BB-RTE: a Budget-Based RunTime Engine for Mixed and Safety Critical Systems

Sylvain Girbal, Jimmy Le Rhun

BB-RTE: A BUDGET-BASED RUNTIME ENGINE FOR MIXED & 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

***Abstract*—The safety critical industry is considering a shift from single-core COTS to multi-core COTS processors for safety and time critical computers in order to maximize performance while reducing costs.**

In a domain where time predictability is a major concern due to the regulation standards, multi-core processors are introducing new sources of time variations due to the electronic competition when the software is accessing shared hardware resources, and characterized by timing interference.

The solutions proposed in the literature to deal with timing interference are all proposing a trade-off between performance efficiency, time predictability and intrusiveness in the software. Especially, none of them is able to fully exploit the multi-core efficiency while allowing untouched, already-certified legacy software to run.

In this paper, we introduce and evaluate BB-RTE, a Budget-Based RunTime Engine for Mixed & Safety Critical Systems, that especially focuses on mixed critical systems. BB-RTE aims at guaranteeing the deadline of high-critical tasks 1) by computing for each shared hardware resource a budget in terms of extra accesses that the critical tasks can support before their runtime is significantly impacted; 2) by temporarily suspending low-critical tasks at runtime once this budget has been consumed.

I. INTRODUCTION

Safety-critical applications are usually characterized by stringent real-time constraints, making **time predictability** a major concern with regards to the regulation standards [11], [12], [18] of the safety-critical and time-critical industries.

The industry is nowadays considering a shift from single-core COTS (component off-the-shelf) to multi-core COTS processors for safety-critical and time-critical products. Such a shift is appealing both in terms of performance as well as in terms of size, weight and power (SWaP) [4]. It also fits with the exponential growth in terms of performance requirements in the embedded domain.

However multi-core processor architectures are introducing new sources of time variations, and the solution providers can no longer rely on resource over-provisioning to enforce time predictability. As a consequence, the industry is facing a trade-off between performance and predictability [14], [16]. As depicted in Figure 1, multi-core processors are characterized by shared hardware resources such as some levels of caches, the interconnect, the main memory or I/O controllers.

At hardware level, concurrent accesses on theses resources are arbitrated, introducing jitter at application level defined as

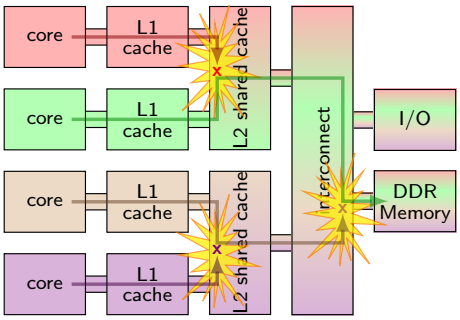

Fig. 1. Timing Interference in multi-core architectures

timing interference [8]. Such interference, caused by electronic competition on shared hardware resources, are breaking the timing isolation principles required by the regulation standards [11], [12], [18] of time-critical software.

Several papers [5], [17] have studied the impact of timing interference on the Worst-Case Execution Time (WCET). Their authors have shown that not trying to tackle the interference problem leads to a performance loss for worst-case that can go far beyond the expected performance gain of using a multi-core processor.

In a survey on **Deterministic Platform Solutions** [7] the authors presented various solutions allowing some level of time-deterministic usage of non-deterministic hardware platforms. Two families of solutions are presented: **control solutions** that aim at eliminating all interference by restricting the usage of the hardware platform, and **regulation solutions** that are focusing on keeping the impact of timing interference below a harmful level.

By introducing over-provisioning or complex runtimes, control solutions such as Marthy [13] or deterministic adaptive scheduling [6] fail to efficiently exploit the multi-core performance. Other solutions relying on execution models restrict too much the usage domain. For instance, The AER model [8] is easily applicable to distributed memory systems but lacks strong guarantees with regular shared memory systems.

Regulation solutions, on one hand, offer a better performance efficiency but lack the strong guarantees required by domains such as avionics. On the other hand they are perfectly adapted to less critical environments such as mixed-critical systems where high-critical tasks are running conjointly with low-critical tasks. In such systems, regulation solutions

offer the possibility to degrade low-critical tasks to guarantee the timing behavior of high-critical tasks. To provide time guarantees, regulation solutions usually rely on budgeting the time with early deadlines or shifting time windows [15].

In this paper, we present the **Budget-Based RunTime Engine**: a regulation solution relying on budgeting the number of shared hardware accesses to guarantee time properties. This is somehow similar to Memguard [21], [19] that pre-allocate application bandwidth requirement in term of memory accesses, but BB-RTE is especially focusing on the number of accesses, is extended to every shared hardware resource in the processor, and is performed per timeslot instead of as a whole.

To be able to do so, first we perform at design time an automatic standalone characterization of the critical applications to figure out the available budgets; and second we rely on this budget to take scheduling decision about non-critical applications at runtime.

The paper is organized as follows: The principles of BB-RTE are presented in Section II and the full characterization and monitoring process is presented in Section III. In Section IV, we present the hardware target we considered, as well as the monitoring facilities we relied on. The mixed-critical software prototype is presented in Section V with its associated timing requirements. Finally, Section VI proposes an evaluation of the full process associated with the Budget-Based Runtime Engine, followed by a conclusion section.

II. Budget-Base RunTime Engine: Principles

The principles of the approach consist in determining, per timeslot, a maximum budget to allocate to low-critical applications in terms of resource accesses. When this budget is spent, low-critical applications are suspended until the next timeslot, as depicted in first timeslot of Figure 2.

