space industrial case study executed on a real unmodified hardware platform and an industrial operating system. We present the challenges faced in its development, in order to achieve MBPTA compliance and the lessons learned from this process. Our results, obtained using a commercial timing analysis tool, indicate that DSR does not impact the average performance of the application, while it enables the use of MBPTA. This results in tighter pWCET estimates compared to current industrial practice.

I. Introduction

Critical real-time systems – like those used in avionics, automotive or space – undergo a Validation and Verification (V&V) process[1] to ensure that their requirements provide the specified functionality and they are fulfilled. Timing V&V derives a timing bound for each software unit together with a scheduling of those software units so that system's timing requirements are fulfilled. Industry extensively relies on measurement-based timing-analysis (MBTA) [28], due to its simple applicability in industrial setups. The quality of the derived WCET estimates depends on user ability to control the conditions in which measurement are made so that they represent those expected at operation [2]. However, the use of advanced hardware features – to respond to the increasing performance demands in modern critical systems – challenges the applicability of MBTA[2]. In this paper we focus on the a challenge found in single core systems, brought by caches due to their jittery response time. Caches complicate providing evidence on representativeness, i.e., assessing whether the execution time conditions (such as the cache states under which the measurements used for analysis are collected) cover worst-case scenarios that can arise during actual system operation.

The PROXIMA project provides different means to deal with this challenge by introducing probabilistic timing analysis technologies, while it still remains attractive to industry by keeping its measurement-based nature. This timing analysis, known as Measurement-Based Probabilistic Timing Analysis (MBPTA)[9], has been shown to be competitive compared to conventional timing analysis methods: static [1] and measurement based[26][27]. MBPTA requires the platform to satisfy certain properties. In particular MBPTA-compliant systems [20] should exhibit control means to ensure that conditions at analysis match or upperbound those at operation; as well as a timing behaviour that can be modeled with independent and identically distributed (i.i.d.) random variables[2].

These properties can be achieved with both hardware and software solutions. The former includes the selective modification of the few jittery resources in the architecture that are hard to model [20], so that their timing behaviour becomes random. While required modifications are minimal, specialised hardware has high recurring costs and a long adoption horizon by the real-time industries which are quite conservative. This has motivated software randomisation techniques as a way to achieve MBPTA compatibility without the need for specialised hardware. This provides a fast path for PROXIMA technology to be available to industry right away for testing over Commercial off-the-shelf (COTS) processors, with an overall goal to be finally adopted in real products.

Both static[19][16][15] and dynamic software randomisation (DSR) [18] share the same basic principle: memory objects (functions, stack frames etc) are randomly placed in the memory, thus affecting the mapping of those elements in the cache. As a result, the execution time variability generated by cache can be analysed with MBPTA. Since DSR has reached a high *technology readiness level*, it is the randomisation version of choice for evaluation in this paper. Despite that DSR has been preliminarily assessed in the past with an avionics case study[27] as well as compared with hardware randomisation, that work was not performed on a real industrial environment, but on a research prototype using a research RTOS[4] and a simulation infrastructure.

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

In this paper, we evaluate PROXIMA's DSR technology with a space case study in an industrial setup. We implemented DSR as part of the compilation toolchain and a runtime system. In particular as specific compiler pass on top of the LLVM (http://llvm.org/) toolchain. The application under analysis is randomised using DSR and executed over an industrial RTOS on top of a real processor used in the space domain. To our knowledge, the first evaluation of DSR in a real setup, using a Space case study on an FPGA platform based on LEON3 [25].

