THE CHALLENGE OF PROFILING MULTI-CORE SAFETY-CRITICAL EMBEDDED SYSTEMS

Sylvain Girbal
Thales Research & Technology
Palaiseau, France
sylvain.girbal@thalesgroup.com

Jimmy Le Rhun
Thales Research & Technology
Palaiseau, France
jimmy.lerhun@thalesgroup.com

***Abstract*—For many decades, the High Performance Computing (HPC) industry has relied on profiling techniques for debug and performance optimization purposes, either for single-core or multi-core systems.**

On the other hand, the safety-critical industry, with the shift from single-core to multi-core COTS processors for time-critical products such as avionics, railway or space computer subsystems is facing new challenges with a trade-off between performance and predictability.

In multi-core processors, concurrent accesses to shared hardware resources are generating inter-task or inter-application timing interference, breaking the timing isolation principles required by the qualification / certification standards associated with for such critical software. Several solutions have been proposed in the literature to control or regulate timing interference, but most of these solutions require to perform some level of runtime profiling or monitoring.

However, regular profiling techniques relying on interrupts, multi-threading, or OS modules are usually not an option with Real Time Operating Systems (RTOS).

Additionally, to accurately quantify the maximum inter-application slowdown due to timing interference, we require specific co-running applications, named stressing benchmarks and designed to produce a preset workload on each shared hardware resource, and generate the ad-hoc traffic.

In this talk, we present the profiling techniques we are using to evaluate time predictability in the presence of timing interference and ensure real-time behaviour of safety-critical applications: 1) METrICS, a measurement environment for multi-core time-critical systems running on top of the industry-standard PikeOS RTOS. 2) A set of Stressing Benchmarks, dedicated at generating workloads on specific hardware resources. 3) xTRACT Visualizer, a GUI allowing us to visualize and to analyze the collected information and measure timing-interference level.

I. INTRODUCTION

The safety-critical industry is facing a demand for cheaper equipment and more stringent SWaP (Size Weight and Power) constraints [2], making the shift to multi-core COTS processor products appealing. A consequence is a larger trade-off in terms of performance versus predictability [17], [21].

On a multi-core processor, different pieces of software will be executed on different cores at the same time. Such different software will compete electronically to use the shared hardware resources of the processor architecture, causing concurrent accesses to the same hardware.

On the hardware resources side, concurrent accesses are arbitrated, introducing inter-task or inter-application jitter defined as **timing interference** [10]. These interference are breaking the timing isolation principles required by the standards [13], [14], [24] of time-critical software.

The literature [9] proposes several Deterministic Platform Solutions to tackle this problem, including **control solutions** [5], [10], [8], [16], [15] aiming at completely preventing such timing interference and **regulation solutions** [26], [29], [18] reducing the amount of interference below a harmful level.

However, most of these solutions (especially regulation solutions) requires some level of **profiling**, to either accurately measure task runtimes, or hardware resource load level.

II. PROFILING EMBEDDED SYSTEMS

Performance monitoring and profiling tools have been existing for a long time to help the HPC programmers with debugging their systems, optimizing their applications, or identifying bottlenecks. A wide variety of generic tools exists for non-RTOS systems [28] such as gprof [7], valgrind [22], or atom [6]. These tools rely on either OS features such as multi-threading, interrupts or timers, or either on pseudo-automatic code instrumentation to collect the required timing information.

However, in real-time operating systems, such features are either not available (with enforced static scheduling), restricted or prohibited due to their impacts on time determinism (such as the impact of interrupts on WCET). This is especially true for safety critical software that is constrained by drastic limitations due to the safety standards [13], [14], [24].

Beyond this limitation, if collecting timing information is enough to observe timing interference, it is not sufficient to regulate the shared resource usage that causes interference due to resource contention. As a consequence collecting additional resource usage information is as critical as collecting timing information.

Generic tools such as oprofile [19] specialize in collecting such information by gathering the Performance Monitor Counters that are usually only available in privileged mode. The claim is that oprofile is low-overhead and non-obtrusive, and it is true from a non-RTOS point of view: Both the monitored application and the kernel remain untouched thanks to a dedicated kernel module. Also, the overhead mainly depends on the interrupt-based sampling frequency.

