Industrial Experiences with Resource Management under Software Randomization in ARINC653 Avionics Environments

Leonidas Kosmidis
Barcelona Supercomputing Center (BSC)

Cristian Maxim*
Airbus S.A.S

Victor Jegu
Airbus S.A.S

Francis Vatrinet
SYSGO

Francisco J. Cazorla
Barcelona Supercomputing Center (BSC)
Spanish National Research Council (IIIA-CSIC)

ABSTRACT

Injecting randomization in different layers of the computing platform has been shown beneficial for security, resilience to software bugs and timing analysis. In this paper, with focus on the latter, we show our experience regarding memory and timing resource management when software randomization techniques are applied to one of the most stringent industrial environments, ARINC653-based avionics. We describe the challenges in this task, we propose a set of solutions and present the results obtained for two commercial avionics applications, executed on COTS hardware and RTOS.

CCS CONCEPTS

• **Applied computing** → **Avionics**; **Avionics**; • **Computer systems organization** → **Embedded and cyber-physical systems**; **Embedded software**; **Real-time systems**; **Real-time operating systems**; **Real-time system specification**; • **Software and its engineering** → **Allocation / deallocation strategies**; **Embedded software**; **Real-time systems software**;

ACM Reference Format:
Leonidas Kosmidis, Cristian Maxim, Victor Jegu, Francis Vatrinet, and Francisco J. Cazorla. 2018. Industrial Experiences with Resource Management under Software Randomization in ARINC653 Avionics Environments. In *IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN (ICCAD '18), November 5–8, 2018, San Diego, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3240765.3240818

1 INTRODUCTION

The increasing performance demand in modern critical real-time systems has seeded the development of new tools to tame the resulted complexity in both their hardware and software. In the worst case timing analysis, which is crucial for this class of systems, Measurement-Based Probabilistic Timing Analysis (MBPTA) [1] has been successful, showing competitive results with current industrial practice [2][3][4].

Despite its promising features, MBPTA requires the presence of specific properties in the underlying platform to be applied [5]. That is a) events affecting the execution time need to have an equal or higher impact during measurement collection (analysis) compared to the system operation, b) the execution times of the platform to be modelled by an independent and identically distributed (i.i.d.) random variable, so that they can be processed by Extreme Value Theory (EVT) to obtain a probabilistic WCET curve. Both properties can be obtained by randomization in the system, either at hardware [6] or software level [7].

Recent works in the literature have demonstrated the industrial viability of deriving pWCET curves estimated with MBPTA using both hardware and software approaches, in realistic setups including real-time operating systems (RTOSes) and applications from different critical domains such as aerospace [3][4][8], railway [9] and automotive [10]. All those studies were focused only on the timing analysis of the system and in particular on the details about how to provide MBPTA's requirements in the measurement collection in order to ensure a trustworthy pWCET and about the implementation of the hardware or the software in charge of providing the required properties.

The application of software randomization, unlike hardware randomization is not transparent to the application and to the RTOS, but it is deemed as the more suitable option for adoption in the avionics domain. However, in the previous studies the practical implications of applying software randomization in an industrial critical real-time system have not been covered, especially in the avionics domain which requires strict provisioning of memory and timing resources. In this paper we provide our useful insights to industrial users of MBPTA, presenting our experience with the application of dynamic software randomization on avionics software executed on a commercial avionics Real-Time Operating System (RTOS) and a COTS platform for the first time. In particular, the contributions of this paper are the following: a) we describe the challenges that software randomization introduces to the avionics domain with respect to the integration of software randomized applications within an ARINC653 operating system, b) we propose solutions to overcome these challenges, namely the computation of upperbounds for the applications' memory requirements and c)

*Also with Inria, Paris at the time of performing this work. Currently only with Inria..

we demonstrate the feasibility of software randomization in a complete industrial avionics setup, presenting results about resource management in this context.