Our evaluation assesses experimentally whether the statistical timing properties for the application of MBPTA can be obtained with DSR on the target board. Further, we assess the accuracy of MBPTA w.r.t. current practice measurement-based deterministic (i.e. not probabilistic) techniques. To that end we perform the timing analysis evaluation with a commercial timing analysis tool [23] that has been properly enhanced to support probabilistic analysis. From our results with a Space case study based on a mixed criticality control and processing application of an on-board scientific instrument, we concluded: a) DSR does not impact the average performance of the application, but improves it in some cases. b) DSR provides the probabilistic timing behaviour required in the platform to apply MBPTA. And c) the obtained WCET estimates with DSR are tighter compared to the typical 20% margin used in current industrial practice, while MBPTA provides more solid argumentation on the validity of the derived WCET estimates.

This paper is structured as follows: Section II introduces MBPTA. Section III explains the changes in the compilation and runtime system to provide MBPTA properties using dynamic software randomisation. Section IV introduces the space case study used in this work. Section V presents the timing analysis infrastructure we use and its integration and adaptation to our platform. Section VI presents the main results obtained and conclusions provided in Section VII.

II. Background

Measurement-based techniques comprise an analysis phase, when verification of the timing behaviour takes place; and the operation phase when the system becomes operational. The goal of measurement-techniques is to derive WCET estimates, from execution runs of the program performed at analysis, that hold valid during the operation of the system. This requires creating an argument that the execution conditions of the experiments at analysis capture those worst-case conditions that can occur at operation [13].

The quality of the derived WCET estimates of current-practice measurement-based analysis (MBDTA) lies on the user ability to design stressful test scenarios, in which the application is run under similar worst-case conditions to those that can arise during system operation. In general, it is impossible for the user to cover all the input space of the program. For some sources of jitter there are tools that help the user determining, for instance, the degree of path coverage in its program and hence allowing the end user to create more input vectors until certain degree of coverage is reached (e.g. MC/DC coverage). In contrast to execution path coverage, which we define as a high-level source of jitter (*hlsoj*), the use of more complex hardware and software in future real-time systems introduces low-level sources of jitter (*llsoj*) [14].

In general, the user cannot exercise the necessary control to force that in the tests worst-case conditions for each *llsoj* are captured. MBPTA attack this problem by deploying statistical analysis through Extreme Value Theory (EVT) [21] and the selective injection of randomisation in the timing behaviour of certain resources [20]. With EVT, MBPTA is able to derive and upperbound the probability that bad behaviour of several of the *llsoj*, whose impact has been captured in the analysis time runs, are triggered in the same run, leading to a high execution time. Randomisation makes events affecting execution time, including those representing bad behaviour of the *llsoj* have a probability of appearance. This way, those events can be probabilistically guaranteed to be captured in the measurements (tests) performed during analysis provided a sufficient number of experiments is performed. Note that not all *llsoj* are handled by MBPTA using randomisation, but only the ones difficult to model and/or with high jitter [20] such as the cache related ones examined in this paper. Other *llsoj* are forced to work in their worst latency during analysis. In this paper we analyse one of the most prominent examples of *llsoj*, the cache. We focus on a single-core setup and do not take into account activities related to i/o devices and alike, i.e. our main focus is at the chip level.

MBPTA [9][26][27] produces a pWCET distribution that describes the highest probability (e.g., 10^{-15}) at which one instance of a program may exceed the corresponding execution time bound. The particular exceedance probability, and the corresponding time budget, to choose is that deemed as sufficiently low based on i) the criticality level of the application under analysis; and ii) the corresponding safety standard.

The advent of more autonomous satellite operations increases the complexity of on-board software. Software timing analysis can be simplified with *incremental software integration and qualification* [12] in which different modules follow an independent development process and WCET estimates derived during early design phases should hold valid across integrations. However, caches makes that the relative cache offset of software unit's can change across integrations. This might invalidate the WCET estimates derived for already integrated software, incurring the cost of re-assessing the WCET estimate of already-integrated software. Further this exposes users to late detection of timing violations with high associated costs. DSR breaks the relation between the memory position of code/data and the cache sets they are assigned to. DSR randomly changes this mapping across different executions, hence factoring in the potential impact of different cache alignments caused by future integration. This has enormous advantages in enabling incremental software integration - and its benefits - in the presence of caches.

