METRICS: A MEASUREMENT ENVIRONMENT FOR MULTI-CORE TIME CRITICAL SYSTEMS

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

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

Hadi Saoud
Thales Research & Technology
Palaiseau, France
hadi.saoud@external.thalesgroup.com

***Abstract*—With the upcoming shift from single-core to multi-core COTS processors for safety critical products such as avionics, railway or space computer subsystems, the safety critical industry is facing a trade-off in terms of performance versus 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 standards for such critical software. Several solutions have been proposed in the literature to control or regulate these timing interference, but most of these solutions require to perform some level of profiling, monitoring or dimensioning.

As time-critical software is running on top of Real Time Operating Systems (ROTS), classical profiling techniques relying on interrupts, multi-threading, or OS modules are either not available or prohibited for predictability, safety or security reasons.

In this paper we present METrICS, a measurement environment for multi-core time-critical systems running on top of the industry-standard PikeOS RTOS. Our framework proposes an accurate real-time runtime and resource usage measurement while having a negligible impact on timing behaviour, allowing us to fully observe and characterize timing interference.

Beyond being able to characterize timing interference, we evaluated METrICS in term of accuracy of the timing and resource usage measurements, intrusiveness both in term of timing and impact on the legacy code. We also present a portfolio of the kind of measurements METrICS provides.

I. INTRODUCTION

For the last decades, industries from the safety-critical domain have been using Commercial Off-The-Shelf (COTS) processors despite their inherent runtime variability. To guarantee hard real-time constraints in such systems, designers massively relied on resource over-provisioning, time and memory space partitioning, and disabling the features responsible for runtime variability.

The demand for cheaper equipment and more stringent SWaP (Size Weight and Power) constraints [5] makes the shift from from single-core to multi-core COTS processor for safety critical products appealing. But, as a consequence, the industry is facing an even larger trade-off in term of performance versus predictability [23], [27].

On a multi-core processor, different pieces of software will be executed on different cores at the same time. Such software will, even if they are completely independent, 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** [15]. These interference are breaking the timing isolation principles required by the standards [19], [20], [30] of time-critical software.

The literature [14] proposes several Deterministic Platform Solutions to tackle this problem, including **control solutions** [9], [15], [12], [22], [21] aiming at completely preventing such timing interference and **regulation solutions** [31], [37], [24] reducing the amount of interference below a harmful level.

However, most of these solutions (especially regulation solutions) require to accurately measure either task runtime or some particular hardware resource loads using performance hardware counters [32].

On one hand, some of these solutions are only **prototyped in bare-metal**, neglecting the timing interference that would have been caused by the operating system itself. However, in a safety critical context, the applications are usually driven by data coming from sensor devices issuing some I/O accesses. To be able to deal with concurrent I/O accesses, the RTOS is introducing some software locks. As a consequence, these locks take a significant part of the timing interference and neglecting them might not be relevant in such a context. Running without an operating system also means the absence of a global scheduler that would allow to easily run concurrent applications. Therefore, these bare-metal models are restricted to single applications able to exploit intra-partition parallelism with a single task running in parallel on several cores. A different inter-partition parallelism scheme with different tasks running concurrently on different cores would imply embedding some scheduling facility inside the application itself.

On the other hand some solutions are **relying on the RTOS** to perform these controls/regulations. Doing so introduces a bias in time measurement caused by the associated system call that might itself introduce a lock (especially when several tasks try to use the same service concurrently). It also prevents us from performing the timing measurement on the operating system itself: how could one rely on a system call to measure the time taken by such a call, or the absence of locks within it?

In the classical Linux world, several techniques or solutions exists to measure timing and perform resource profiling such as

gprof [11], valgrind [28], atom [10] or oprofile [25]. However, as detailed in the next section, these solutions rely on features that are not available or prohibited for hard real-time systems.

As a consequence, 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 behavior, 3) running outside of the operating system (avoiding system calls) to be able to profile the RTOS as well as the running applications.

Ideally, the toolset should be able to provide some of the information previously available only with a JTAG probe [13], but without requiring the hardware devices, nor the associated skills, making the information available to every software programmer.

In this paper we introduce **METrICS**: a Measurement Environment for Multi-Core Time Critical Systems running on top of the PikeOS [1] RTOS from SYSGO. This tool is intended to help safety-critical software and system developers to evaluate their design choices in terms of performance and predictability. We evaluate the impact of METrICS in term of accuracy and intrusiveness and provide some examples of usage to extract timing-interference related information.

The paper is organized as follows: In Section II we detail the challenges of runtime and activity measurements for time critical software. In Section III we present the software architecture of METrICS, our proposed measurement environment and in Section IV we furthermore detail how it operates.

Evaluation starts with Section V, analyzing the accuracy and intrusiveness of METrICS both in terms of source code and timing. In Section VI, we provide an example of METrICS usage to evaluate different deployments of a multi-core application, measuring communication overhead due to timing interference. In Section VII we explain how we deal with the large number of experiments required to explore all possible configuration of previous use-case, justifying the need for automatic instrumentation and statistical processing. Finally, in Section VIII, we detail the associated GUI we developed as a support for analysis and interpretation of the results obtained with METrICS.