In RTOS systems, features like modular kernels do not exist, and using interrupt-based sampling is not an option for

systems based on static scheduling. Such systems are relying on micro-kernels and modularity is even prohibited for safety and security reasons. Besides, "low-overhead" does not have the same meaning for large scale systems running minutes to hour-long applications where a cost of tens of milliseconds is negligible and for periodic safety critical systems that are likely to have tasks deadlines in the order of 10 millisecond or less, and to have RTOS services that should be completed in microseconds.

Furthermore, dealing with timing interference forces us to perform measurement at function-call or system-call level, where even a cost of tens of microseconds might not be acceptable.

In addition, resource contentions (the main sources for timing interference) only occur at specific moments in time, during the cycles when an arbitration occurs. As a consequence, measurement and overheads have to be evaluated at cycle level.

Finally, if sampling techniques are very efficient for best effort applications, such techniques can be very troublesome for safety critical applications that focuses on how the worst case should behave. The sampling just acts as a filter that could filter out the worst case.

To summarize, the challenge is to provide a way to 1) perform an accurate real-time runtime and resource usage measurement, 2) with a negligible impact on timing behaviour, 3) running outside of the operating system (avoiding system calls) to be able to profile both the RTOS and the running applications.

III. METrICS: A MEASUREMENT ENVIRONMENT FOR TIME-CRITICAL EMBEDDED SYSTEMS

In [11] we introduced **METrICS**: a Measurement Environment for Multi-Core Time Critical Systems that is running on top of the PikeOS [1] RTOS from SYSGO. This framework proposes accurate runtime and resource usage measurement while having a negligible impact on timing behaviour.

A. METrICS architecture

METrICS consists of several core components appearing in green in Figure 1. On the left side, we present the components actually running on the target hardware board, and on the right side the METrICS server, running on a Linux host, and in charge of driving the experimental campaign to be run on the board and collect all the gathered profiling information.

The **METrICS library** is meant to be linked with the target applications to provide them with an access to the measurement probes API, allowing the collection of time and resource access information.

The **Syscall instrumentation layer** provides a way to automatically instrument each APEX system calls for ARINC-653 avionic applications.

The **Hardware Monitor Kernel Driver** provides the supervisor-level privilege necessary to access to hardware performance monitor counters (PMC). Such counters introduced in [27] allow us to count some hardware events, including the accesses to some shared hardware resource.

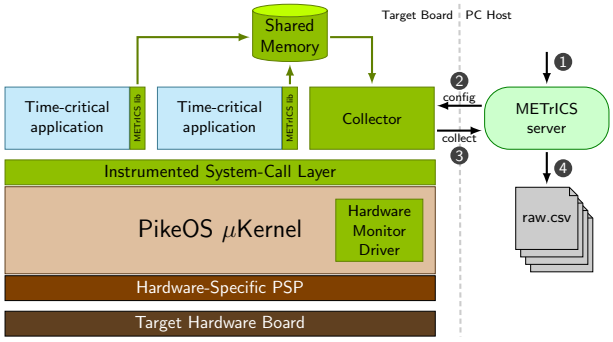

Fig. 1. Architecture of the METrICS measurement tool

The **collector partition** is in charge of 1) defining a shared memory space to collect measurements; 2) configuring specific measurement scenarios; 3) transferring the collected profiling information to the Linux host.

Finally, the **METrICS server** running on the Linux host. It drives the experimental campaign and gather the collected profiling information.

B. METrICS intrusiveness

A major challenge in profiling tools is its intrusiveness in the system it monitors. We distinguish **execution time intrusiveness** and **code intrusiveness**. The former limits the accuracy of the measurement due to the monitoring overhead, whereas the latter requires an effort from the developer to instrument the code of the application, which could be an issue for legacy software.

As our context is safety-critical and time-critical embedded applications, we focused METrICS into limiting **execution time intrusiveness**, to have a minimal impact on the timing interference phenomenon.

A time intrusiveness had been performed of a full METrICS probe consisting of: 1) retrieving the timing information thanks to the core-dedicated special registers; 2) retrieving the performance monitor counters, again through direct register access; 3) retrieving thread-specific information from the OS; and 4) storing the collected information into the shared memory. The results are presented in Figure 2.

Over 180K runs, the probe time varies from 85ns up to 392ns. For 97% of the runs the overhead is below 110ns, and the overhead is above 191ns for only 0.002% of the cases.