2 SOFTWARE RANDOMIZATION

At software level, randomization is applied in the form of software randomization. Software randomization places the various software elements (code, stack, global data) in randomly different memory positions, effectively modifying the application's memory layout. Since in both caches and TLBs – the two hardware structures with higher impact in the execution time – the physical address determines the various element's placement and replacement, software randomization achieves both MBPTA's requirements. In particular, the events that affect the execution time (cache and TLB misses) obtain a random behavior which is the same at both analysis and system operation, and at the same time yield randomly distributed execution times, modelled by an i.i.d. variable.

Software randomization can have a dynamic [7] or static form [11], which identifies whether the placement of the program's elements at system operation takes place on-line (before program execution) or off-line. Both solutions have been demonstrated to work in industrial setups, with the static variant being successful in the automotive domain due to hardware limitations of automotive microcontrollers [10], while the dynamic implementation has been applied in the space domain. In addition to full industrial studies with COTS hardware [4], the dynamic variant which has been introduced much earlier in the literature has been also explored in early limited industrial experiments with simulators and research oriented Real-Time Operating Systems [2]. For this reason, dynamic software randomization (DSR) is considered to have reached a higher Technology Transfer Level (TRL) and therefore it is selected as the basis of our study.

3 AVIONICS ENVIRONMENT

The avionics domain, although it shares some common requirements with other critical domains such as the automotive and space, it is the one with the strictest requirements. This creates additional challenges in the application of software randomization in this context, compared to its previous industrial applications [10][4]. In fact, the authors of [4] present the required modifications that dynamic software randomization had to undergo in order to be applied in an industrial COTS environment, as well as how the requirements of MBPTA were satisfied in their platform so that they could compute a valid pWCET of their space application. In contrast, our work is mainly focused on the particular challenges which arise in an avionics environment from the use of software randomization without any modifications. In this aspect, our primary goal is to identify the implications of the application of software randomization in the avionics domain and show how they can be addressed, so that it can be applied.

At the core of an avionics system there is an ARINC653 [12] RTOS, which provides the required spatial and temporal isolation between software elements.

The avionics software is organized in *partitions* which are composed by ARINC653 processes. Partitions are software entities which guarantee time and space isolation among them, a requirement for critical systems in order to ensure the robustness of the system both in avoiding deadline overruns as well as to provide fault containment from potential memory violations. Processes are execution units within a partition, which share the partition's resources. In order to obtain memory isolation, each partition has its own address space, thus leveraging the protection offered by the target processor's Memory Management Unit (MMU), while their processes execute in the same address space. In that aspect, partitions are similar to processes in Unix systems, while ARINC653 processes are equivalent to POSIX threads. However, the main difference of ARINC653 is that both partitions and processes have strict resource requirements provided by the system integrator in XML form. The RTOS ensures that these requirements are respected, otherwise countermeasures are applied. Among the requirements which are specified for each partition and process are memory and timing requirements.

Regarding memory requirements, the starting address and the maximum size of each memory segment used by the software is specified. The system integrator can map the desired memory segment in particular virtual addresses and grant specific access rights on each one, such whether it can be read, written or executed as well as if it can use the cache or it is forced to bypass it.

As a part of this process, the maximum stack size of each process and the cumulative stack size of all partition's processes need to be specified. The RTOS ensures that this limit is not violated, by mapping an invalid page (*guard* page) beyond the permitted virtual address, which results in a memory exception if it is accessed. The timing properties of partitions and processes are also provided in both a cumulative and individual basis. Those include their periods and their deadlines. Finally, it is important to note that avionics RTOSes do not provide dynamic memory allocation features since in the general case, programs relying on such mechanisms are hard to reason about their maximum memory consumption or worst case timing.

From the above discussion, it is clear that using any piece of software in an avionics environment requires detailed information about its memory and timing resource consumption.

4 CHALLENGES WITH DSR IN AVIONICS

Using DSR in an avionics environment presents a set of challenges due to its implications on the memory and timing characteristics of the randomized application. In particular, in order to modify the memory layout of the program, dynamic memory allocation is employed. Moreover, since the placement of software elements is performed in a random way, the exact size of each memory segment varies from execution to execution. Finally, changing the memory layout involves an execution time overhead, while the different cache layouts create the necessary variability in execution time required by MBPTA. Below, we examine in detail each of these problems and sketch the solutions we implemented.