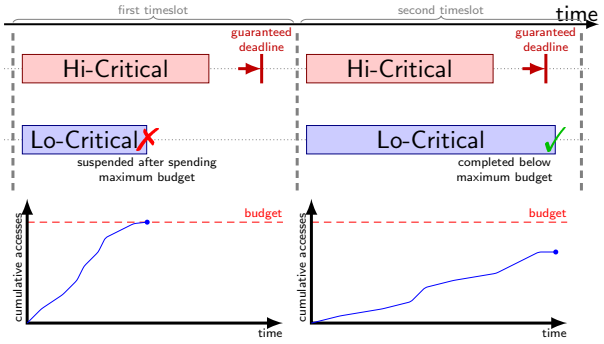

Fig. 2. BB-RTE Principles

The budget is computed per timeslot and per hardware resource to ensure that the critical applications will be matching their deadline in each particular timeslot.

During the second timeslot of Figure 2, the low-critical applications manage to terminate without spending the whole allocated budget, again guaranteeing that the critical applications will match their deadlines during this timeslot.

With such a process based on budgeting, the major challenge consists in **determining the budgets** that guarantee the time behaviour of high-critical applications.

III. Process to determine and ensure budgeting

The process to determine resource budgeting, depicted in Figure 3, is performed with two major steps.

First, both the architecture and the high-critical applications are characterized in an **offline characterization** step with regards to every hardware resources 1) to determine the total available resource budget; 2) to quantify the resource requirements of the high-critical task; 3) to figure out the maximum level of extra accesses to the resource before hampering the high-critical tasks.

Second, in a **runtime regulation** step, from the characterization information is inferred the maximum number of resource accesses allowed for non critical tasks, these tasks being suspended by a **Budget-Based RealTime Engine** once they reach this maximum number of total access during a given time slot.

A. Hardware characterization & budgeting

As introduced in Figure 3, in the first sub-step of the offline characterization phase, we perform a characterization of the target hardware platform. This hardware characterization consists in defining a set of low-level (assembly code) Stressing Benchmarks. Each of these stressing benchmarks is responsible for stressing a particular hardware resource of the selected multi-core target, by multiplying the number of accesses to this particular resource.

By progressively stressing each resource while monitoring both the execution time and the effective number of access to this resource thanks to Performance Monitor Counters (PMC) [20], we are able to determine the maximum available bandwidth in terms of access to this resource, and that corresponds to the **total available budget** for the resource.

By iterating over all the potentially shared hardware resource, we obtain a vector of such total budgets that fully characterize the hardware limitations of the selected platform.

B. Critical application characterization & budgeting

During the second characterization sub-step of Figure 3, we are characterizing the usage made by high-critical applications of the shared hardware resources. To do so, we run the high-critical applications concurrently with the stressing benchmarks described above, progressively increasing the stressing level, and only monitoring the effect in terms of runtime of the high-critical applications.

It allows us to extract two different kinds of information: First the **required per-resource budget** needed by our high-critical applications; and second, the level of extra resource access supported by our high-critical applications before being significantly slowed down. This **supported extra accesses** is the access budget that can be safely used by the low-critical applications.

The process to determine the acceptable level of slow-down and the associated extra access budget is depicted in Figure 4. The y-axis represents, during the current timeslot, the maximum observed runtime of the monitored application. The x-axis represents the extra access load performed on the associated hardware resource by the stressing benchmarks.

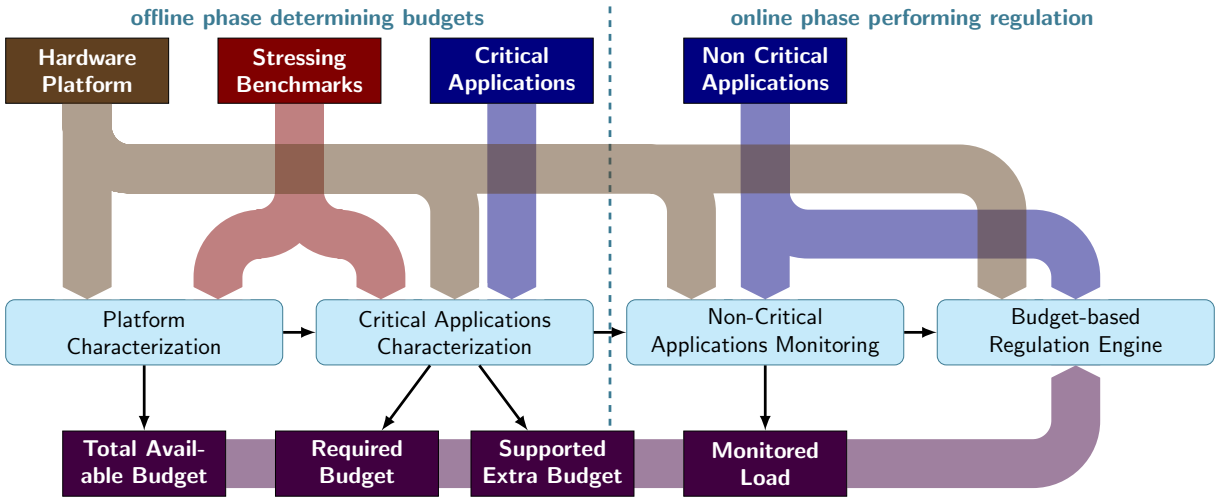

Fig. 3. Timing integrity process for mixed time-critical systems

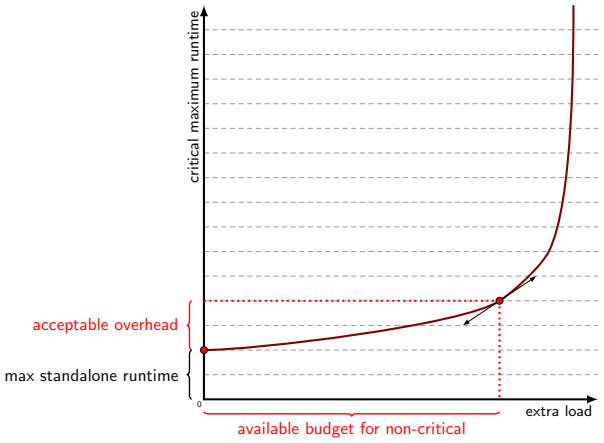

Fig. 4. Determining an acceptable level of slowdown and the associated extra access budget

The leftmost point in the chart corresponds to the application running alone in isolation (with a null stressing benchmark activity). It therefore corresponds to the classical WCET of the application running in isolation. The rightmost point in the chart corresponds to a permanent maximum load from the stressing benchmark actually preventing the monitored application to access the required resource.

