Porting a safety-critical industrial application on a mixed-criticality enabled real-time operating system

Antonio Paolillo, Paul Rodriguez,
Vladimir Svoboda, Olivier Desenfans,
Joël Goossens, Ben Rodriguez

PARTS Research Centre
Université libre de Bruxelles, HIPPEROS S.A.

Sylvain Girbal,
Madeleine Faugère,
Philippe Bonnot

Thales Research & Technology

***Abstract*—This paper presents the practical implementation of a multi-core mixed-criticality scheduling algorithm. The goal of this work is to show the practical platform utilisation gain by allowing the concurrent execution of applications having different levels of criticality. We implemented the port of an existing industrial application provided by Thales Research & Technology on an embedded real-time operating system featuring task execution budget control, multi-core scheduling and multiple execution mode changes. We evaluated our solution by measuring the time that remains available for a low-criticality application running concurrently with the high-criticality use case mentioned above.**

***Index Terms*—Real-time, mixed-criticality, scheduling, Thales, system, RTOS, HIPPEROS.**

I. Introduction

Since the seminal paper by Vestal in 2007 [1], a large number of scientific papers related to mixed-criticality scheduling techniques have been published [2]–[4]. However, few of these techniques have been implemented or tried on practical problems [5]. At the same time, the idea of mixed-criticality in industrial applications is not new. Even the Apollo Guidance Computer developed in 1966 and embedded in the Saturn V rocket had a primitive form of criticality management which took effect during the moon landing stage [6].

In tightly certified industries, isolating software components of varying criticality results in important economical gains and shorter time to market. Indeed, software development at high levels of certification is extremely costly. In current systems, some tasks are certified at a higher degree of safety than what is strictly necessary due to sharing hardware with more critical tasks. Mixed-criticality techniques may contribute to partially solve this problem by ensuring strong temporal isolation between tasks even when running on the same hardware.

The IMICRASAR [7] project targeted the creation of a version of the HIPPEROS Real-Time Operating System (RTOS) [8] under two main requirements. First, porting the OS to a PowerPC multi-core platform brought by Thales Research & Technology. Second, to operate with a multi-mode isolated mixed-criticality scheduling policy in order to provide predictable hard real-time guarantees.

The scope of the project included those two items, as well as experiments validating the proposed software solution with the Thales industrial use case. In this paper, we present the system architecture providing isolation between the different criticality applications and the experiments validating our solution.

Our contributions in this paper are as follows. We show there is evidence that OS-level mixed-criticality scheduling allows using computing platforms more efficiently not only in theory but also in real industrial scenarios. Our results show that significant shares of the computing power of the platform are made available to non-critical tasks while conserving the safety guarantees offered to high criticality tasks.

II. The HIPPEROS RTOS

The HIPPEROS multi-core kernel architecture follows a master-slave asymmetric model [8]. User tasks are executed on slave cores and potentially on the master core. As opposed to symmetric kernel designs, the master core has exclusive access to some of the critical pieces of the kernel data structures.

In particular, the master core is the central authority in terms of scheduling. Whenever an event related to scheduling happens (such as a job release of a periodic task), the master core will update its scheduling model and send context switch requests to the slave cores accordingly and independently from one-another. This design has its pros and cons, one clear advantage is that it greatly limits contention on locks by avoiding direct data dependencies between slave cores [8]. By concentrating most of the scheduling-related interrupt handling on the master core, this design also keeps scheduling overheads on slave cores to a minimum. Finally, as the scheduling data must not be shared across cores of the platform, it can be contained in the master core cache, allowing faster scheduling operations and minimising the OS impact on applications. The master core has the necessary structures to handle asynchronous system calls without direct mutual exclusion between slave cores.

HIPPEROS is based on a micro-kernel design. This means that only the most basic system operations run in kernel mode (scheduler, memory paging, etc.) while most of the drivers and high-level system operations (file system, logging, etc.) are pushed in RTOS services running in user mode next to the application tasks. To allow applications to communicate

with these service tasks, the kernel exposes a multi-core Inter-Process Communication (IPC) API based on both message passing and shared memory. For example, in the application use case presented in this paper, the user tasks use the logging service in order to send display information to an external application. Therefore, multi-core IPC mechanisms are extensively used in the execution of the experiments of this paper. The logging service uses a UART driver to use the serial device of the platform.