III. Achieving MBPTA Compliance via Software

Software randomisation handles the jitter caused by caches in COTS platforms in an MBPTA conformant manner. Soft-

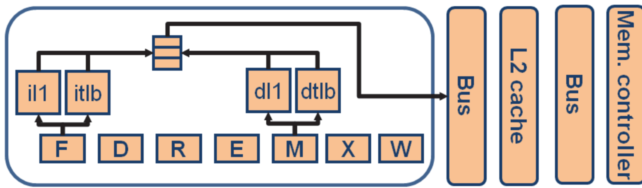

Fig. 1. Target Architecture of PROXIMA's LEON 3-based platform.

ware randomisation techniques [18][19] achieve this effect in an indirect way. In the case of the caches , the placement function is fixed and hence cannot be modified (i.e. randomised). Software randomisation exploits the direct relation between the memory address of the object in main memory and its location (set) in the cache. Based on this basic principle, software randomisation changes across runs the location in main memory for each object in a random way, achieving the same effect as with a hardware randomised cache. The main difference though, is in the granularity of the randomisation: while a cache with random placement randomises each cache line, software randomisation techniques randomise entire memory object, whose size might be smaller or usually bigger than a cache line. Hence, software randomisation is able to alter *inter-object* [18] conflicts, while *intra-object* conflicts are retained. The changes in *inter-object* conflicts, due to the different memory placements have an effect also in the LRU stack of their respective cache lines, since now the time between accesses in the same cache line are also modified. For this reason, a similar effect to a random-replacement policy is achieved between runs [18].

Software randomisation techniques are classified into *dynamic* and *static* ones. Dynamic Software Randomisation [18] (DSR) changes the location of memory objects at runtime during system operation, while static implementations [19] are based on pre-compiled binaries with different memory object placements. Independently of the static or dynamic implementation, both solutions are equivalent in enabling MBPTA and have been successfully assessed with industrial case studies on top of timing simulation environments [27][19]. In particular, DSR has been applied to two avionics case study applications [27], while a static variant has been used in the automotive domain [19] as it suits better its requirements. For our space case study we have opted for the dynamic variant following the example of [27], since the avionics and the space domains have similar requirements and the DSR technology in PROXIMA has reached a higher industry-readiness level.

A. Target COTS Platform Description

We use a LEON3 [25] based platform implemented on an FPGA. The organisation of our platform is shown in Figure 1. LEON3 processors comprise first level instruction (IL1) and data (DL1) caches, with the DL1 implementing a write-through no write-allocate policy; a bus that propagates DL1 and IL1 misses to the unified L2 cache. L2 misses are propagated to the DRAM memory controller, see Figure 1. IL1 and DL1 are 16KB with 4-way set-associative caches, while the L2 is 32KB direct-mapped with write-back policy. TLBs with 64 entries for instructions and data are also included.

The processor implements a pipelined architecture comprising seven stages: fetch (F), decode (D), register access (R), execution of non-memory operations (E), DL1 access (M), Exceptions (X) and write back (W). As a SPARCv8 processor, LEON3 features a microarchitecture based on *register windows*. The register file comprises 8 register windows. The floating-point unit takes a variable latency depending on the particular values operated, with a jitter of up to 3 cycles.

B. DSR on the LEON 3: Challenges and Solutions

DSR is implemented combining a compiler pass and a runtime system, based on Stabiliser (a software randomisation tool proposed for unbiased performance evaluation of high-performance software [10]). The compiler pass is implemented on top of the LLVM toolchain and it generates metadata required for the relocation of memory objects at runtime, as well as modifies the code of the compiled software to transparently access these metadata. The runtime system takes care of the actual memory object moving at runtime. The original version of the software randomisation framework supports x86, x86-64 and PowerPC variants. Since our LEON3 target platform is based on a different architecture (SPARC v8), we have ported the framework to this environment. This has been a challenging task due to the peculiarities of the target architecture. In the following we explain in detail how the original toolchain works, which modifications we have performed to our port of the library, the problems we encountered in this process and how we have overcome them. In particular DSR randomises functions and their stack frames, in random memory locations on each invocation.