Dynamic Memory Allocation: DSR is based on dynamic memory allocation for modifying the code and data layout of the randomized software. In the absence of this functionality in ARINC653, we use the RTOS to map a memory region in the virtual address space of the randomized partition. This solution is completely transparent

Algorithm 1 Determine an optimal memory code pool size requirement for a software randomized avionics system

for all $p_i \in SoftwareRandomisedPartitions$ **do**
  $UpperboundCodePoolSize_i = CodePoolSize_i = CodeSize_i$
  $UpperboundPartitionSize_i = \sum_{n \in \{Code,Data\}} nSize_i + nPoolSize_i + StackSize_i$
  while p_i cannot start **do**
    $UpperboundCodePoolSize_i = UpperboundCodePoolSize_i * 2$
    $UpperboundPartitionSize_i = UpperboundCodePoolSize_i + DataPoolSize_i + \sum_{n \in \{Code,Data\}} nSize_i$
  end while
  $LowerboundCodePoolSize_i = UpperboundCodePoolSize_i/2$
  while $(UpperboundCodePoolSize_i - LowerboundCodePoolSize_i) > 4KB$ **do**
    $NewCodePoolSize_i = UpperboundCodePoolSize_i - LowerboundCodePoolSize_i$
    if p_i cannot start **then**
      $LowerboundCodePoolSize_i = NewCodePoolSize_i$
    else
      $UpperboundCodePoolSize_i = NewCodePoolSize_i$
    end if
  end while
  $CodePoolSize_i = UpperboundCodePoolSize_i$
  $NewPartitionSize_i = \sum_{n \in \{Code,Data\}} nSize_i + nPoolSize_i + StackSize_i$
end for

to software randomization runtime and the underlying application, because DSR only requires the starting address of the memory which it can use as a pool for its internal randomized memory allocator. We introduce two memory regions, one for code and one for data, to provide the required protection between different segments within the same partition. We use a different pair of memory regions for each partition, so that the spatial isolation between partitions is preserved.

Determining Memory Pool Size: The previous solution requires to know the maximum size of each memory region mapped to the partition address space. In order to achieve this, we developed an iterative process which allows to derive those sizes.

Code randomization allocates for each function a piece of memory with size S_c, so that the code of the function can be mapped in any possible cache sets of its cache memory hierarchy. Consequently, S_c is platform dependent, equal to two times the size of the target processor's last level cache way. Therefore, the maximum size of the code pool is a function of the system's cache and the number of functions in the software. Our DSR system implements an eager code relocation scheme similar to [4], which performs all function relocations before starting the program. We take advantage of this feature, which ensures that if the code memory pool is not sufficient, the application of a partition will not be able to start. Therefore in order to determine the upperbound of the memory pool, we employ the exponential search algorithm, which is an efficient algorithm for unbounded searching problems [13] like ours. Algorithm 1 shows our implementation of the algorithm. We perform a series of testing runs starting with the code size limit of the partition before software randomization and doubling the memory requirement until the upperbound is found, when the application can start successfully. In each step we also update the total memory requirement of the partition, which is equal to the addition of all memory requirements. In order to avoid memory overprovisioning, we continue to narrow down the memory requirement by performing binary search in the opposite direction, until we find the minimum upperbound within a 4KB region, which is the granularity that memory resources are specified in the RTOS. Note that the procedure converges very fast, since the software fails quickly (before the application is started) in case of insufficient allocated memory.

Data randomization is performed in a similar way, therefore the followed procedure is the same for the data pool. Note that both iterative searches can be combined in the same test campaign, because in the case of insufficient memory, the memory violation address is reported. Therefore, with a careful selection of the virtual address at which each pool is mapped in the partition's address space, it is possible to distinguish whether it was caused by insufficient code or data memory.