II. Challenges to Real-Time Measurement

Performance monitoring and profiling tools have been existing for a long time to help the programmers with debugging their systems, optimizing their applications, or identifying bottlenecks. A wide variety of generic tools exists for non-RTOS systems [36] such as gprof [11], valgrind [28], or atom [10]. 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.

In a real-time operating system, 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 [19], [20], [30].

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 resource usage information is as critical as collecting timing information.

Generic tools such as oprofile [25] 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. Also ”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.

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.

Also, 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.

III. METrICS Software Architecture

METrICS consists of several components appearing in Figure 1. The brown parts in the figure correspond to, from bottom to the top, the selected embedded computer architecture and the PikeOS operating system above the platform support package (PSP) corresponding to the board. The blue parts correspond to the running applications we wish to monitor, each in their own partition.

Finally the green parts in Figure 1 are the core components of the METrICS environment and are described below:

A. The METrICS Library

The **METrICS library** is meant to be linked with the running applications to provide them with an access to the measurement probes API, allowing the collection of time and resource access information. The library contains: 1) the instrumented system call layer; 2) the application instrumentation interface; 3) the user-level interface to the instrumentation kernel driver; and 4) the user-level interface to the collector.

Syscall instrumentation layer: the instrumentation of system calls of some PikeOS personalities (e.g. the APEX

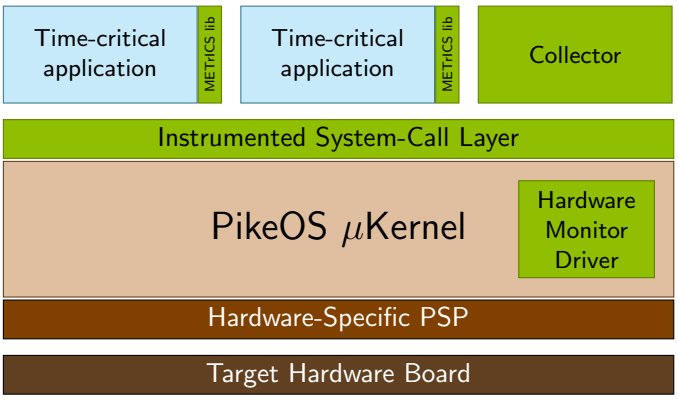

Fig. 1. Architecture of the METrICS measurement tool

system calls in case of ARINC-653 applications) automatically inserts measurement probes before and after every system call. It especially allows us to determine communication times for the intra-partition and inter-partition communications that relies on such system calls, and that can be a significant part of the application running time.

Application instrumentation: besides the syscall-level instrumentation, we provide the ability to manually insert measurement probes directly in applications. This is achieved by adding a pair of functions (`metrics_probe_begin()` and `metrics_probe_end()`) around the section of the code to be monitored. Upon execution, these functions collect the highest precision time-base counter available (usually corresponding to the number of elapsed CPU cycles since booting) and the PMC registers of the current core. The later function is also responsible for sending the monitored data to the collector.

B. The Hardware Monitor Kernel Driver

On most hardware architectures, the access to hardware performance monitor counters (PMC) requires supervisor-level privileges. Also on all supported hardware targets, the configuration of these registers to select the events that should be counted does require these privileges. As PikeOS legitimately prevents applications from getting such privileges, it was necessary to develop a **kernel driver** (a new feature since PikeOS 4.0) allowing us to select the PMC we wish to collect on the target.

The services provided by our driver are: 1) selecting the hardware events monitored by the hardware performance counters of local processor core; 2) starting the counters; 3) stopping the counters.

The user interface for these services, implemented in the library, uses the regular ioctl calls provided by PikeOS for drivers. For each core of the ARMv8 Juno board, about 50 events can be selected for 6 different counters. For each e6500 PowerPC core of the NXP QorIQ, about 250 events can be selected for 6 counters.

C. The Collector

The collector is implemented as a native PikeOS partition which role is: 1) to define a shared memory space where each instrumented application will save its collected measurements; 2) to configure specific measurement scenarios (like selection of events via the driver); 3) to launch the measurement campaign (relying on PikeOS scheduling schemes); 4) to transfer the content of the shared memory to the host computer, either at the end of the measurement campaign (preferred to keep time intrusiveness level low) or during the run (to allow huge data collection).

In case of a failure (e.g. a deadline miss, or an unexpected timeout that is causing the application to stop), the collector also transfer the current memory content, allowing us to perform a post-mortem analysis and debugging.

IV. METrICS INTERNAL OPERATION

As introduced in Section I, our measurement environment has to provide the user with **accurate** runtime and resource usage measurements while minimizing both the **intrusiveness** and the **adherence** to the hardware.

A. Intrusiveness Trade-off

A major challenge in performance monitoring 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 either the application or the RTOS.

Automated instrumentation tools commonly suffer from a trade-off between measurement granularity (from process level down to instruction level), time intrusiveness, and code intrusiveness. Enhancing one usually have a detrimental impact on the other two. In METrICS we chose to focus on **minimizing timing intrusiveness**, due to the fact that 1) we focus on time-critical and safety-critical applications where time determinism is of prime concern and 2) our major objective is the ability to characterize timing interference, and correlate them to shared hardware usage.

To minimize time intrusiveness, we used several development techniques that make METrICS as light as possible during the execution of the application under test.