In comparison, the corresponding RTOS system call to only obtain current time (p4_get_time for PikeOS) requires 240ns, and it only get the current time and no PMC information. This is due to the fact that a system call involves at least two context switches, and possibly some privilege level changes.

Therefore the low intrusiveness of the overall METrICS probe makes it viable even for characterizing few micro-second long system calls of the RTOS.

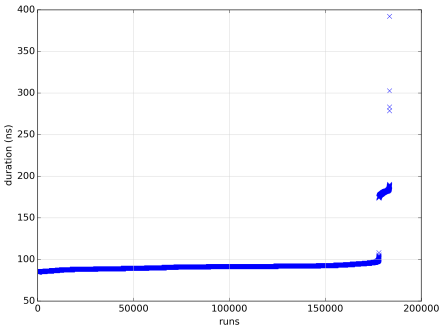

Fig. 2. Completion time of a METrICS probe over 180000 runs

C. Profiling Design Space

Target ARM or PowerPC embedded hardware usually provide a selection of about 250 hardware events that could be measured with performance monitor registers. Among these events around 50 of them are actually related to shared hardware resource to be profiled.

However, these architecture only provides a limited number of performance monitor registers (from 4 to 6), only allowing to concurrently profile a few hardware resource. As a consequence, a large number of runs are necessary to systematically study correlations between performance monitor counters and observed runtime (around C_{50}^{6} runs).

To perform such a large number of experiments some form of automation is necessary, and driven by the METrICS server on the Linux host. The different steps of the automated profiling process are appearing in Figure 1, with 1) the selection of target executable and test configuration, 2) the configuration of hardware counters to use, 3) the collection of measurements and, 4) the storage of result files.

Such experimental campaigns generate a rather large amount of raw data, making the direct analysis quite difficult. In the Section V, we present the visualization tools we developed to assist the analysis.

IV. Stressing Benchmarks for Exercising Interference Channels

A non-intrusive ability to measure runtime and resource accesses is not enough to quantify **timing interference** and identify **interference channels** as now required by the certification authorities for multi-core safety critical systems.

To exercise the sensitivity of each applications to additional accesses to shared hardware resource from other cores, it is required to design specific co-running benchmarks specialized in performing a preset number of resource accesses.

Such benchmark have been introduced [25] as **stressing benchmarks**. They were designed as a way 1) to characterize both the partially documented hardware architecture and blackbox applications; 2) to actually identify the shared hardware resources and associated contention mechanisms; and 3) to identify the hardware resources each application is sensitive to.

In [3], stressing benchmark are extended to be able to perform a tunable access workload to simultaneous hardware resource to compute application signatures in terms of hardware resource usage. These signatures are then used as an alternative way to bound WCET of co-running applications.

A. Designing stressing benchmarks

A stressing benchmark is designed to perform periodic accesses on a memory mapped region, performing either read or write accesses.

Depending on the **target memory address and size**, we can stress out different peripheral, from the DDR memory to PCIe peripherals.

The **stride** (or address distance) between two accesses defines the spatial access pattern that impacts which part of the memory hierarchy will be targeted (L1 cache, L2 cache, or DDR memory / interconnect).

Finally, the **stress-level** of the benchmarks indicates how often the memory access are performed, defining the accesses temporal pattern : from continuous burst accesses to repeated elementary accesses, and various duty cycles in between.

To effectively control the ratio of memory instructions over other kind of instructions stressing benchmarks are written directly in assembly code, progressively inserting NOP instructions to lower the memory load level.

B. Using stressing benchmarks

The main purpose of the stressing benchmark is to be used as a co-runner benchmark, to be able to compare the applications runtime in isolation and with the co-runner stressing benchmark. This will allow us, for each application, to evaluate its sensitivity to timing interference.

In the context of time-critical software and timing interference, it is critical to bound the most adverse effect a co-running application can have on a specific application time behavior.

Monitoring with METrICS both the execution time of an application and the effective number of accesses to hardware resource, while progressively increasing the stress level allow us to determine: 1) the maximum available bandwidth in terms of access to this resource; and 2) the level of extra resource access supported by our applications before being significantly slowed down by the timing interference phenomenon.

This is depicted in Figure 3. The y-axis represents the observed runtime of the monitored application, while the x-axis represents the stress level (aka the number of extra accesses performed by the stressing benchmark).