Determining Stack Size: Instead of dynamically allocating each stack frame, the stack frame is randomly increased by a number between 0 and S_s in order to be mapped in any potential cache set of its cache hierarchy. Similar to the code case, S_s is processor dependent, equal to the target processor's cache set way of its last level cache. Although maximum stack size can be computed with commercial tools, there is no support for dynamic software randomization. Moreover, this procedure is complicated from the fact that not all the software inside a partition may be software randomized eg. libraries.

For this reason, we used the same iterative method described earlier. We modify the stack randomization used in our test campaign, so that it always increases each stack frame by its maximum size (S_s). This is equivalent to the eager relocation for the code, which ensures that the maximum bound is found for a given software. Next the iterative search is performed for each process inside each partition. Finally, once the the maximum stack size of each process is determined, their cumulative size is the maximum stack size for the partition. Note that in this iterative process, the partition stack must be always larger than the summation of the process stacks.

Timing: The runtime overhead of DSR is small compared to the execution time of the application (few instructions per function),

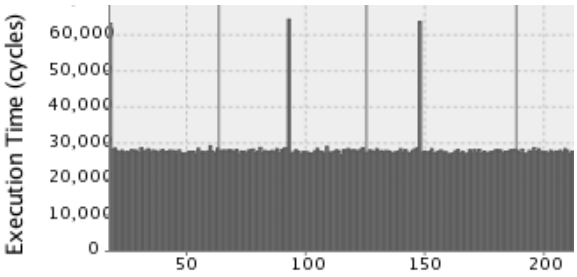


Figure 1: Example High Execution time observed in a MIF of APP1 without software randomization on P4080. The plot shows the first 250 samples of the 1000 collected, without software randomization.

therefore it does not affect its period or the deadline. However, the function relocation before the start of the application can have significant relative overhead. For this reason, we perform it at the partition boot time from a process with unlimited deadline, which creates the partition threads once the relocation is complete.

5 RESULTS

In our study we applied software randomization in two real avionics applications (APP1 and APP2 for confidentiality reasons) with high-criticality, hosted on IMA Line-Replaceable Module (LRM). The two applications have diverse resource requirements and each one is deployed in a separate partition. The APP1 has a smaller memory footprint than APP2, a little less than 1MB and it consists of a single ARINC653 process, while APP2 is slightly bigger than 5MB and has 3 processes. Moreover, each application has its own timing requirements, eg. period, deadline and priority for each of the processes it contains. Both applications have a periodic execution behavior, which is repeated after 16 Minor Frames (MIFs), following a cyclic static schedule.

The experiments have been conducted on a PowerPC-based P4080 platform [14], one of the platforms of choice of the PROXIMA project. In our platform we use SYSGO's PikeOS APEX paravirtualized guest RTOS [15] over the PikeOS microkernel-based hypervisor, which is compliant with the ARINC653 specification [12]. The operating system is configured to have exclusive access to the L3 cache of the platform, which uses as scratchpad, in order to provide high performance execution of the RTOS kernel, while it provides a predictable behavior, since it doesn't have an impact on the user space application cache contents, neither on their execution time.

5.1 Memory Resources

The determination of the new memory requirements for both applications using our iterative method required less than a working day in total, which confirms its fast convergence and its industrial viability. For both applications the upperbound of the code and data pool has been identified to 16M, while the binary search pinpointed the minimum required sizes to 11MB and 15MB for the small and the big applications respectively. This increase is mainly due to the high number of functions contained in each application, which are eagerly randomized. For example, even functions that are only invoked in a specific operation mode (eg. take-off) or perform error handling and therefore they might not be invoked frequently are software randomized. Since the cache way of the P4080 instruction cache is 4KB, for each function 8 extra KB are allocated, in order to allow for a random placement of the function in this range, so that the mapping of the function may be in any possible set.

Regarding the stack size limits of the ARINC653 processes, all 4 processes had a small maximum stack size, in the order of KBs. In 3 out of the 4 processes, doubling the maximum stack size has been sufficient, since the size of the stack was already small partially due to the small stack frame size of each function and also due to the small function call depth. In one of the processes however, the call depth was deeper requiring a larger increase, 16 times larger as an upperbound, and 10 times using the binary search. This apparently big relative increase is exaggerated due to the fact that the size of the P4080's data cache way is also 4KB, and therefore this is the amount of padding introduced by the stack randomization in order to achieve the mapping of the stack frame in any data cache line. This amount is much bigger than the few bytes of each stack frame, but in absolute numbers it is low, since it still allows the stack limit to stay within the range of hundreds of KBs.