Selecting an acceptable level of slowdown and the associated extra access budget is performed by selecting a point on the chart. The projection of this point on the y-axis then corresponds to the acceptable level of slowdown and the projection on the x-axis directly provides the extra access budget available for non-critical applications with an acceptable impact on critical applications.

We used two different techniques to select this point: First by directly selecting a maximum acceptable slowdown, but doing so for every hardware resource led us to too much over-provisioning, failing to fully exploit the multi-core efficiency. Second, we defined a maximum slope for the curve. This second solution allowed us to vary the level of slowdown relatively to the shape of the curve, focusing on the hardware resources the high-critical application was the most sensitive to.

By repeating this procedure for each shared hardware resource, we obtain again a budget in the form of a vector of both the required amount of access by the critical application (the leftmost point of the curve) and the number of supported extra access (the selected point of the curve).

Characterization steps involve large-scale experimentation due to the limited monitoring resources of multi-core processors. These architectures usually propose from tens to hundreds of hardware events that can be monitored, but only allow to monitor a few PMC at a given time (6 for ARM-v8 architectures for instance). As a consequence, testing all the countable PMC events involves many days of experimentation, and this characterizations phases have to be performed off-line. Anyhow this characterization only needs to be performed once per critical application for a particular hardware target, and such a characterization could also provide useful informations for the qualification or certification documents.

C. Budget-Based RunTime Engine (BB-RTE)

Once the characterization phases are over and all budgets have been gathered, both high-critical and low-critical applications can be deployed on the hardware target together with the runtime engine. As shown in Figure 3, the online regulation phase happens during the execution and consists in two sub-steps.

First, using the same process as the characterization phases, the low-critical applications are monitored with PMC counters, this time not to compute a budget, but to monitor the load in terms of hardware resource accesses.

Second, the BB-RTE runtime engine compares this load with the maximum extra budgets to decide if, during the current timeslot, non-critical task may continue executing or if they need to be suspended until the next timeslot. Doing so makes sure the slowdown of critical tasks does bot hamper their ability to match their deadlines.

One of the specific challenge of the BB-RTE runtime engine for time-critical systems is that time intrusiveness of

the associated monitoring features has to be kept minimal, not to bias the time characterization results. Also the BB-RTE itself should make a minimal usage of shared hardware resources to not impact the resource access budgets. The final intrusiveness footprint of both the monitoring features and the BB-RTE engine will be presented in the result section.

IV. Experimental Setup

The BB-RTE engine has been developed on top of PikeOS [1] which is both a hypervisor and a real-time operating system relying on partitioning.

We relied on the METrICS toolsuite [9] to perform the characterization steps and implemented the runtime engine as a native PikeOS partition, altering the scheduling in real-time as the monitoring information is gathered.

A. Target Architecture

The results presented in the Section VI of the paper were evaluated on an ARM Juno board [3] embedding a big.LITTLE architecture composed of a cluster of 2 high-performance Cortex A72 cores and a cluster of 4 more predictable Cortex A53 cores. The block diagram of the architecture is presented in Figure 5.

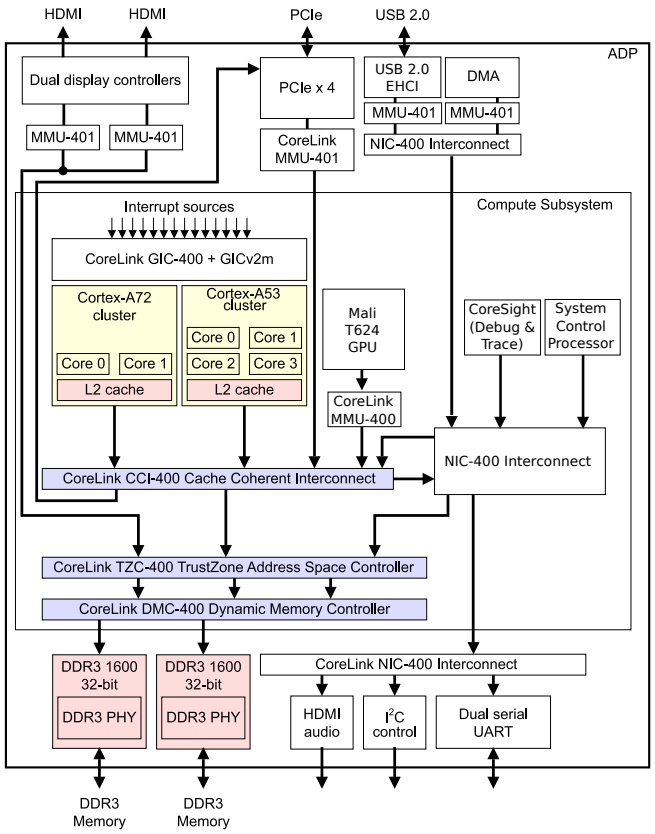

Fig. 5. Simplified block diagram of the ARM Juno Board

The highlighted parts in the figure correspond to the shared memory path from the cores towards the main memory that are prone to timing interference. Other initiators like the MALI GPU, the PCI express links and the DMA controllers can also create contention on these shared resource, but for this paper we will only focus on memory contention from the cores.