Firstly, all initialization and external communications were implemented in the Collector and run outside of the operational scheduling. We configured the time-base and hardware performance counter registers to be used in user-space, avoiding time-consuming context switches associated with protected mode. Time-base and PMC accesses are performed in inline assembly code to minimize latency. Finally, the shared memory containing the sample collection is mapped into the memory of the processes to avoid accesses trough system calls.

B. Scheduling Policies minimizing Time Intrusiveness

In order to minimize the runtime intrusiveness on application code, the collector is executed outside of the operational scheduling. We distinguish three scheduling phases in the course of a measurement campaign implemented with ”scheduling schemes” in PikeOS:

- **SCHED_BOOT** during which only PikeOS and the collector are running in time partition 0 (always

schedulable) of PikeOS. During this scheduling phase, the collector performs the initialization of the measurement campaign, preparing the shared memory and the PMC of each core. Once the initialization is completed, the collector shifts to the MONITORING scheme.

- **MONITORING** during which the time-critical application partitions are scheduled according to their unmodified deployment scheme, using the collector's shared memory to store their collected measurements. During this scheme, the collector remains blocked on a semaphore from the application to notify for the end of the execution, and is therefore not schedulable. Upon receiving this notification, the collector shifts back to the SCHED_BOOT scheme.
- **SCHED_BOOT** during which again only PikeOS and the collector are running. During this phase, the collector performs the transfer of the collected measurements to the host using the MUXA protocol of PikeOS.

Using such a set of scheduling schemes allows us to minimize the time intrusiveness of the measurement environment regarding the applications. Indeed, all MUXA communications, system calls to the driver, and the collector itself are not running at the same time as the time-critical applications. It is therefore not necessary to dedicate a time slot for the collector, and the original static schedule of the application remains unchanged.

C. Transmission of collected measurements to the host

At the end of the execution the collector is responsible for transmitting the collected data to the host. This transmission is performed using the MUXA service provided by PikeOS [1]. This service offers multiplexed channels of communication between the target and host computers, for debug and monitoring purposes. It offers abstraction of the hardware communication link.

To avoid the regular issues with transmitting binary data through this channel (endianess issue, control/command issues), we transmit this data directly in ASCII format as the content of a comma-separated-value (CSV) file. Each sample contains information about timestamps, hardware performance counters, as well as task ID, core ID and probe ID.

We dedicated a MUXA channel for this communication (channel 4, port 1506) initially doing MUXA over UART. However we quickly reached the maximum throughput of the UART controller, so we had to switch to MUXA over Ethernet. The implication is that PikeOS has to support the Ethernet controller of the target board, and that one of the Ethernet ports has to be dedicated to MUXA (which itself is able to multiplex).

On the host side, a driving script is performing a telnet connection to the MUXA server to dump the collected data directly in a corresponding CSV file upon reception.

V. Evaluating METrICS accuracy and intrusiveness

The METrICS environment allows us to collect various measurements during the execution of safety critical applications, including execution time distribution and shared hardware resource access information. Rather than only extracting minimum, average and maximum values, the METrICS tool suite extracts the whole distribution of each measured data, allowing us to study the correlations between runtime and hardware resource usage.

The METrICS environment has currently been ported to the PikeOS 4.0, 4.1 and 4.2 RTOS, to the 32-bit NXP QorIQ P4080 [6] based on PowerPC e500mc, to the 64-bit NXP QorIQ T2080 [29] based on PowerPC e6500, to the 64-bit ARMv8 Juno Board [4] based on ARM Cortex A72 and ARM Cortex A53, as well as to simulation environments such as QEMU [7] or ARM FVP [3].

This Section will evaluate METrICS in terms of accuracy, precision and intrusiveness. Next sections will present an example of METrICS usage and focus more on the ability to correlate timings and resource usage, as well as its ability to perform timing interference characterization.

A. Selection of time measurement mediums

To be able to perform fine-grain timing, we need to rely on some kind of time measuring instrument. This time measurement medium could be either external, provided by any of the layers of the operating system or directly provided by the target processor as part of the instruction set.

The kind of events we wish to accurately measure includes complex functional chains (up to several seconds in avionics), runtime of individual tasks (with deadlines typically in the order of a few hundreds of milliseconds) or time spend in system calls (typically in the order of microseconds).

Each measuring medium relies on a software or a hardware mechanism that itself has a working period, thus limiting the obtainable **precision** to no less than this period. Also, each of these mediums actually consumes time to perform a measurement, making it impossible to accurately measure time below this measurement time **overhead**.

Additionally, for multi-core processors, clocks are not necessarily synchronous between all cores, introducing the concept of inter-core **clock offset**. If this offset does not remain constant (which could be the case if not connected to the same quartz oscillator (or clock PLL) or if the core is subject to dynamic frequency scaling), then it additionally introduce the concept of **clock drift**.

Using the APEX (avionic) personality of PikeOS, the operating system provides two system calls allowing us to measure time: `p4_get_time()` is provided directly by the PikeOS kernel, and returns the system time since boot in nanoseconds. `GET_TIME()` is provided by the APEX personality, and returns the system clock time, that is common to all processors.

However, being system calls, these time measurement mediums involve at least a context switch from the task to the operating system, and may involve switch(es) to privilege mode(s), depending on how the OS is handling system calls.