B.1. *Code Randomisation*

Relocation Scheme. Code randomisation comprises moving a function to a new memory location selected randomly. Function relocation can take place either in an *eager* or in a *lazy* manner. Eager implementation requires all function relocations to take place before the program execution, while lazy one relocates only the functions used by the software, at the moment of their first use. However, lazy relocation complicates the estimation of the worst-case memory consumption as well as the WCET, both of which are necessary in critical real-time systems. For these reasons, despite the original version supports lazy relocation, in our porting of the library we selected to implement an eager relocation scheme.

Cache Consistency: SPARC does not provide hardware consistency between the instruction and data caches as opposed to x86. Therefore, after relocating a function, the updated data cache lines first need to be written in the memory before they can be fetched for execution. This is not a problem for DL1 which is write through, but it is for the write-back unified L2. In addition to that, any updated IL1 or L2 entry corresponding to the old location, need to be invalidated to guarantee the memory consistency across the different caches. To achieve this, we have implemented a fully SPARC v8 compliant address invalidation routine.

B.2. *Stack Randomisation*

Previous uses of stack randomisation in the real-time domain used a scheme in which each stack frame was allocated dynamically at function entry [18][19]. We opted for the solution described in [10], in which stack randomisation is implemented by making the stack non-contiguous. The compiler pass inserts code in each function's prologue and epilogue, which adjusts the stack frame with an offset (ranging between 0 and the way size of the cache) read from a table assigned to this function, which is part of the metadata created during compilation. This table is initialised in the program start-up with a random positive value, which randomises the location of the stack frame's initial address.

This implementation appeared as one of the most challenging parts of porting software randomisation in SPARC architecture due to the presence of the register window. We had to increase the stack pointer by the random offset within the SAVE instruction of the caller (which invokes the register rotation), and restore it within the RESTORE instruction at function return. This way we ensure that stack pointer is modified atomically and remains always valid. Note that the random offset used to adjust the stack pointer must be a multiple of 8 in order to keep the stack pointer aligned in double word boundaries, as the architectural specification of SPARC requires.

B.3. *Software Random Number Generator*

The position of the software objects is selected randomly inside memory chunks obtained using a memory allocator based on HeapLayers[11]. In particular, the starting offset is between zero and the maximum way size to ensure that the memory object can be mapped in any cache line inside a cache way. The random numbers required are generated using a software implementation of the Multiply-With-Carry (MWC) [22] PRNG. The quality of this PRNG in terms of period is shown in [3] to be sufficient, as for the LFSR proposed in the same work. However, while LFSR can be efficiently implemented in hardware, the MWC is the simplest one to implement in software. Therefore, the random source used for DSR is the MWC PRNG.

B.4. *Second-Level Unified Cache*

All software randomisation works so far, have considered a single cache level. As described in the previous point, the random offset of the memory object need to be up to the size of the cache way, so previous works set this number according to the L1 size. However, our target platform features also a second level unified cache. For this reason, we set the offset equal to the L2 cache way size, in order to randomise also the cache layout of the second level cache. Since the L1 way size is multiple of the L2, this achieves also cache layout randomisation of the first level caches.

Unified L2 caches create complex interactions between instruction and data objects. This has causes L2 caches to be usually disabled in critical real-time systems. MBPTA has been shown to be effective in the analysis of arbitrary levels of hardware time-randomised caches [17]. Since by our offset selection we randomise the cache layout of all cache levels, the entire memory hierarchy becomes time-randomised similar to the hardware randomisation case, so MBPTA can be also used to analyse it. In addition to the obvious advantage that DSR offers MBPTA analysability, the fact that it breaks the complex interactions between instruction and data objects reduces the possibility of cache risk patterns, and this way it can result in performance improvements over the non-software randomised case, as we show in the Evaluation Section.