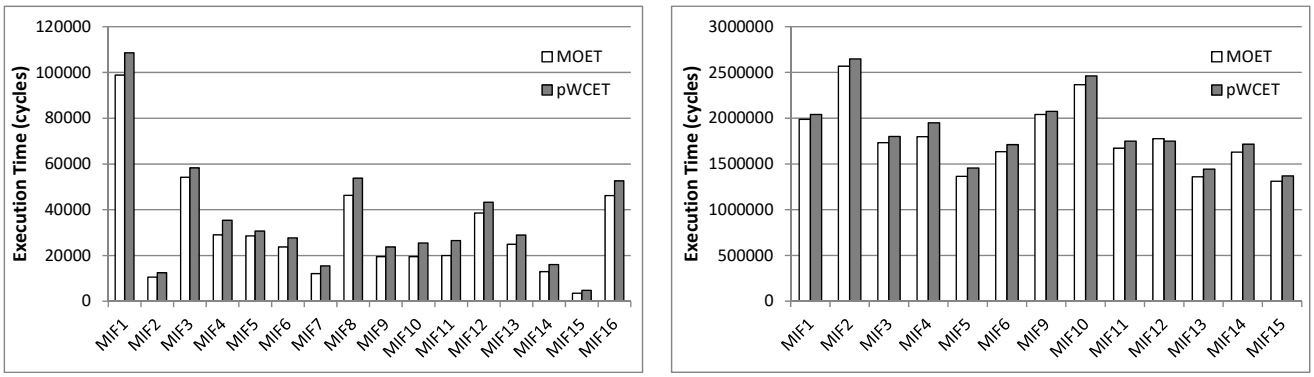
The resulting overall partition sizes are well within the specifications considered for the memory requirements of future avionics applications, as well as the total memory capacity of the future avionics hardware platforms.

5.2 Timing Resources

With respect to the timing properties of the partitions and their processes, none of the tasks periods or deadlines needed to be adjusted. Current industrial practice [2] adds an engineering margin around 20% over high watermark execution times as WCET [16], which incorporate further slack compared to the actual deadline. Note however, that this margin is specific to the target platforms used in current avionics systems, such as the Freescale MPC755 and cannot be safely used in other architectures, especially the ones featuring more high-performance hardware components, which can create a big discrepancy between average and worst-case performance, such as our platform, based on a highly speculative out-of-order execution core, featuring multiple levels of cache.

In order to verify this hypothesis, we have run several experiments of the applications without software randomization. For a pool of 1000 observed execution times of both applications, we have observed cases with unusually high execution times. Such a case is shown in Figure 1, where we show the first 250 executions of a MIF of APP1. We observe that there are 3 cases with a very big execution time, in the order of 2× bigger than the rest of the execution times. This behavior is not consistent with the application behavior on flight computers based on the Freescale MPC755, and due to the lack of full hardware documentation and deep observation facilities it was not possible to identify its exact source. For this reason, the 20% margin is not safe for this architecture and it is only provided below as a reference.

After applying software randomization in both applications, we examined the execution time of the MIFs of interest of each one and derived their pWCET using the commercial pWCET timing analysis software RVS [17]. The timing analysis has been performed only



(a) APP1 (b) APP2

Figure 2: MOET and pWCET values for 10^{-12} probability.

for the most critical process of each application. The pWCET results for cut-off probability 10^{-12} and the Maximum Observed Execution Time (MOET) for both applications are shown in Figure 2.

In APP1, the pWCET estimates for probability 10^{-12} are between 3-8% compared to the high watermark, while in APP2 for probability 10^{-12} between 6-36%, with only six MIFs above 20%. However, as we have already mentioned, the 20% margin is not appropriate for P4080, since the platform exhibits much larger jitter, which might not be able to be observed during analysis time measurements.