The expected overhead for such switches is more than 1000 CPU cycles, and relying on such calls for time measurement will simply prevent us to measure short events such as context switches and system calls themselves.

On the other hand, both the ARM-v8 and PowerPC ISA are providing low-level time measurement mediums. For instance, the e500mc/e6500 PowerPC provide two special registers that can be read with the `mfspr` assembly instruction: The `time base` register is a 64-bit register, set to 0 at board reset and incremented at the Platform Clock frequency, which is provided by a different PLL than the core clock frequency. The `time base` is thus 16 to 64 times slower than the Core clock frequency, 48 being a common prescaler ratio. An advantage of the `time base` is that it corresponds to a global system clock, synchronized on all cores. The `alternate time base` register is a 64-bit register, also set at 0 upon reset, that increments at every core clock cycle. No specific guarantees are provided in the documentation about it being synchronous in all the cores.

B. Evaluation of time measurement mediums

The resolution of time measurement mediums are provided in their respective documentation. To evaluate the overhead of the above-mentioned mediums, we set up experiments using each medium twice in a row. The time offset between the two measurements is an upper bound of the time overhead. Each measurement pair was performed 180000 times to ensure that each overhead is not subject to variability. The results evaluated on a 1.8 GHz e6500-based NXP T2080 with PikeOS 4.1 are summarized in Table I.

TABLE I. RESOLUTION (PERIOD) AND OVERHEAD

medium	layer	period	frequency	overhead
p4_get_time()	kernel	1 ns	n/a	240 ns
GET_TIME()	APEX	10 ms	n/a	10 ms
time base	register	48 cycles	37.5 MHz	1.67 ns
alt. time base	register	1 cycle	1.8 GHz	1.67 ns

As expected, system-call-based mediums have a much higher overhead than special-register-based mediums. The APEX version is clocked with the Time Partition tick, used to define application time windows. Such a low resolution medium has a huge impact on overhead, making it impractical for fine grain timing.

Both special register mediums exhibit a 3 cycles (1.67ns) overhead, the `alternate time base` version providing a much better precision. For this reason, we chose this time measurement medium as the preferred method of measurement for METrICS.

With regards to timing offset between cores, only the `alternate time base` does not provide a null offset guarantee. We measured this offset against the synchronized `time base` and evaluated it being below 200ns for core 0 with respect to the other cores, and within the measurement precision between cores other than core 0. If a very high precision for inter-core measurements is necessary, the method we used for this evaluation can also be used for calibration at boot.

Finally, none of the mediums showed a measurable drift among the 180000 runs.

C. Evaluation of a complete METrICS probe

A METrICS probe involves: 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 (thread and partition identifiers, core number); and 4) storing the collected information into the shared memory.

The intrusiveness of a METrICS probe in the source code is quite low, just adding a function call at the begin and the end of the code sequence to be monitored. Also, all the APEX system calls are automatically instrumented, requiring no further code modification. To do so, we overloaded the APEX function definitions with identical functions surrounded with our probes.

We also measured the intrusiveness in term of timing of a complete METrICS probe by performing successive calls to metrics probe the same way we did in previous section. Figure 2 presents the completion time results of such a probe, sorted over 180000 runs.

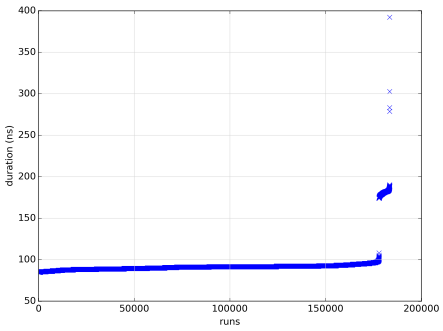

Fig. 2. Completion time of a METrICS probe over 180000 runs

The probing time varies from 85ns up to 392ns. For 97% of the runs the overhead is below 110ns. For 2.998% of the runs it is between 110ns and 191ns. And for 0.002% of the cases, it is above 191ns and up to 392ns.

More precisely, the first three steps of a METrICS probe are quite stable with ∼25ns to retrieve both the timing and the performance monitor counters, and ∼20ns to retrieve the thread-specific information. The dominant part corresponds to the storage in the shared memory, varying from 40ns to 347ns, probably because of cache effects.

As a consequence, the proposed measurement mechanisms presents a very limited overhead, and the highest achievable precision. Indeed the latency added by the full measurement probe remains in most cases shorter than the sole time measurement provided by the RTOS, and we used the fastest clock available in the target processor.

The next evaluation section will provide an example of usage of these METrICS probes for an example application.

VI. EXAMPLE OF METRICS USAGE

In this section, we provide an example of METrICS usage to evaluate how different multi-core deployments of a simple avionic-like application react with regards to timing interference. The goal would be to evaluate which of these deployments is less sensitive to timing interference.

The experiments conducted in this section were performed with METrICS running on top of PikeOS 4.1 on a 4-core PowerPC-based NXP T2080 hardware target. The application is running with the PikeOS ARINC-653 personality commonly used in the context of avionic applications.

A. Evaluated Application

We evaluated an in-house application: eDRON (embedded Directed Rotodrone Operated Network), that is guiding a fleet composed of four quadricopter drones along a preset flight route. The purpose of this application is to mimic a representative behaviour of an avionic application, while exercising classical ARINC-653 communication services and proposing several multi-core deployment options. The software architecture of eDRON is presented in Figure 3.