Alongside this memory path, the shared hardware resource are: The shared L2 cache that is shared by all the cores of each cluster. The CCI-400 cache-coherent interconnect (a high-bandwidth crossbar interconnect combining routing and coherency functions), and two DDR3 controllers served through the AMBA-compliant DMC-400 memory controller.

TABLE I. JUNO Board memory structure sizes

	A72 cluster	A53 cluster
Number of cores	2	4
IL1 cache size	48KB	32KB
IL1 cache line size	64B	64B
IL1 associativity	3	2
DL1 cache size	32KB	32KB
DL1 cache line size	32B	64B
DL1 associativity	2	4
L2 cache size	2MB	1MB
L2 cache line size	64	64
L2 associativity	16	16

The sizes of the memory structures alongside this path are presented in Table I. This information will be useful when trying to design stressing benchmarks dedicated at stressing a particular hardware resource.

B. Monitoring facilities

In Section III, we stated that the budget characterization is performed by using Performance Monitor Counters (PMCs).

The ARM-v8 architecture includes a Performance Monitors Unit (PMU), a non-invasive resource primarily used for debugging that provides information about the internal operations in the core. It includes a 64-bit cycle counter and 6 performance monitor counters able to count the occurrence of around 60 different events.

Among these events we selected a subset of 13 events that correspond to either private or shared hardware resource composing the memory path, listed in Table II. The description appearing in the table is the level of details provided by the documentation on each countable event.

TABLE II. Hardware events measured with Performance Monitor Counters

Performance counter	Counting
inst_retired	Instruction architecturally executed
cpu_cycles	Cycles
L1I_cache	Level 1 instruction cache access
L1I_cache_refill	Level 1 instruction cache refill
L1D_cache	Level 1 data cache access
L1D_cache_refill	Level 1 data cache refill
L1D_cache_wb	Level 1 data cache write-backs
L2D_cache	Level 2 cache access for data
L2D_cache_refill	Level 2 cache refill for data
L2D_cache_wb	Level 2 cache write-backs for data
bus_access	Bus access
mem_request	Memory access
prefetch	Linefill because of prefetch

A trial and error experimental process was therefore necessary to understand the exact meaning of each of these

counters. `inst_retired` corresponds to the number of executed instructions whereas `cpu_cycles` corresponds to the number of CPU cycles measured so far. Therefore the ratio of the two correspond to the regular instruction per cycle (IPC) measurement commonly used to determine an application performance.

The events `L1D_cache`, `L1I_cache` and `L2_cache` indicate the number of accesses to the L1 data cache, L1 instruction cache and L2 cache by load/store requests respectively. The `refill` counterparts indicate how many time a cache line was obtained from a higher memory structure (e.g. the number of time a cache line was provided to the L1 cache by the L2 cache) so it is a good measure of the cache misses that indicates that a higher memory level in the datapath is accessed.

The `write-back` counterparts indicate how many time a modified cache line was written back to higher level memory so that the local cache line could be freed to fit a new data. This write back traffic, not appearing directly in the source code could be a significant part of the memory traffic.

The events `bus_request` and `mem_request` are respectively counting the requests on the local bus connecting core and caches, and the number of requests leaving the core to be send to the CCI-400 interconnect. This local bus traffic does not only correspond to memory accesses in the code, but also to instruction fetches and to the coherency traffic.

Finally, `prefetch` indicates the number of time a cache line of the L2 cache is filled due to an automatic hardware prefetch of the cache line. This feature has proven to be very problematic for our stressing benchmarks later presented in this section as the prefetching was limiting the stress level, as shown in the evaluation section of the paper.

Some of the other hardware resource appearing in Figure 5 also provide some debugging support, but the documentation if often lacking or only available under NDA, and restricted to third-party providers of debugging probes such as Lauterbach. We definitely want to monitor these SoC-level events in the future, but we considered them out-of-scope for this paper.

C. Target Environment

In order to measure precise execution times and to sample hardware performance counters, we used our Measurement Environment for Time Critical Software (METrICS). Its main concept is to sample performance counters (including the cycle-count register) with a very short timing overhead, before and after the code sequences to monitor.

The METrICS toolsuite is composed of several elements: a kernel driver used to configure the hardware performance counters (as this requires privileged instructions), a library providing measurement probes to the application, and a Collector performing various initialization and transmission of measurement results. We heavily modified this latter component to include the part corresponding to the Run Time Engine. In addition, the METrICS suite includes host-side scripts and tools for measurement campaign automation, post-processing of raw data, and visualization.

The latency of a METrICS probe has been evaluated to be less than 392ns at worst. This whole probe thus has a comparable latency to a system call, and is precise enough considering the millisecond deadlines we have in the mixed-critical prototype presented in Section V.

The initial behaviour of the collector component was to run out of the monitored application operational cycles, to minimize the impact it had on the application timings. More precisely, it was 1) performing initialization and performance counter selection prior to running the monitored applications, 2) being completely suspended during the application operational cycles, 3) being activated again at the end of the application to dump the collected data to the host.

As appearing on Figure 6, we transformed in the BB-RTE context the collector component into the runtime engine. It is now also running during the application operational cycle to monitor resource usage, and to suspend low-critical tasks when they have consumed all their budgets. In the experimental context, we kept the third dumping phase to also collect the applications runtime and performance counter data including the number of deadline misses observed during the applicative phase.