The leftmost point in the chart corresponds to the application running alone in isolation (aka the classical WCET of the application running in isolation). The rightmost point in the chart corresponds to a permanent maximum workload from the stressing benchmark actually preventing the monitored application to access the required resource (aka some denial of service from a co-runner).

The Pareto optimal point of the curve, appearing in red in Figure 3, corresponds to the level of stress beyond which the induced slowdown overhead is no more acceptable. The

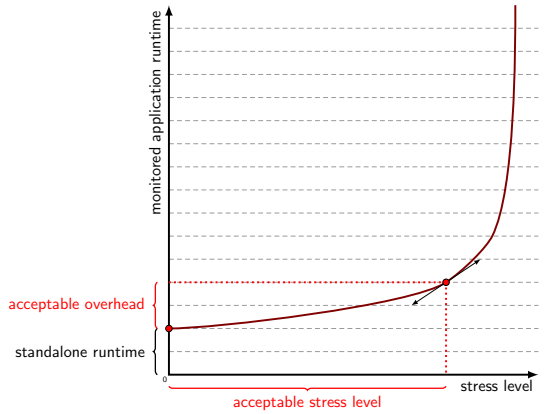

Fig. 3. Determining an acceptable level of slowdown and the associated extra access budget

asymptotic shape of such curves show that not dealing with timing interference and continuing to rely on resource over-provisionning to compute a WCET bound is not an option for multi-core processors [23], [9].

C. Overall profiling process

By repeating the above-mentioned procedure for each shared hardware resource, monitoring with METrICS and exercising the different resource with adhoc stressing benchmarks, we are able to actually identify the interference channels, and to quantify the impact of timing interference. The overall two-step process is depicted in Figure 4.

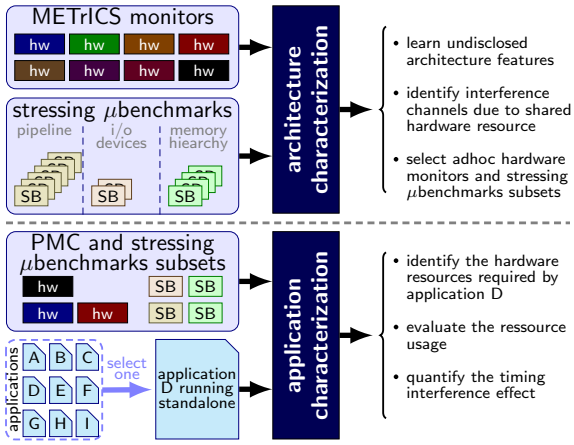

Fig. 4. Overall profiling process

The purpose of the first step is to performed an hardware-only characterization, running the stressing benchmarks concurrently to identify some undisclosed architecture features such as the interference channels, and quantify the actual available bandwidth to each hardware resource.

The second step focuses on software characterization, running each target application concurrently with stressing benchmarks to quantify the application sensitivity to hardware resource usage and measure actual timing interference.

V. xTRACT VISUALIZER FOR PROFLING ANALYSIS

Considering the large amount of profiling information collected during our profiling processes, it is also necessary to somehow automate data mining and profiling data analysis.

To this extent, we developed a GUI providing different ways of visualizing the collected data: **xTRACT visualizer** (expert Timing and Resource Access Counting Trace visualizer). Its technology is based on pandas [20] for scalable data mining, matplotlib [12] and d3.js [4] for graphical rendering, and a Qt-based GUI using the pyside Python binding to bundle all the visualization / filtering options.

A. Visualizing runtime variability

As METrICS collects the full distribution of events (rather than only minimum / maximum values), it allows us to build histograms to visualize runtime variability, showing the distribution of the observed runtimes during successive runs, as depicted in Figure 5.

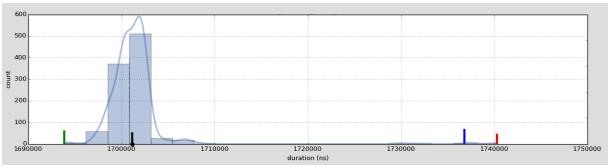

Fig. 5. Histogram of the drone partition runtime as appearing in xTRACT visualizer

The x-axis corresponds to the observed duration while the y-axis indicates how many times each runtime has been observed. The best (shortest) runtime appears on the left, the worst (longest) observed execution time on the right, the median value being identified with a black dot.