B.5. *TLBs*

The random memory allocations performed by [5] use two separate memory pools for code and data. Each pool is sufficiently large and is comprised by a diverse set of pages, which effectively randomises both Instruction and Data TLBs, as in the case of the hardware randomised platform.

IV. Space Case Study

We use mixed-criticality space application controlling an integrated active optics instrument for space telescopes. It consists of both a data processing task which computes the wave front error using data from a collection of sensors (low criticality) and a control task which elaborates commands to the actuators controlling mirror displacements and is in charge of the interface with the rest of the spacecraft (high criticality).

The control task is robust to functional misbehaviour of the image processing application. However the temporal interferences caused by a malfunction in the image processing task could affect the timing of the high criticality control task.

The image processing computes the passive deformation of a mirror in a satellite instrument and comprises 2 phases. During the former, a coarse offset is computed and while during the latter the offset is computed in a finer granularity. Image processing is both CPU intensive (it executes significant amount of floating point operations) and memory intensive (it performs many reads and writes to the pixels from the lenses).

The input vectors of the application are composed of 12x12 array of lenses of 34x34 pixels each. Not every lens is processed, but only the most lightened ones which are around 70% of the total lenses. This creates a variation in the duration of the computation directly linked to the input data, which makes the timing analysis of the application challenging.

Each task is self-contained and is implemented in a separate partition of the underlying hypervisor RTOS, PikeOS Native, to ensure spatial and temporal isolation. The control is invoked periodically every 1 second while the processing partition every 100ms. In order to ensure that in each period the partition executions start with the same initial hardware state, we use PikeOS features to automatically flush instruction and data caches. Moreover, PikeOS is configured to prevent preemptions during the execution of the application. DSR has been successfully applied in both tasks, however only the timing of the critical task is captured and analysed using MBPTA. Whether it is sufficient to randomise only the critical one is left for future work. For the measurement collection during the MBPTA analysis phase, after the end of each

partition execution, the partition is rebooted through software means to guarantee that each execution starts with a different memory layout.

V. Timing Analysis

MBPTA technology has been integrated into Rapita Verification Suite [23] (RVS), a commercial tool used in domains such as avionics or automotive. RVS executes the UoA on the target board, extracting a trace of execution times at instrumentation point boundaries which is then processed to extract execution times, that are then processed by MBPTA.

We define as unit of analysis (UoA) the execution time of the control task which is repeated in each partition invocation.

GRMON debug monitor for LEON processors [8] communicates with the LEON debug support unit (DSU) and allows non-intrusive debugging of the complete target system. We use GRMON to initialise the FPGA and load the binary file to execute. The application code is instrumented by RVS at UoA granularity, recording the execution time of entry and exit points by reading the value of the execution time register, which provides the cycle count.

While the instrumented program runs, time-stamps are stored in a buffer allocated on a second memory bank to avoid interference with the application, and results are dumped to a file by GRMON through the Ethernet interface after the program execution finishes. This binary trace is processed and converted to a format which is readable by RVS.

VI. Evaluation

We assess the effectiveness of the DSR technology to expose cache-alignment variability (jitter) in the measured execution times and show the pWCET estimates obtained by processing those measurements with MBPTA. We start by analysing the default platform that contains no randomisation source. In the performed analysis the effect of the jittery floating point operations is not taken into account. The reason is their expected low impact since the space application executes less than 2% of floating point instructions. Further, only two types of those instructions have a maximum jittery of 3 cycles.