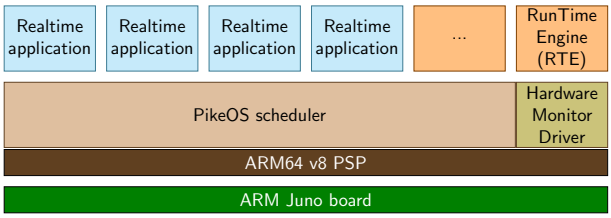

Fig. 6. METrICS infrastructure in the context of BB-RTE

The application deployment also appear in Figure 6. We dedicated the two A72 cores to running both the operating system and the runtime engine. Several tests have shown that it was more efficient to run the RTE on the same core as the operating system due to the large number of system calls required to alter the scheduling. We therefore reserved the more deterministic A53 cores to the applications. Further deployment details for this cores will be provided in the evaluation section.

D. Stressing benchmarks

The mixed-critical applications we will use for the evaluation are described in Section V, but the characterization steps described in Figure 3 also involve stressing benchmarks.

These benchmarks are simple applications performing repeated accesses to a shared resource. This involves executing a number of load or store instructions with regards to a memory region or a memory-mapped peripheral. In order to focus the stress applied by these benchmarks on a particular resource, we developed them in assembly language, thus allowing the greater control on the low-level behaviour of the stressing benchmark.

The accesses are parametrized in four ways: the direction of the transfer, the amount of data transferred, the address offset between consecutive accesses, and the repetition rate. The first parameter is either read or write. The second parameter, representing the buffer size, spans from 1KB to 2MB in power of 2 increments. This allows us to experiment the behaviour

of data fitting or not in L1 and L2 caches. The address offset, called stride, is either 4 bytes for continuous accesses, 32 bytes to stress A72 L1 cache lines, or 64 bytes to stress A53 L1 and L2 cache lines. To vary the amount of accesses performed in a given time, we add a configurable number of NOP instructions (modeling calculations performed in isolation) per load or store instruction. This parameter spans from 0 (full stress) to 100 NOPs per access.

V. Mixed Critical Prototype

To evaluate the Budget-Based runtime Engine, we set up a multi-domain mixed-critical software prototype composed of a high-critical application representative of the avionic domain, and low-critical control-command applications from the industry domain. The characterization of these applications as well as the evaluation of these applications running concurrently while supervised by the BB-RTE run-time engine will be provided in Section VI.

A. High-critical Application: Flyance

As the high-critical application, we selected Flyance, a mark-up Flight Management System (FMS) application mimicking the real-time behaviour of a regular FMS from the avionics domain.

The purpose of a Flight Management System (FMS) in modern avionics is to provide the crew with centralized control for the aircraft navigation sensors, computer-based flight planning, fuel management, radio navigation management, and geographical situation information. Taking charge of a wide variety of in-flight tasks, the FMS allows to reduce the workload of the flight crew.

The FMS is especially responsible for services that allow in-flight guidance of the plane. From pre-set flightplans (take-off airport to landing airport), the FMS is responsible for plane localization, trajectory computation allowing the plane to follow the flightplan, and reaction to pilot directives.

Flyance is composed of 25 time-critical tasks that are regrouped into different task groups as presented in Figure 7. The Sensors task group is in charge of generating all the localization data from various sensors. The Localization task group is in charge of computing the most probable position of the aircraft (BCP) by merging sensor information with different trustworthiness levels. The Nearest Airports task group continually builds a list of the nearest airports during the flight. The Flightplan task group is in charge of managing and processing modification requests on the flightplans that are pre-set routes used to guide the airplane. The Trajectory task group aims at computing both lateral and vertical profiles for the three flightplans set by the flightplan task. The lateral profile is composed of waypoints as well as leg information (path before, after and between the waypoints). The vertical profile provides altitude information (cruise altitude interceptions, crossing altitudes and slope angles) as well as performance information (estimated time of arrival, estimated fuel on board).

The FMS application also embeds a large Navigation Database that does not fit in any cache structure. It is both linearly and regularly accessed by a task from the Nearest airport task group, as well as randomly and sporadically accessed by tasks of the Flightplan task group. Accesses to this database in the main memory is very timing interference prone.

All the tasks composing the FMS have stringent real-time requirements presented in Table III. Additionally, aperiodic tasks have 100ms deadlines and can not be activated more than twice every 200ms.

TABLE III. Flyance timing requirements

Task	Period	Deadline
SENS_{C1}	200 ms	200 ms
LOC_{C1}	200 ms	200 ms
LOC_{C2}	1600 ms	1600 ms
LOC_{C3}	1200 ms	1200 ms
LOC_{C4}	1000 ms	1000 ms
TRAJ_{R1}	200 ms	200 ms
TRAJ_{R2}	300 ms	200 ms
TRAJ_{R3}	300 ms	200 ms
NEAR_{P1}	1000 ms	1000 ms

During the evaluation, one A53 core of the Juno board will be dedicated to running the Flyance application.

B. Low-critical Application: BiQuad

As the low-critical real-time application for the mixed-critical prototype, we selected Biquad: a control-command application implementing bi-quadratic on some streamed data acquired from analog sensors. Such an application is representative of classical distributed control system applications commonly found in the industry.

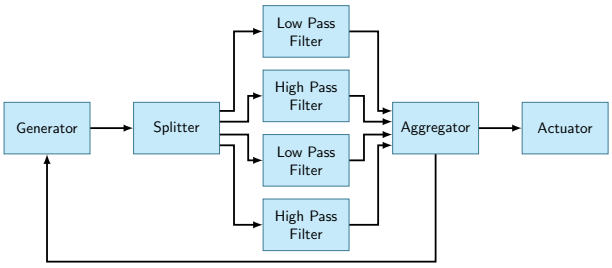

Fig. 8. Software architecture of the BiQuad application

The software architecture of the application is presented in Figure 8. BiQuad is composed of eight tasks: the Generator producing input data, either through generation on the first pass, or iterating on the received data on the next passes; the Splitter routes the data to the filtering tasks; the four Filters respectively apply their bi-quadratic filter on the data; and the Aggregator fuses the filtered data and sends it back as feedback to the generator task. The fused data is also periodically driving the Actuator task.