B. Visualizing shared hardware resource usage

We also build histograms with the collected Performance Monitor Counter data, as shown in Figures 6 and 7. Comparing these histograms with the previous one allows us to intuit possible correlations between resource usage and runtime.

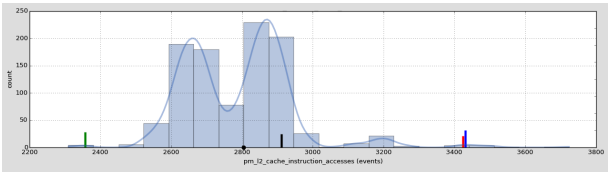

Fig. 6. Histogram of correlating resource accesses (L2 read cache accesses) as appearing in xTRACT visualizer

C. Correlating timing interference

To confirm the intuited timing interference slowdowns with hardware resource accesses, we build scatterplots such as the one appearing in Figure 8.

Scatterplots are a way to easily identify correlations. Each point of the scatterplot indicates that a particular run has been observed with a number of resource accesses equal to the value

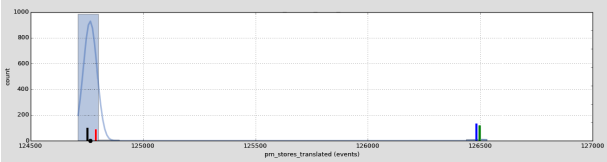

Fig. 7. Histogram of not correlating resource accesses (issued store instructions) as appearing in xTRACT visualizer

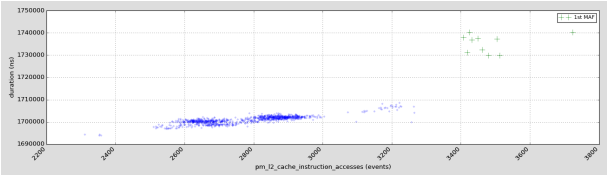

Fig. 8. Scatterplot showing linear correlation between runtime and L2 read cache accesses as appearing in xTRACT visualizer

on the x-axis, and an observed runtime equal to the value on the y-axis. If the points approximate a straight line, there is a linear correlation.

D. Visualizing system call preemption

We also render various charts related to the probes automatically inserted around system calls. It allows us to split the effective runtime into the classical user time (time really spent in the application) and the system time (time spent in the operating system to deal with the application I/O). Alternatively it can be used to observe the usage of kernel locks in system calls.

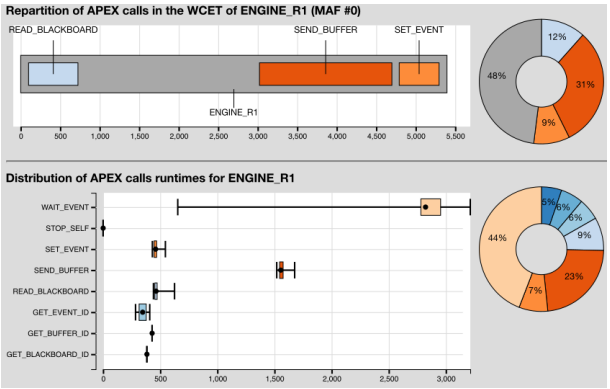

Fig. 9. Visualizing ARINC-653 syscalls in ENGINE_R1 task with xTRACT visualizer

For instance, the top charts of Figure 9 shows the repartition of APEX system calls in an instance of a particular task. The bottom part of figure shows with boxplots the variability of the execution time of APEX system calls for different iterations of the same task, actually showing that the RTOS is also affected by timing interference.

VI. CONCLUSION

In this paper, we have shown the specific challenges faced by the safety-critical computing industry in characterizing the fluctuating performance of multicore COTS system, and presented an ensemble of tools we developed for that purpose. In addition to precise execution time measurement, usage of shared resources has to be monitored, and specific stressing applications are required for calibration and stimulation. The large size of measurement results implies the usage of automation, visualization and statistical analysis tools.

This instrumentation suite allows the development and the experimental evaluation of novel techniques for mitigation of timing interference, as well as other fine-grain optimizations. Future improvements include the integration of state-of-the-art data analytics techniques such as machine learning ,either for interference channel identification or application deployment optimization.

REFERENCES

[1] SYSGO AG. PikeOS 4.2: RTOS with hypervisor-functionality, March 2017.