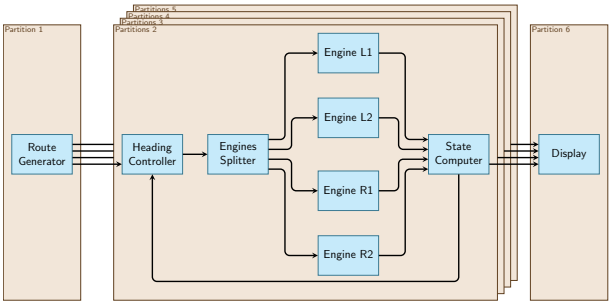

Fig. 3. Software architecture of the eDRON application

This application is composed of six ARINC-653 partitions. The first one sets up the preset flight route for all the drones, the last one displays all the drone positions, and each of the four remaining partitions is dedicated to pilot a particular drone. These later partitions are composed of 7 tasks, with most of the computation being performed in the four engine control tasks, each being dedicated to control the velocity for one of the four engine of a drone, so that it follows the preset route.

When mapping such an application on a multi-core processor, we first need to decide what will run in parallel. The application offers two obvious parallelization schemes: **inter-partition** parallelism where each core will deal with a single drone, running the velocity control tasks sequentially for each drone; and the **intra-partition** parallelism where each core will focus on one particular engine, dealing with each drone sequentially and running all sequential tasks on core 0. Some other parallelization options are available: for example parallelizing along the pipeline, or performing loop-level parallelization of the tasks, but those are beyond the scope of this paper as they require deeper modifications of the application.

These two deployments have advantages and drawbacks: The Amdahl's law [2] may limit the performance of the intra-partition version, while the inter-partition version will benefit from the Gustafson's law [17], running independent applications in parallel. However, with regards to timing interference, the intra-partition version offers a white-box context where the partition scheduling can limit the level of interference between known tasks. The inter-partition parallelism on the other hand corresponds to a black-box context where no easy control is possible to limit the interference level of another independent application.

B. Deployment evaluation

To evaluate the eDRON application with METrICS, we instrumented each task appearing in Figure 3 by adding a pair of begin / end probes around each task and partition. We then measured task runtimes while executing the application on one of the cores (single-core version), and later compared the results with both parallel deployments.

Note that while the full application is expected to be faster on a multi-core deployment, at the level of each task / partition, the runtime is only expected to vary due to possible timing interference. As a consequence, we expect to observe a slowdown at task level for the parallel versions.

Table II compares the runtime variability of a single Drone partition for three different deployments: a sequential / single-core deployment, and the deployments presented above with inter-partition or intra-partition parallelism.

TABLE II. EVALUATING DEPLOYMENT IMPACT ON RUNTIME AND VARIABILITY OF ONE DRONE PARTITION

parallelism	runtime (ms)				
	min	25%	median	75%	max
none	16.75	16.76	16.76	16.76	19.56
inter-partition	16.91	16.98	16.99	17.02	33.92
intra-partition	55.17	55.18	55.18	55.19	55.31

As expected, the single-core version is the deployment exhibiting the less variability with runtimes between 16 and 20ms, with a total execution time of 80ms to sequentially run the 4 drones. The multi-core deployment with inter-partition parallelism has similar lower bound and quartiles, but a much larger (x1.7) upper bound around 34ms. The deployment with intra-partition parallelism exhibits close to no variability, but with much larger runtimes of 55ms (x2.8).

We furthermore studied this later version as the increased minimum runtime was suspicious. We figured out that the extra runtime was spent during the system calls performing inter-task communications.

C. Communication evaluation

Within the eDRON application, data communication is performed through the ARINC-653 system call layer. This API, corresponding to the PikeOS APEX personality, allows us to perform both intra-partition communication and inter-partition communication using either buffer-based or fifo-based communication services as illustrated in Table III.

TABLE III. ARINC-653 COMMUNICATION SERVICES

level	type	write / read
intra	buffer	DISPLAY_BLACKBOARD()
		READ_BLACKBOARD()
intra	fifo	SEND_BUFFER()
		RECEIVE_BUFFER()
inter	buffer	WRITE_SAMPLING_MESSAGE()
		READ_SAMPLING_MESSAGE()
inter	fifo	SEND_QUEUING_MESSAGE()
		RECEIVE_QUEUING_MESSAGE()

METrICS allowed us, thanks to the instrumented system call layer to automatically collect runtime information relative to these communication services during the experiments we ran to build Table II.

The results corresponding to the deployment with inter-partition parallelism, that will serve as a reference, are presented as boxplots [34] in Figure 4 for both inter and intra partition communication results. The results corresponding to the deployment with intra-partition parallelism will later appear in Figure 5.