The timing requirement of this application is uniform across all the tasks: each task has a period of 200ms and a deadline equal to the next period.

For the evaluation, several instances of the BiQuad application can run at the same time, each running on a different A53 core.

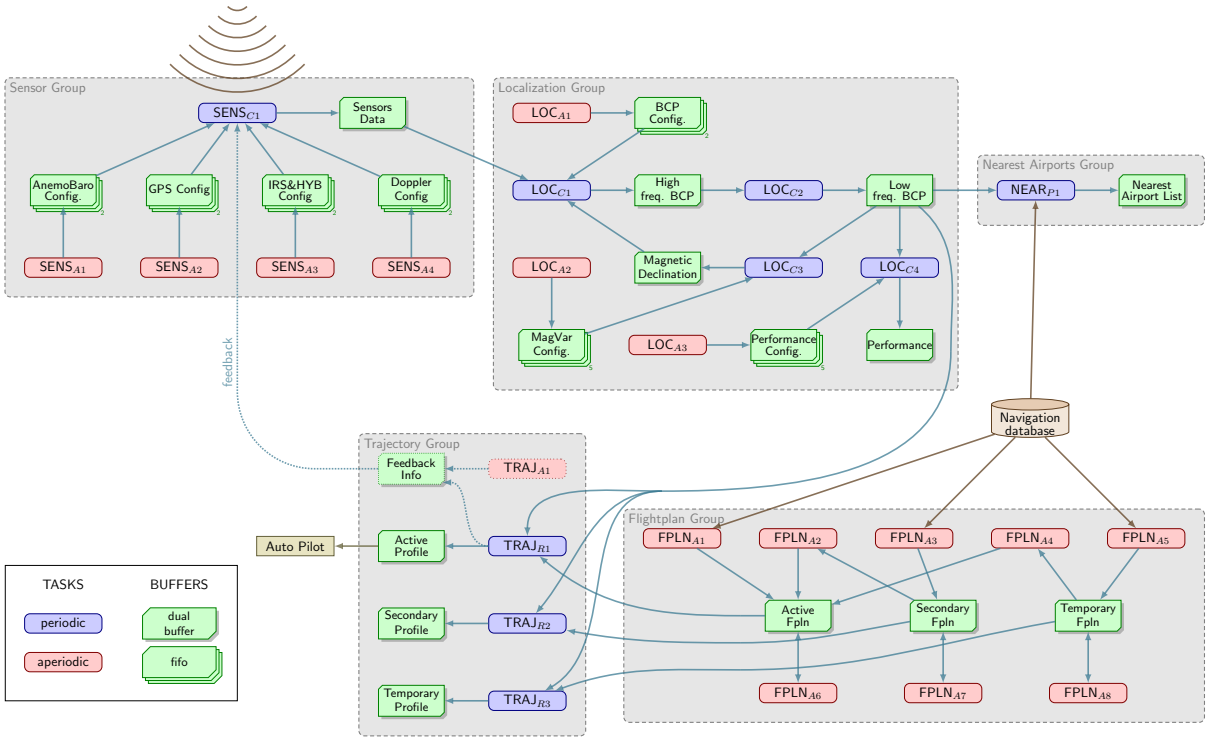

Fig. 7. Software architecture of the Flyance application

VI. Evaluation

This section first reports the budget characterization results obtained following the process described in Section III and summarized by Figure 3. Then, it presents an evaluation of the runtime engine, by comparing the results in terms of deadline misses compared to an unregulated run of the same set of applications.

A. Platform characterization and Total available budget

Performing the hardware platform characterization of Figure 3 involves concurrently running stressing benchmarks until some saturation phenomenons are observed.

On the first A53 core, we run a monitored stressing benchmark with a fixed number of iterations and a full stressing profile (zero NOPs). On the other 3 A53 cores, we run the stressing benchmark as an infinite loop, starting it before the monitored one to be sure that the monitored benchmark is run under stress. For these stressing cores, we are varying the buffer size and the stride concurrently with the monitored core. We are also varying independently the number of NOPs (and therefore the amount of concurrent access to the resources).

By varying the stride, we impact which memory structure and therefore which part of the architecture will be exercised: With a 4-byte stride, we will will exercise the L1 cache maximizing the number of L1 hits, while strides larger than the L2 cache line size will mostly produce L1 and L2 cache misses, actually exercising the DDR controller and the memory.

Table IV shows the runtime results of such a characterization, varying the buffer size (how well the data will fit in the different cache structures), as the stressing level varying to maximum stress (no NOP instruction) to no stress (no load instructions).

TABLE IV. Run-time variation observed on the monitored core with a 4-byte stride

		runtime (cycles)				
Buffer size	stress	min	25%	median	75%	max
16384	none	12338	12352	12360	12369	13101
16384	max	12338	12352	12360	12369	13175
32768	none	24680	24786	24839	24959	26129
32768	max	24679	24785	24838	24960	26272
65536	none	49340	49951	50300	50338	52499
65536	max	49339	49531	49752	50759	52795
524288	none	393962	394096	394219	394633	417123
524288	max	393978	394535	394785	395224	421488
1048576	none	789994	791536	792055	792817	827872
1048576	max	787995	791789	793538	795033	823978
2097152	none	1599528	1601997	1603697	1606266	1751393
2097152	max	1577078	1603203	1605720	1607560	1732193

Selecting a 4-byte stride puts the L1 data cache under pressure, but this cache structure is private to each core. As a consequence, results presented in Table IV show very small variations while increasing the number of NOPs, the runtime only varying as the buffer size grows. Due to the high L1 hit ratio, having large buffer size not fitting in the L1 does not impacts the performance either.