Finally, in Figure 3 we show the pWCET curve for a representative MIF of APP1 and APP2. We observe that the measurements of the application, represented with the red line (Measured Execution Time-MET) can only reach a probability up to 10^{-3}, since we have collected 1000 execution time samples. RVS integrates the latest MBPTA implementation, the open-source MBPTA-CV [18], which ensures that enough samples from the tail of the execution time distribution are present in the measurements in order to accurately obtain the parameters of the distribution required to compute the pWCET. If more measurements from the tail of the distribution are required, it instructs the user to collect more samples. In all the cases, 1000 samples have been deemed enough for the application of MBPTA, which have been collected in less than a working day, too. Note that the execution time measurements are then processed by MBPTA-CV in order to compute the pWCET curve, which is shown with the magenta colored line (WCET_EVT). In that case, the distributions can reach very low probabilities and as expected, the pWCET distribution upperbounds the collected execution time measurements.

6 RELATED WORK

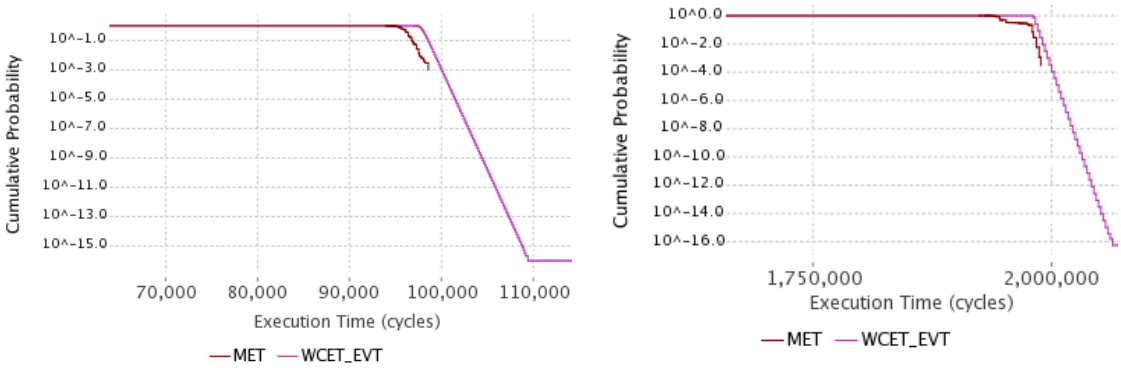
Software randomization has been proposed in the literature for several purposes. Security is probably the domain which has used randomization more than any other one. The first notion of randomization for security can be found in the work of Forrest et al. [19], who propose several methods to diversify applications. Address space layout randomization (ASLR) [20] has been adopted by both Unix-based and Microsoft's operating systems for the desktop and server markets, in order to make hardware attacks exploiting information about the memory layout of programs, like buffer overruns, harder. Recently, a microarchitectural flaw has been discovered called Meltdown[21] in the design of modern processors based on speculative execution, which allows leaking of information even under ASLR. An improved version of ASLR, called Kernel Page Table Isolation (KPTI) also known as KAISER [22] has been proposed, in order to prevent such leakage of information.

Other uses of software randomization have found application in software resilience to bugs. Berger et al. [23] proposed a series of techniques which increase the robustness of applications in the presence of memory access violations. Such techniques have also found their way to commercial operating systems, such as Microsoft Windows 8 and later.

In the real-time and critical embedded systems domain, software randomization is the enabler of MBPTA [1] properties on conventional hardware. In particular, software randomization allows to comply with the requirement of execution time of end-to-end measurements to exhibit a behavior that can be described by an independent and identically distributed random variable. At the same time, the random execution time behavior of the system provides the same behavior both at analysis and during the execution time, satisfying the second requirement of MBPTA, too.

Several previous works in the literature have applied software randomization in the context of MBPTA on industrial case studies. Wartel et al. [2] applied dynamic software randomization on two avionics applications and provided a comparison with hardware software randomization. However, their evaluation was based on a simulated hardware platform and used an academic open-source RTOS, while our work is executed on a real industrial setup, with actual COTS hardware and a commercial ARINC653 RTOS. For this reason, we had to deal with the resource management issues which have not been detected in earlier preliminary evaluations.