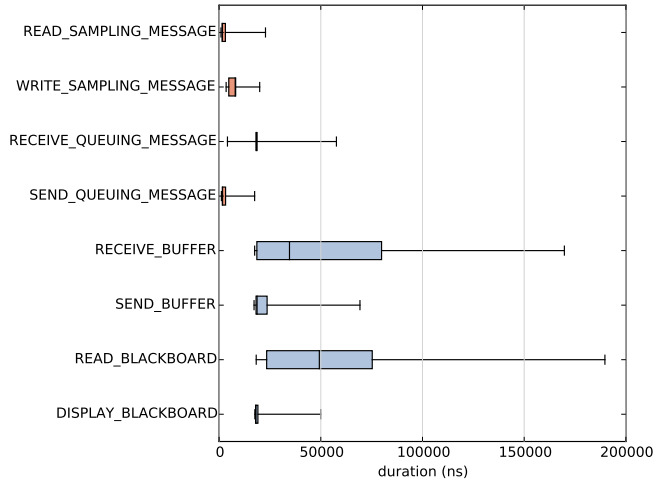

Fig. 4. Measurements of inter-partition communication (top) and intra-partition communication services (bottom) for the deployment with inter-partition parallelism

Intra-partition communications are much more costly than inter-partition communication for the deployment with inter-partition parallelism, especially for the receiving functions. This is expected, as running tasks sequentially introduces waiting time for communication receivers.

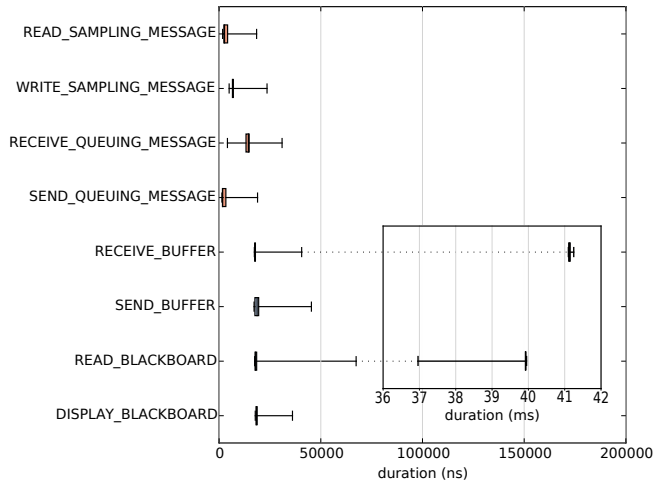

Fig. 5. Measurements of inter-partition communication (top) and intra-partition communication services (bottom) for the deployment with intra-partition parallelism

Figure 5 presents the communication time for the deployment with intra-partition parallelism. However, to be able to represent intra-partition communication times, we had to distinguish some outlier results. We therefore added a specific inset for both RECEIVER_BUFFER and READ_BLACKBOARD outliers with communication time above 36.9ms (36 965 107 ns).

Such high communication times occurs in 25% of the runs for READ_BLACKBOARD and in 49% of the runs for RECEIVE_BUFFER. With such a high occurrence rate, they are clearly responsible for the low performance of the deployment with intra-partition parallelism.

Further studying this strange timing behaviour, we figured out than the runtime of these system calls was mostly equal to their timeout value (The ARINC-653 layer specifies communication with timeouts). Strangely, the communication that reached their timeout were successful as well. Increasing the timeout value or reducing the data size did not decrease the phenomenon, but setting infinite timeout fixed the issue.

We reported this strange behaviour to SYSGO and this issue is now fixed in the latest 4.2 version of the RTOS.

D. Conclusion

Even tough the identified bug did not really allowed us to figure out which of the multi-core deployment is best with regards to interference, we have a proof of concept that METrICS can be used for such a study, and further can be used to identify software bugs with regards to timing.

In the next section, we will evaluate the number of total experiments that would be required to perform a full characterization of several deployments of the eDRON application while varying both the hardware target configuration and the application memory footprint.

VII. Dealing with the Design Space and Automation

A full characterization of a selection of multicore deployment of the eDRON application would require a number of experiments described in Table IV. It tests four deployment options including the three presented in previous section; three different options for the application memory footprint (fitting in L1 cache, in L2 cache or not fitting in caches); and whether or not to flush all caches between time partitions (a common practice in avionics to reduce variability).

TABLE IV. Number of runs for a full characterization

Application deployments	4
Buffer size	3
Cache Policies	2
Hardware Counter Selection	C_{41}^{2}
Number of iterations	1000
TOTAL RUNS	19.68 million

The target PowerPC architecture provides a selection of about 250 hardware events that could be measured with performance monitor registers. Among those events, we identified a set of 41 events related to the shared hardware resource the cores may compete on. Both the e6500 PowerPC architecture and the ARM-v8 architecture propose 6 performance monitor

counters, meaning that testing all possible pairs of counters would require $C_{41}^2/6$ different configurations.

As shown in Table IV, this represents a large number of experiments to run, and some form of automation is desirable. Therefore we developed a series of Python scripts running on the host computer to automate all aspects of running a series of experiments: selecting the right executable file for booting the target by tftp, rebooting the board using a debug probe, providing the collector with the right set of hardware counters to use, receiving the results with a telnet connection to the MUXA server, and storing the CSV file in a directory named after the experiment configuration and date. This automation infrastructure allows us to perform large measurement campaigns by iterating on the steps presented in figure 6 without frequent user supervision.