Selecting larger stride values allows us to characterize further hardware components shared by all the A53 cores starting with the L2 cache and even with the A72 cores beyond the CCI 400.

As we were limited to count core-related events with the Performance Monitor Counters, we mainly focused on the level 2 cache, that is also the first hardware memory resource shared by all the cores of the A53 cluster.

TABLE V. RUN-TIME VARIATION OBSERVED ON THE MONITORED CORE WHILE PERFORMING L2 CACHE MISSES

		runtime (cycles)				
Buffer size	stress	min	25%	median	75%	max
16384	none	818	826	832	839	2883
16384	max	818	824	831	839	4975
32768	none	1728	1866	1902	1955	9739
32768	max	1734	1883	1929	2006	7382
65536	none	6772	8237	8885	10131	19378
65536	max	6720	9080	9771	10777	20170
524288	none	79625	107638	108660	111344	179419
524288	max	97018	109665	110268	110897	177007
1048576	none	169753	210845	214317	229595	366017
1048576	max	207393	227506	237746	242661	353066
2097152	none	478261	522073	526574	529720	725379
2097152	max	500723	547663	551346	553397	705256

Table V presents the running time distribution on the monitored core while varying again the buffer size and the number of NOP instructions, to compare standalone versus maximum stress deployments.

Whereas the results should allow us to observe timing interference with an expected runtime variability of fully-stressed versus standalone larger than x4, we still observe very little difference between the two standalone and stressed versions.

Further tests, including unrolling the stressing benchmark loop to further increase the load or store ratio did not significantly change the above results. We therefore focused on the results corresponding to the maximum stress condition (2MB buffer size, with 0 NOP per iteration on the unrolled version) and studied the associated hardware counters presented in Table VI.

TABLE VI. PERFORMANCE MONITOR COUNTERS RELATED TO THE L2 CACHE UNDER STRESSING CONDITION

counter	min	median	max
l2d_cache	33645	34654	35129
mem_request	32473	32636	32863
prefetch	30304	31716	32001

In the worst case, out of the 35K data accesses to the L2 cache, 32K accesses are going to the external memory interface, meaning that our configuration successfully maximizes the number of L2 cache misses in order to maximize the interference on the interconnect. However, we can also observe 32K occurrence of prefetch, meaning that the hardware prefetcher successfully manages to capture the access pattern of our stressing benchmark and was able to anticipate nearly all of the external memory accesses. As a consequence, our stressing benchmark fails to effectively continuously stress the hardware resource.

In such a configuration the hardware prefetcher, instead of worsening runtime variability, smooths the contention on memory accesses. This seems to be a case of unexpected positive contribution to determinism from a dynamic, non-deterministic mechanism.

We tried to disable the hardware prefetcher, unfortunately we observed that it is periodically turned back on. We suspect the System Control Processor to be responsible for that. However, from our real time system, we have no control nor access on this particular core.

An alternative approach would be to develop another stressing benchmark, performing accesses that are more difficult for the prefetcher to predict. This would for example involve pseudo-random address increments.

B. Critical application characterization and Extra supported budget

The characterization of the critical application was performed by running the FMS application on the first A53 core. Again, thanks to METrICS, we instrumented the FMS application by inserting probes around each task composing the application, and at the level of a full operational cycle from the sensor task to the trajectory computation.

We first ran the the FMS application standalone, with the other core being idle, to collect the budget in terms of L2 accesses required by each timeslot of the application. Contrary to the stressing benchmarks, that are regular in their behavior and throughput requirements, the observed runtimes and resource requirements vary a lot with the FMS application, as all the tasks are not triggered in every 200ms timeslots due to heterogeneous periods.

The LOC_C4 task for instance is only executed in 1 out of 5 periods, whereas the LOC_C1 task is executed every period. As expected the timeslots corresponding to the least common multiple of all the periods, during which all the tasks must execute, is the one with the larger observed runtime and L2 access count, as pointed out by Figure 9.

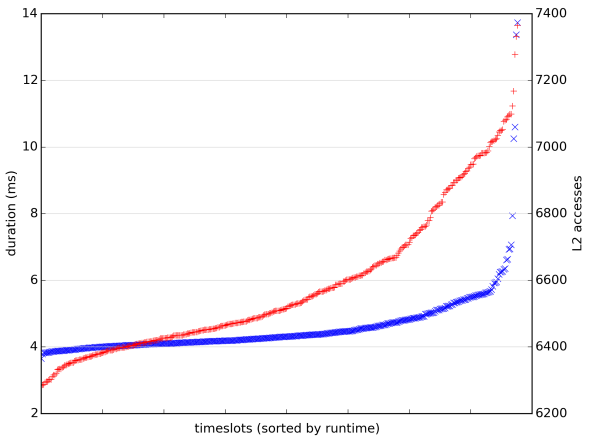

Fig. 9. Variation of the runtime (in blue x) and L2 accesses (in red +) while running the FMS application standalone

The next step, as defined in Figure 3, would have been to run the FMS application on the first A53 core, while running stressing benchmarks on the other A53 cores to build a figure looking like Figure 4. Such experiments led to the same issue as presented in the previous section, with the benchmarks failing to successively stress the architecture.