[2] Thomas G. Baker. Lessons learned integrating COTS into systems. In *Proceedings of the First International Conference on COTS-Based Software Systems*, ICCBSS '02, pages 21–30, 2002.

[3] Jingyi Bin, Sylvain Girbal, Daniel Gracia Pérez, and Alain Merigot. Using monitors to predict co-running safety-critical hard real-time benchmark behavior. In *International Conference on Information and Communication Technology for Embedded Systems (ICICTES)*, 2014.

[4] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, December 2011.

[5] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software and Systems*, ERTS '14, 2014.

[6] Alan Eustace and Amitabh Srivastava. Atom: A flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 25–25, Berkeley, CA, USA, 1995. USENIX Association.

[7] Jay Fenlason and Richard Stallman. GNU gprof. November 1998.

[8] Stuart Fisher. Certifying Applications in a Multi-Core Environment: The World's First Multi-Core Certification to SIL 4, 2013.

[9] Sylvain Girbal, Xavier Jean, Jimmy Le Rhun, Daniel Gracia Pérez, and Marc Gatti. Deterministic Platform Software for hard real-time systems using multi-core COTS. In *Proceedings of the 34th Digital Avionics Systems Conference*, DASC'2015, 2015.

[10] Sylvain Girbal, Daniel Gracia Pérez, Jimmy Le Rhun, Madeleine Faugère, Claire Pagetti, and Guy Durrieu. A complete toolchain for an interference-free deployment of avionic applications on multi-core systems. In *Proceedings of the 34th Digital Avionics Systems Conference*, DASC'2015, 2015.

[11] Sylvain Girbal, Jimmy Le Rhun, and Hadi Saoud. METrICS: a measurement environment for multi-core time critical systems. In *Embedded Real Time Software and Systems*, ERTS '18, 2018.

[12] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[13] International Electrotechnical Commission. IEC 61508: Functional safety of electrical, electronic, or programmable electronic safety-related systems, 2011.

[14] International Organization for Standardization (ISO). ISO 26262: Road Vehicles – Functional Safety, 2011.

[15] Xavier Jean. *Hypervisor control of COTS multi-cores processors in order to enforce determinism for future avionics equipment*. Phd thesis, Telecom ParisTech, June 2015.

[16] Xavier Jean, David Faura, Marc Gatti, Laurent Pautet, and Thomas Robert. Ensuring robust partitioning in multicore platforms for ima systems. In *31st Digital Avionics Systems Conference (DASC)*, pages 7A4–1. IEEE, 2012.

[17] Raimund Kirner and Peter Puschner. Obstacles in worst-case execution time analysis. In *Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 333–339, 2008.

[18] Angeliki Kritikakou, Claire Pagetti, Christine Rochange, Matthieu Roy, Madeleine Faugère, Sylvain Girbal, and Daniel Gracia Pérez. Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems. In *Proceedings of the 22th International Conference on Real-Time and Network Systems (RTNS)*, pages 139–148, 2014.

[19] John Levon. *OProfile - A System Profiler for Linux*. Victoria University of Manchester, 2004.

[20] Wes McKinney. pandas: a foundational python library for data analysis and statistics. November 2011.

[21] E. Mezzetti and T. Vardanega. On the industrial fitness of wcet analysis. In *Proceedings of the 11th International Workshop on Worst Case Execution Time Analysis (WCET2011)*. 2011.

[22] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the Programming Language Design and Implementation Conference*, 2007.

[23] Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. *European Dependable Computing Conference*, pages 42–52, 2012.

[24] Radio Technical Commission for Aeronautics (RTCA) and EURopean Organisation for Civil Aviation Equipment (EUROCAE). DO-297: Software, electronic, integrated modular avionics (IMA) development guidance and certification considerations.

[25] Petar Radojkovic, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *TACO*, 8(4):34, 2012.

[26] Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russel Kegley, Dennis Perlman, Greg Arundale, et al. Single Core Equivalent Virtual Machines for Hard Real-Time Computing on Multicore Processors. Technical report, University of Illinois at Urbana-Champaign, Nov 2014.

[27] Brinkley Sprunt. The basics of performance-monitoring hardware. *Micro, IEEE*, 22(4):64–71, 2002.

[28] Ben Wun. Survey of software monitoring and profiling tools.

[29] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 55–64. IEEE, 2013.