Cros et al. [4] applied software randomization on an aerospace case study. Their work is similar to ours, since they also use dynamic software randomization on COTS hardware (LEON3) with a commercial RTOS (SYSGO RTEMS SMP). However, their work is only focused on the timing analysis of the application with MBPTA. In particular they present the modifications required to the dynamic software randomization to be ported to their target platform and to support a bounded amount of memory and execution time, which is



(a) APP1 (b) APP2

Figure 3: pWCET plots and measurements for representative MIFs of each application.

required for real-time systems. In contrast, our work solely focuses on the challenges arising from the use of software randomization in more restrictive environments such as avionics, which require the determination and allocation of explicit memory and timing budgets for each application and need to be adjusted accordingly when they are software randomized.

Kosmidis et al. [10] used static software randomization on an academic model-based generated application, resembling an automotive control application derived from an industrial application specification. Their setup is based on the AURIX Tricore processor and uses the ERIKA open source RTOS. Similar to [4] this work is also only focused on the ability to compute a WCET on a deterministic platform using software randomization, and does not take into account the impact of software randomization in the resource allocation of software units.

Finally, several other works used commercial applications on top of MBPTA compatible hardware. Wartel et al. [16] performed the first application of MBPTA on avionics software, however they used a hardware simulator and no RTOS. Fernandez et al. [3] computed the pWCET of a thruster application from the aerospace domain, while Hernandez et al [8] evaluated the implementation of a hardware randomized processor with an aerospace case study, too. Agirre et al. [9] define a safety concept for the application of MBPTA on either a hardware or software randomized platform for a railway application, while the same application is used by Mezzetti et al. [24] for the evaluation of a path coverage technique for MBPTA and randomized caches. However, hardware randomization does not present any challenge from the application or RTOS point of view, since it is completely transparent to both.

7 CONCLUSION

In this paper we explained the implications of using Dynamic Software Randomization in an avionics environment and we presented the solutions we developed for this task. In particular, software randomization requires the adjustment of memory and timing budgets allocated for ARINC653 applications. To that end, we presented a method to compute the upperbounds of the various memory requirements of avionics applications and showed that it can be used in practice even for applications with large memory footprint. To demonstrate its effectiveness, we successfully applied DSR in two commercial avionics applications on an industrial COTS hardware and RTOS setup, obtaining their pWCET estimations with MBPTA, which are competitive with respect to current industrial practice based on measurements.

ACKNOWLEDGMENTS

The work leading to these results has been funded by the European Community's Seventh Framework Programme (FP7/2007-2013) under the PROXIMA Project (grant agreement 611085) and by the European Research Council with Horizon 2020 (grant agreement No. 772773). Moreover, it has been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2015-65316-P and the HiPEAC Network of Excellence.

REFERENCES

[1] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. J. Cazorla, "Measurement-Based Probabilistic Timing Analysis for Multi-path Programs," in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2012, pp. 91–101.

[2] 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. Cazorla, "Timing Analysis of an Avionics Case Study on Complex Hardware/Software Platforms," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2015, pp. 397–402.

[3] M. Fernandez, D. Morales, L. Kosmidis, A. Bardizbanyan, I. Broster, C. Hernandez, E. Quiñones, J. Abella, F. Cazorla, P. Machado, and L. Fossati, "Probabilistic Timing Analysis on Time-Randomized Platforms for the Space Domain," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, March 2017, pp. 738–739.

[4] F. Cros, L. Kosmidis, F. Wartel, D. Morales, J. Abella, I. Broster, and F. J. Cazorla, "Dynamic Software Randomisation: Lessons Learned From an Aerospace Case Study," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, March 2017, pp. 103–108.