HIPPEROS allows the execution of real-time tasks with time budget allocation and deadline control. The specific time-related RTOS mechanisms used in this work are described in Sections IV and V.

III. The Thales use case

This research is largely based on an industrial use case provided by Thales Research & Technology [9]. The use case was an application written in C++ and structured into multiple recurrent communicating tasks. The use case itself was largely comprised of tasks that would be certified at one assurance level. However, the aim of this research was in part to show that given an existing industrial application, applying mixed-criticality techniques could lead to additional features at little cost.

The task set consisted in essentially two types of tasks. First, many sensor-like applications retrieving exterior information about the system it is supposed to be embedded in. Second, a few heavier tasks doing database operations and distance calculations. As can be seen in Section VI on experiments, the tasks that handled distance calculations made heavy use of the Floating Point Unit (FPU) of the platform. The first kind of tasks mainly fed the second kind with information, sometimes through pipelines with multiple stages.

The coherency of communications between tasks was handled by time-slicing which required support at the OS level for HIPPEROS. We implemented time-slicing by offsetting the periodic releases of the tasks.

IV. Mixed-Criticality Model

This paper covers the design of the mixed-criticality solution that was implemented in the HIPPEROS kernel within the IMICRASAR project. These changes affected the existing OS in specific, well-defined components. Namely, the scheduler, the task description system, the system call layer and the event handler. The implemented mechanisms are known in the literature as elastic mixed-criticality scheduling [10]. In relation to the state of the art, this section goes into the details of how the multi-WCET and related techniques for mixed-criticality scheduling were designed.

A. Task types

The system supports three types of tasks: highly critical tasks ("HI tasks"), real-time tasks ("LO tasks") and best effort tasks. Best effort tasks are free to be released at any rate and to consume an arbitrary amount of computation time. However, they will be considered with the lowest priorities by the scheduler and may therefore not get the resources needed to perform their purpose within strict time bounds. LO tasks have design-time defined periods (or inter-arrival times), deadlines and worst-case execution times. The scheduler will preempt best-effort tasks to execute real-time tasks to completion. Knowing bounds on the worst-case execution times ("WCET") of the tasks allows the operating system to allocate a time budget to each real-time task. Real-time tasks must be divided into two sub-categories: compressible and incompressible. Compressible LO tasks allow the OS to reduce their arrival rate (increase the periods) within predefined limits, therefore allowing some form of Quality of Service (QoS) adjustment. Incompressible real-time tasks simply do not allow such period changes. Finally, highly critical tasks have the characteristics of incompressible real-time tasks but additionally provide an intermediate execution time limit ("LO WCET") between zero and their WCET. The use of this intermediate execution time limit is described in Section IV-B on mode switching.

Criticality and priority are different concepts. To guarantee the deadlines of regular real-time tasks, some highly critical tasks may have lower priority than some regular real-time tasks. In this sense, priority is merely a tool to ensure that the proper scheduling decisions are taken to meet all deadlines. Criticality is a degree of assurance that temporal (and other) constraints will be met for a given task. Tasks that have higher criticality undergo stricter development and verification techniques including more pessimistic WCET bound evaluation.

B. Mode switching

Two modes of operation are defined. Under normal circumstances, the system uses all resources as efficiently as possible. Best effort tasks are allowed to run with no constraints (other than their low priorities) and compressible real-time tasks are released at the nominal rate. Whenever a highly critical task overruns its LO WCET, the system enters a critical mode of execution. In critical mode, best effort tasks are suspended and compressible real-time tasks have longer periods. Some compressible tasks may have application-defined maximum delays between job releases such as a lower bound on the update frequency of a sensor. To accommodate for such tasks, the task may be allowed to run to completion when a switch to critical mode occurs. This design achieves the double goal of providing highly critical tasks with the assurance that they will meet their deadlines even in the event of extremely long execution times and allowing other tasks to use the otherwise wasted resources when execution times are closer to the average. In most cases, the LO WCET would be set such that the production environment never switches to critical mode. The mode switch is therefore present to ensure a recovery plan in the rare case of the HI task overrunning its LO WCET.