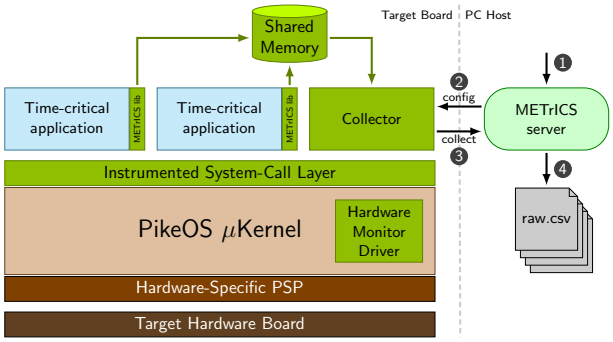

Fig. 6. Host-side automation server, performing 1) selection of target executable and test configuration 2) configuration of hardware counters to use 3) collection of measurements and 4) storage of result files.

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

VIII. Support for Analysis

Considering the large amount of data collected in the experiments of previous sections (over 110GB of data collected over 14 cumulative days of runtime), it would be great to also fully automatize the data mining, allowing us to analyze the collected results. This is however far beyond the scope of this paper.

With METrICS we aim at providing means to perform an expert-driven analysis. In such a context, we developed a GUI providing different ways of visualizing the collected data. In this section, we will present a portfolio of the available visualizations.

A. Underlying visualization technology

As the purpose of METrICS is to collect the full distribution of events (rather than only minimum / maximum values) to study correlations between timing and shared hardware resource contention, we need the same kind of visualizations as the ones used in the statistics domain.

As we are manipulating gigabyte-large datasets, a major concern for data mining is scalability. Some other important features will be rendering speed and interactivity, to ease the expert analysis.

In the domain of data science and big data, academic research usually often on Python language coupled with numpy [35] and pandas [26] for data analysis, coupled with matplotlib [18] and jupyter [33] for visualization.

An alternative for online and interactive data visualization is to rely on dedicated javascript libraries such as d3.js [8] or google charts [16]. These libraries render charts as SVG (Scalable Vector Graphics) which enable the user to interact with each element of the chart, typically with zooming or filtering ability.

Data visualization in METrICS involves both charts with millions of points (usually scatterplots) as well as charts with much fewer points (e.g. boxplots) but many filtering options. As a consequence, we opted for two different rendering options: pandas coupled with matplotlib for rendering static large-scale charts, and pandas coupled with d3.js for rendering interactively filterable data. All these visualization are bundled into a single custom Qt-based GUI using the pyside Python binding: xTRACT visualizer (expert Timing and Resource Access Counting Trace visualizer).

B. Visualization related to user probes

User probes are typically used to monitor task runtime and resource usage. Such information allow us to build up classical Gantt charts, effectively showing what is running in parallel, but it does not help to focus on the runtime variation caused by timing interference.

To better visualize runtime variability, we build for each user probe an histogram showing the distribution of observed runtimes during the successive runs, as depicted in Figure 7.

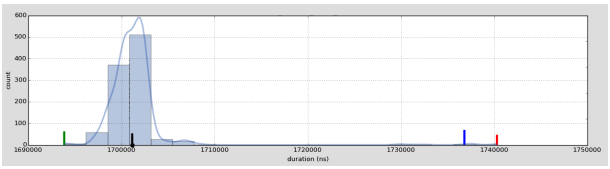

Fig. 7. Histogram of the drone partition runtime as appearing in xTRACT visualizer

The x-axis corresponds to the observed duration while the y-axis indicates how many time 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.

We also added colored vertical bar markers, corresponding from left to right on Figure 7 to the best case in terms of runtime, to the median, to the first iteration of the application (that frequently behave quite differently) and finally to the worst case execution time observed.

To figure out correlations between runtime and hardware resource access, we also build histograms with the collected Performance Monitor Counter data, as shown in Figures 8 and 9.

In these two figures, the colored vertical bar markers still correspond to the best, worst, median and first iteration case

with regards to runtime. In these figures, the x-axis corresponds to the number of accesses to a particular hardware resource. In 8, accesses seems to somewhat correlate with execution times, whereas it is not the case for 9.

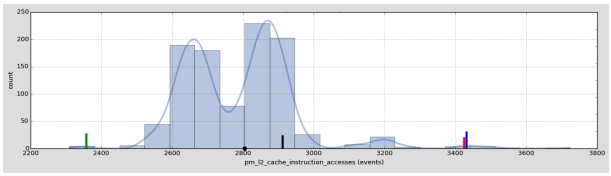

Fig. 8. Histogram of correlating resource accesses (L2 read cache accesses) as appearing in xTRACT visualizer

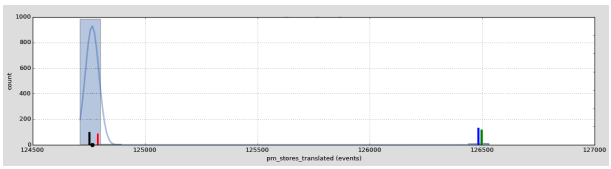

Fig. 9. Histogram of not correlating resource accesses (issued store instructions) as appearing in xTRACT visualizer

Potential correlations could be better observed with scatterplots such as the one appearing in Figure 10.