[5] F. J. Cazorla, J. Abella, J. Andersson, T. Vardanega, F. Vatrinet, I. Bate, I. Broster, M. Azkarate-Askasua, F. Wartel, L. Cucu, F. Cros, G. Farrall, A. Gogonel, A. Gianarro, B. Triquet, C. Hernandez, C. Lo, C. Maxim, D. Morales, E. Quiñones, E. Mezzetti, L. Kosmidis, I. Aguirre, M. Fernandez, M. Slijepcevic, P. Conmy, and W. Talaboulma, "PROXIMA: Improving Measurement-Based Timing Analysis through Randomisation and Probabilistic Analysis," in *Euromicro Conference on Digital System Design (DSD)*, 2016.

[6] L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla, "A Cache Design for Probabilistically Analysable Real-time Systems," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2013, pp. 513–518.

[7] L. Kosmidis, C. Curtsinger, E. Quiñones, J. Abella, E. Berger, and F. J. Cazorla, "Probabilistic Timing Analysis on Conventional Cache Designs," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2013, pp. 603–606.

[8] C. Hernández, J. Abella, F. Cazorla, A. Bardizbanyan, J. Andersson, F. Cros, and F. Wartel, "Design and Implementation of a Time Predictable Processor: Evaluation With a Space Case Study," in *29th Euromicro Conference on Real-Time Systems, ECRTS 2017*, 2017, pp. 16:1–16:23.

[9] I. Agirre, M. Azkarate-Askasua, A. Larrucea, J. Perez, T. Vardanega, and F. J. Cazorla, "A Safety Concept for a Railway Mixed-Criticality Embedded System Based on Multicore Partitioning," in *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, Oct 2015, pp. 1780–1787.

[10] L. Kosmidis, D. Compagnin, D. Morales, E. Mezzetti, E. Quiñones, J. Abella, T. Vardanega, and F. Cazorla, "Measurement-Based Timing Analysis of the AURIX Caches," in *International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2016, pp. 9:1–9:11.

[11] L. Kosmidis, R. Vargas, D. Morales, E. Quiñones, J. Abella, and F. Cazorla, "TASA: Toolchain Agnostic Software Randomisation for Critical Real-Time Systems," in *International Conference On Computer Aided Design (ICCAD)*, 2016, pp. 59:1–59:8.

[12] ARINC, "Avionics Application Software Standard Interface: ARINC Specification 653P1-3. Aeronautical Radio," 2010.

[13] J. L. Bentley and A. C. Yao, "An Almost Optimal Algorithm for Unbounded Searching," *Information Processing Letters*, vol. 5, pp. 82–87, 1976.

[14] FreeScale, *P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual. Rev 1*, 2012.

[15] SYSGO, "Pikeos," https://www.sysgo.com/products/pikeos-hypervisor/partitions-guest-os, Accessed August 2018, 2003.

[16] 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 *IEEE Symposium on Industrial Embedded Systems (SIES)*, June 2013, pp. 241–248.

[17] Rapita Systems Ltd., "Rapitime, part of the rapita verification suite," 2004, http://www.rapitasystems.com/products/rvs, Accessed August 2018.

[18] J. Abella, M. Padilla, J. Castillo, and F. Cazorla, "Measurement-Based Worst-Case Execution Time Estimation Using the Coefficient of Variation," *ACM Transactions Design Automation of Electronic Systems*, vol. 22, no. 4, pp. 72:1–72:29, Jun. 2017.

[19] S. Forrest, A. Somayaji, and D. H. Ackley, "Building Diverse Computer Systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems*, May 1997, pp. 67–72.

[20] PaX Team, "PAX," 2001, https://pax.grsecurity.net, Accessed August 2018.

[21] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, Jan. 2018.

[22] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," in *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings*. Cham: Springer International Publishing, 2017, pp. 161–176.

[23] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic Memory Safety for Unsafe Languages," in *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*. ACM Press, 2006, pp. 158–168.

[24] E. Mezzetti, M. Fernandez, A. Bardizbanyan, I. Agirre, J. Abella, T. Vardanega, and F. Cazorla, "EPC Enacted: Integration in an Industrial Toolbox and Use against a Railway Application," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2017, pp. 163–174.