From critical mode, the system goes back to normal mode when there are no available jobs (i.e. at the first point in time when the system is idle).

Note that any task overrunning its WCET will be instantly killed by the kernel. This holds for single- or mixed-criticality system execution.

C. Mixed-criticality scheduling algorithms

In the literature, mixed-criticality task systems are often considered in tandem with variations on either fixed priority scheduling or deadline-driven (EDF) scheduling [3]. Indeed, the mode switch mechanism described above alone does not guarantee anything, as it might already be too late to execute a highly critical task to completion before its deadline when a mode switch occurs. Mixed-criticality scheduling techniques explicitly make reservations to be able to handle the worst case scenario.

However, we did not implement these techniques in the context of this work. Only the mode-switching system required by such scheduling algorithms has been implemented, and it is applied in our experiments with a simple partitioned static priority scheduler. The HIPPEROS implementation of mode-switching is scheduler-agnostic, which means it can be used with any supported scheduler without special configuration. The static priority scheduler gives the CPU to user processes according to user-defined task priorities. We used priorities and partitioning guaranteeing the correct behaviour of the Thales use case (i.e. not Rate Monotonic or any other standard priority attribution policy).

V. Mixed-Criticality Considerations

Mixed-Criticality applications are primarily concerned with the isolation of tasks. A vast number of hardware and software components in a real-time system are shared between tasks and can potentially lead to less critical tasks interfering with more critical tasks. In a RTOS, ensuring mixed-criticality operation means that the OS ensures as much temporal isolation between tasks as possible across criticality levels. In other words, while temporal isolation between any two tasks is already an important feature of RTOS design, additional steps must be taken to ensure that less critical tasks cannot cause more critical tasks to fail. Of course, there may be interference caused by hardware constraints or application behaviour that the OS cannot eliminate or mitigate and that cannot be accurately accounted for.

As such, the challenges resulting from the use of modern general purpose platforms in the context of mixed-criticality real-time systems must be faced using an array of techniques ranging from application to hardware design, including RTOS scheduling mechanisms. The design goal of the HIPPEROS mixed-criticality subsystem is to gather a set of OS level solutions aimed at temporal isolation and efficient use of resources in mixed-criticality applications. However, the challenges posed by temporal isolation are much more difficult to tackle at the software level and it is also usually difficult to evaluate the impact of isolation techniques on a real platform.

Theoretical works have been produced on techniques aimed at mitigating the impact of highly critical tasks upon platform utilisation. The theory derives from the observation that the strict static techniques used to evaluate the WCET of critical tasks give results that are usually considerably higher than their actual WCET bound. In particular, there is a difference between the WCET evaluation methods used for critical tasks and non-critical (but still real-time) tasks [1]. Additionally, the real probability distribution of execution times of a complex program on most modern embedded systems does naturally tend to have a long tail of extremely high and extremely unlikely execution times. This statistical tendency leads to excessively pessimistic WCET upper bounds.

In practice, a real-time system can implement multiple modes of execution and switch between them according to the status of a critical task after some execution time threshold has been met. Under normal circumstances, the system is utilised efficiently, allowing low criticality tasks to use the available computing power. In the event that a critical task has not completed before a given execution time threshold, the system enters another mode where low criticality tasks receive less computing power. Naturally, such techniques only make sense if critical and non-critical tasks are mixed on the same core. Analysis techniques exist to keep the loss of computing power of non-critical tasks to a minimum [11]. In the context of the IMICRASAR project, we choose to implement this multi-mode scheme in the HIPPEROS kernel. Another solution would be to use virtualisation. An hypervisor would execute two operating systems simultaneously, one for each criticality level. Therefore, it would be the role of the hypervisor to ensure the temporal isolation between the tasks of different criticality levels. However, this solution has already been tested thoroughly in the literature [12] and would suffer from heavier system overheads than the multi-mode RTOS solution.

VI. Experiments

A. Hardware platforms

The experiments have been executed on two different embedded platforms: on a NXP T2080RDB and on a Boundary Devices SABRE Lite.