As a consequence, we failed to build such a figure with well identified asymtotes, and had to rely on an alternative technique:

Let $R = [R_0, R_1, \ldots, R_N]$ be the set of all the required access budget per timeslot for the critical FMS application running standalone without any activation of the asynchronous

tasks. We set $E = [E_0, E_1, \ldots, E_N]$ the set of per-timeslot extra supported budget such as $E_i = (max(R) - R_i) \times 20\%$.

Doing so is not representative of the total budget available on the target platform, but this way the supported extra budget is inversely proportional to the resource usage performed by the critical application, as it should really be.

C. RTE Evaluation

As a baseline for the evaluation of the runtime engine we started to deploy the FMS application on the first A53 core, and the BiQuad applications on the remaining A53 cores. Doing so, we observed a deadline miss ratio (number of timeslot with a deadline miss compared to total number of timeslots) of 24.7%.

We then run the same application deployment, this time supervised by the Budget Based Runtime Engine. As a budget, we used the previously computed budget E. And again we ran the FMS application without any asynchronous tasks. The deadline miss ratio decreased to 2.6% while the number of slots with suspended non-critical tasks increased to 30.9%, proving the ability of the Runtime Engine to suspend low-critical tasks when they are endangering the high-critical ones.

The high-critical application normally also encompass some asynchronous tasks that we ignored so far. However some of these sporadic task have a huge memory footprint, especially the tasks from the Trajectory task group that are performing random accesses to the 100MB large navigation database.

It would make no sense to have a preset static scenario, forcing in which timeslot each asynchronous task would be triggered. It would be similar to considering these tasks as periodic with a very large period. On the other hand a random scenario will raise a reproductability issue as we are gathering the timeslot statistics.

As a consequence, we created randomly two static scenarios for asynchronous task triggering, one was used during the characterization phases to compute a R' extra supported budget set, and the second one was used while running under the supervision of the runtime engine. In such a scenario we observed a final miss ratio of 13.0% and a suspend ratio of 52.7%.

One one hand, the increase in suspend ratio is due to the over-provisioning of the timeslots with some asynchronous tasks in E', leading to unnecessary suspends at runtime. One the other hand, the increased deadline miss ratio could be explained by the lack of such provisioning in the timeslots were the asynchronous tasks finally occur.

D. Limitation of the approach

Beyond the issues related to the characterization phases because of the hardware prefetcher preventing our dedicated benchmarks to sufficently stress the hardware resources, we identified a set of issues and limitations while evaluating the runtime engine.

First relying on Performance Monitor Counters restricts us to the hardware resources within the cores. As a consequence we were not able to evaluate shared hardware resources such as the DDR controller. With the prefetchers disabled, furthermore increasing the stride of the stressing benchmarks would have caused memory page reloads at the level of the memory controller, causing extra delays. These sources of interference have not yet been evaluated.

A more fundamental limitation is the issue caused by the aperiodic tasks. When pre-computing budgets with characterization phases, we face the same issue as with machine learning with both over-learning, and outlier issues, and as a result are facing either deadline misses or unnecessary suspends.

Also having per-timeslot budget is not practical for real applications running for hours. During our evaluation, the FMS ran for up to 5 minutes. With 200ms timeslots, that involves 1500 different budgets for this critical application. Also, because of unpredictable aperiodic tasks, it is not really possible to identify a shorter repeating patterns.

We performed test with an average timeslot budget, but doing so only reduces the number of deadline misses by 2.04%. Using a maximum timeslot budget on the other hand eliminates all the deadline misses but the low-critical tasks are systematically suspended.

VII. Conclusion and Future works

In this paper, we presented a regulation solution based on budgeting that aims at guaranteeing the temporal behavior of high-critical tasks in a mixed critical system, while degrading the behaviour of non critical tasks.

The budgeting approach is performed offline and per timeslot. If the approach worked well with purely periodic tasks, high-overhead aperiodic or sporadic tasks caused some major problems: either over-provisioning if considered during the offline characterization phase, causing very low budget for high-critical tasks and as a consequence a high rate of low-critical suspend rate; or causing high-critical deadline misses if occurring at runtime during an unprepared timeslot.

In single-core systems, it is common practice to have a dedicated periodic slot to deal with sporadic asynchronous tasks. The applicability of such a practice for multi-core architecture depends on the way the applications are deployed in the system to benefit from parallelism: an option is to parallelize inside applications / partitions, granting all the cores to a single application during each time slot. Such a deployment is compatible with the usual way of dealing with aperiodic tasks. But it also forces to re-write the applications, which performance will then be constrained by the Amdahl's law [2].

Another option is to run different independent applications, running partitions in parallel. If such a scheme is good for software development that could carry on producing single-thread applications, and even though it could bring better performance, exploiting the Gustafson's law [10] and not being hampered by data dependency. It does not allow anymore to have dedicated timeslots for a particular kind of traffic or tasks, unless introducing costly synchronization.

As a consequence, before selecting the most adequate control or regulation solution to deal with timing interference on multi-core, a first step should be to consider the possible

deployment of the applications, figuring out which kind of parallelism will be exploited. This choice has consequences on the timing interference level, on the visibility of interference (from white-box in case of intra-partition parallelism as they are coming from well known other tasks of the application, to black-box when coming from a potentially unknown independent application in case of inter-partition parallelism).

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. Versatile Express: 64 Bit Juno r2 ARM Development Platform, Nov 2015.

[4] 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.

[5] Jingyi Bin, Sylvain Girbal, Daniel Gracia Perez, Arnaud Grasset, and Alain Merigot. Studying co-running avionic real-time applications on multi-core cots architectures. *Embedded Real Time Software and Systems conference*, Feb 2014.

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

[7] 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.

[8] 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.

[9] 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 (under review)*, ERTS '18, 2018.

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

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

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

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

[14] 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.

[15] 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'14)*, pages 139–148, 2014.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.