Current practice for WCET estimation include measuring the execution time of the supposed worst-case scenarios and adding an engineering margin to the highest observed value. The engineering margin is computed depending on the previous experience with the software under analysis and the confidence that the validation expert has on the elaboration of the worst-case scenarios. A typical margin for relatively simple single-core processors is 20%, which we use as reference.

Figure 2 shows the minimum, maximum (MOET) and average execution times obtained with and without DSR for the critical task. Interestingly the results with DSR are quite similar to the ones obtained without DSR. In fact, the maximum observed time is a little bit smaller.

TABLE I
Performance counter readings for the control task

	icmiss	dcmiss	L2miss	FPU	Instr
No Rand	126-127	2088	402-558	3504	163800
Sw Rand	154	2129-2131	398-555	3504	166748

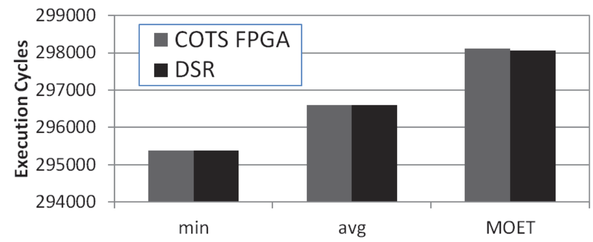

Fig. 2. Average Performance Comparison between original and software randomised version of the space application.

In Table I we can see values from performance counters for both setups. DSR has very small impact on number of instructions for the software analysed (<2%). Recall that DSR redirects each function call to a new location and adjusts each stack frame, therefore this has a necessary overhead. Our space application has a small number of function calls in the UoA code compared to the number of the total instructions executed, therefore this overhead is negligible. The number of L1 misses for both the instruction and data cache increases with DSR. However, the L2 miss ratio with DSR (computed as the ratio of the number of L2 missed and the sum of L1 instruction and data misses which represent the total number of L2 accesses) is 1% lower. In particular the code with DSR exhibits 17-24% miss ratio, versus 18-25% in the COTS configuration. This small difference results in slightly lower execution times, which in turn yields a lower MOET as presented in the previous figure. The reason for this is that the software in the COTS version has a bad and rare cache layout for the L2. Hence, DSR results in better L2 cache layouts and therefore even in the presence of the 2% software randomisation overhead, this long MOET is not observed.

Fulfilling the i.i.d properties. The application of MBPTA – and in particular its EVT component – requires the execution times (data) provided as input to be modellable with i.i.d. random variables. We test independence with the Ljung-Box test [7] and a 5% significance level (a typical value for this type of tests). For identical distribution we use the two-sample Kolmogorov-Smirnov test [6] also with a 5% significance level. These means that i.i.d. is rejected only if the value for any of the tests is lower than 0.05. For our experiments we obtain values above 0.05. meaning that both tests are passed, hence enabling the application of EVT.

pWCET estimates. Figure 3 provides a screenshot of the RVS Viewer which shows the pWCET curve obtained by processing the measurements of the software randomised application using MBPTA. In the X-axis we have the execution time while on the Y-axis we observe probabilities in logarithmic scale. We observe that the pWCET prediction, straight line, tightly upper-bounds the measured execution times values (MET).

The pWCET estimates for DSR are close to the MOET and well under the value 20% margin. In particular, the pWCET estimation at 10^{-15} is only 0.2% higher than the MOET observed with DSR enabled. This means that the value computed by MBPTA is much closer to the observed values than what can be achieved with a 20% margin, and therefore

Fig. 3. pWCET curve of the DSR version of the application.

the result is less pessimistic. When this is compared with the current industrial practice adding an engineering margin of 20% over the MOET of the non-randomised application, it results in a 19.6% tighter WCET prediction. Furthermore, thanks to MBPTA, the values obtained can be used with a higher degree of confidence than adding a simple margin.