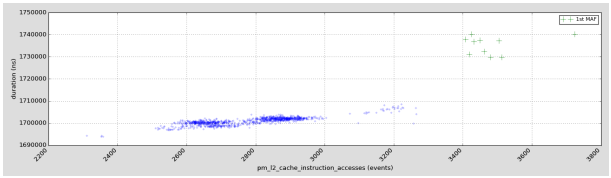

Fig. 10. Scatterplot showing linear correlation between runtime and L2 read cache accesses as appearing in xTRACT visualizer

Scatterplots allow to easily identify linear correlations. Each point of the scatterplot indicates that a particular run has been observed with a number of resource accesses equal to the value 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. In Figure 10 most of the points are on a line except the one corresponding to the first iteration that we highlighted with a specific symbol, confirming the correlation.

Another option could be to identify correlation among performance monitor counters to eliminate redundant information provided by correlated hardware resources (like the obvious redundant information of L1 cache misses versus L2 cache accesses) to reduce the experimental design space.

C. Visualization related to the instrumented syscalls

We also rendered various charts related to the probes automatically inserted around system calls.

These renderings allow the expert user to split the 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.

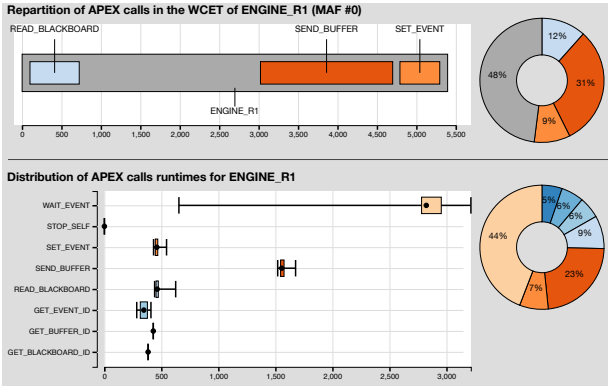

Fig. 11. Visualizing ARINC-653 syscalls in ENGINE_R1 task with xTRACT visualizer

For instance, the top charts of Figure 11 shows the repartition of APEX system calls in the ENGINE_R1 task of the eDRON application running on the 6-core Juno board ARM-v8 architecture. The x-axis corresponds to the time, and a gap can be observed between the SEND_BUFFER and the SET_EVENT system calls despite being called sequentially. This is due to the fact that the application is no more schedulable during system calls, and therefore at least two time-consuming context switches occur between the two function calls.

The bottom part of Figure 11 shows with boxplots the variability of the execution time of APEX system calls for different runs of the ENGINE_R1 task, actually showing that the RTOS is also affected by timing interference.

IX. Conclusion and Future Works

With the foreseen shift to mutli-core architectures in the time- and safety-critical domain, dealing with the timing interference issue inherent to multi-core processors becomes of prime concern.

Several solutions have been proposed in the literature [14], but quantitative evaluations are mostly missing. In this paper, we have presented METrICS, a toolsuite dedicated to perform fine-grain time and resource access measurements in safety critical systems, allowing to actually measure timing interference and search for the causes of these interference.

We evaluated METrICS in terms of timing intrusiveness, code intrusiveness and accuracy, and obtained cycle-level precision with an overall overhead similar to regular RTOS system calls to obtain just the timing information.

Beyond the layer running on top of PikeOS on the target system, METrICS is also a host-side server than can automatically drive large-scale test campaigns, and xTRACT visualizer, a GUI that can help the expert to analyze collected results.

A next step for us would be to use METrICS to perform a quantitative evaluation of the Deterministic Platform Solutions presented in the DPS survey [14], that aim at either controlling or regulating interference.

ACKNOWLEDGMENT

The research leading to this work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644080 (SAFURE).

REFERENCES

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

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, pages 483–485, Atlantic City, April 1967. ACM.

[3] ARM Limited. Fixed Virtual Platforms: FVP Reference Guide, Nov 2015.

[4] ARM Limited. Versatile Express: 64 Bit Juno r2 ARM Development Platform, Nov 2015.

[5] 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.

[6] Greg Bartlett. QorIQ P4080 Communications Processor design in 45nm SOI. In *IC Design and Technology, 2009. ICICDT '09. IEEE International Conference on*, pages viii–viii, 2009.

[7] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, pages 41–46, 2005.

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

[9] 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.

[10] 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.

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

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

[13] Lauterbach GmbH Germany. Trace32: RTOS debugger for linux. February 2017.

[14] 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.

[15] 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.

[16] GOOGLE. Google charts: Interactive charts for browsers and mobile devices. https://developers.google.com/chart/.

[17] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

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

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

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

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

[22] 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.

[23] 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.

[24] 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.

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

[26] Wes McKinney. pandas: a foundational python library for data analysis and statistics. In *PyHPC 2011: Python for High Performance and Scientific Computing*, Seattle, USA, November 2011.

[27] 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.

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

[29] NXP Semiconductors. T Series - QorIQ T2080 and T2081 communication processors, Feb 2017.

[30] 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.

[31] 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.

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

[33] Kluyver Thomas, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. Jupyter notebooks—a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing*, ELPUB, Göttingen, Germany, 06/2016 2016.

[34] J. W. Tukey. *Exploratory Data Analysis*. Behavioral Science: Quantitative Methods. Addison-Wesley, Reading, Mass., 1977.

[35] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: a structure for efficient numerical computation. *CoRR*, abs/1102.1523, 2011.

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

[37] 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.