It is worth noting that the current industrial practice (and also the academic one) finds difficulties to analyse unified multilevel caches, since the complex interactions between instruction and data in the second level cache can cause abrupt time changes. Randomisation allows to explore large cache memory layout counts, which in combination with MBPTA's upperbounding using EVT reduces the risk of observing at system operation a cache layout with a worse behaviour that the ones exercised at analysis.

VII. Conclusions and Future Work

We have summarised the foundations behind the dynamic software randomisation and showed the challenges in implementing it in real setup: space application, commercial RTOS and a FPGA board. Despite its runtime overhead, we show that in the particular case study the software randomisation results in lower WCET estimates than with current industrial practice, while it helps providing evidence on the cache-generated jitter. We provided experimentally evidence that MBPTA can be applied and that i.i.d. properties are passed.

As part of our future work, and based on the promising results obtained with our current setup, we plan to extend our work towards: (i) using MBPTA-compatible methods to achieve full path coverage [29] and (ii) dealing with COTS multicore contention-related jitter.

Acknowledgments

The research leading to these results has received funding from the European Community's FP7 [FP7/2007-2013] under the PROXIMA Project (www.proxima-project.eu), grant agreement no 611085. This work has also been partially supported by the Spanish Ministry of Science and Innovation under 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 postdoctoral fellowship number RYC-2013-14717.

References

[1] J. Abella et al. On the comparison of deterministic and probabilistic wcet estimation techniques. In *ECRTS*, 2014.

[2] J. Abella et al. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *SIES*, 2015.

[3] I. Agirre et al. IEC-61508 SIL 3 compliant pseudo-random number generators for probabilistic timing analysis. In *DSD*, 2015.

[4] A. Baldovin, E. Mezzetti, and T. Vardanega. A time-composable operating system. *WCET Workshop*, 2012.

[5] E. D. Berger and B.G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *PLDI*, 2006.

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

[7] G. E. P. Box and D. A. Pierce. Distribution of residual autocorrelations in autoregressive-integrated moving average time series models. *Journal of the American Statistical Association*, 1970.

[8] Cobham Gaisler. GRMon. http://www.gaisler.com/index.php/products/debug-tools/grmon.

[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*, pages 219–228, 2013.

[11] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *PLDI*, 2001.

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

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

[14] F. J. Cazorla et al. PROXIMA: Improving measurement-based timing analysis through randomisation and probabilistic analysis. In *DSD*, 2016.

[15] L. Kosmidis, D. Compagnin, D. Morales, E. Mezzetti, E. Quiñones, J. Abella, Tullio Vardanega, and F.J. Cazorla. Measurement-based timing analysis of the aurix caches. In *WCET*, 2016.

[16] L. Kosmidis, R. Vargas, D. Morales, E. Quiñones, J. Abella, and F. J. Cazorla. TASA: Toolchain Agnostic Software Randomisation for Critical Real-Time Systems. In *ICCAD*, 2016.

[17] L. Kosmidis et al. Multi-level unified caches for probabilistically time analysable real-time systems. In *RTSS*, 2013.

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

[19] L. Kosmidis et al. Containing timing-related certification cost in automotive systems deploying complex hardware. In *DAC*, 2014.

[20] L. Kosmidis et al. Measurement-based probabilistic timing analysis and its impact on processor architecture. In *Euromicro DSD*, 2014.

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

[22] G. Marsaglia and A. Zaman. A new class of random number generators. *Annals of Applied Probability*, 1(3):462–480, 1991.

[23] Rapita Systems Ltd. Rapita Verification Suite. http://www.rapitasystems.com/products/rvs. Accessed Jan 2015.

[24] L. Santinelli et al. On the sustainability of the extreme value theory for WCET estimation. In *WCET Workshop*, 2014.

[25] http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=13&Itemid=53. *Leon3 Processor*. Cobham Gaisler.

[26] F. Wartel et al. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *SIES*, 2013.

[27] F. Wartel et al. Timing analysis of an avionics case study on complex hardware/software platforms. In *DATE*, 2015.

[28] 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.

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