The T2080RDB is a development board which is part of the NXP QorIQ PowerPC64 family. The processor is made of 4 dual-threaded cores. We only used one of the two threads per core in the experiments. The processor supports the partitioning of the L2 caches. Both the L1 and L2 caches were activated for the experiments and the L2 cache was partitioned: each core having its own set of exclusive cache ways. On this processor, the Translation Look-aside Buffer entries must be loaded by software through a TLB miss exception handler. The application and the kernel are small enough so that all the TLB entries can be loaded concurrently at initialisation.

The SABRE Lite is a development board which contains a quad-core ARM Cortex-A9 processor. Only the L1 cache was activated for the experiments. The MMU (Memory Management Unit) was activated and the TLB entries are automatically loaded by the core when needed.

B. Experimental design

We designed a set of experiments with the aim of evaluating the trade-off between different choices of LO WCET in HI tasks. Intuitively, it is expected that choosing high values for the LO WCET of highly critical tasks (i.e., values close to their actual high criticality WCET) will result in fewer

Fig. 1. Maximum measured execution times of the main recurring tasks of the Thales application on the SABRE Lite.

Fig. 2. Maximum measured execution times of the main recurring tasks of the Thales application on the T2080RDB.

criticality mode switches and therefore more total execution time available for low criticality tasks. It is also expected that using such high values will result in shrinking schedulability windows for low criticality tasks. In particular, setting the low criticality execution time limit to the WCET of each high criticality task turns the mixed-criticality system into a single-criticality system (i.e., a system without any possible mode switch). To ensure schedulability, low criticality tasks must be tested while taking the low criticality execution time limit of high criticality tasks into account.

In essence, the system designer is faced with a balancing act that consists in finding LO WCETs that are simultaneously low enough to really give more breathing room to low criticality real-time tasks and high enough to avert using too many criticality mode switches (or completely avoid them). An excessive amount of mode switches, while irrelevant to high criticality tasks, will defeat the point of putting more low criticality workloads on the system.

In every graph below, the system with the Thales application was run for the time of the mission critical application (approximately 92 seconds). This length of time corresponds to a standard demonstration run of the application and a representative slice of what it is supposed to do. Available CPU time is expressed in seconds over the whole run.

C. The task set

We ran the Thales use case application as a single criticality task set without any interference to measure the execution times of its tasks on both boards. The results for SABRE Lite are on Figure 1 and the results for T2080RDB are on Figure 2, taking the maximum observed job execution time over 10 executions of the application for each data point. We then used the complete measurements (negligible tasks have been stripped out of the charts to make the figures legible) to arbitrarily choose safe WCET bounds that would typically be found using static WCET evaluation techniques. We set the WCET of each task to approximately 4 times the time of the longest measured job of this task, over 10 measurements. This low amount of repetitions was sufficiently accurate considering the stability of the results using simple sampling techniques. Obviously, this choice of WCET influences our other results.

As can be noted from these figures, the SABRE Lite and the T2080RDB have different execution time profiles. Not only is the T2080RDB faster in general, there is also a much greater difference between tasks that make extensive use of the FPU (“near_p1” and “traj_r1” in the figures) and tasks that don’t compared to the SABRE Lite. Running the same application using one or all cores of the platform also affected the execution times. Note that none of the tasks were multi-threaded, they were simply partitioned on multiple cores instead of one. Multi-core execution times were shorter than mono-core execution times across the board, which suggests that the caches of both platforms were better used in multi-core, although that is merely our hypothesis. This is one of many reasons why deriving WCET bounds is complicated and prone to extreme pessimism.

D. Effect of the LO WCET

We ran the Thales use case application on the SABRE Lite and the T2080RDB. We wanted to evaluate the impact of mode switches on potential low criticality tasks to be added to the set of high criticality tasks in the original application. So we added low criticality dummy tasks (one per core) using the CPU exclusively (i.e. not performing I/O of any kind). These dummy tasks effectively act as coal mine canaries as they are set to be suspended when a mode switch occurs. At the end of the run, the CPU time of each job execution of every task is collected. Obviously, all runs of the experiments were checked to guarantee that none of the Thales use case tasks missed a deadline (i.e., all the HI tasks). This ensures that the HI tasks did not suffer from interference from the LO tasks in the experiments. Notice that we only used incompressible LO

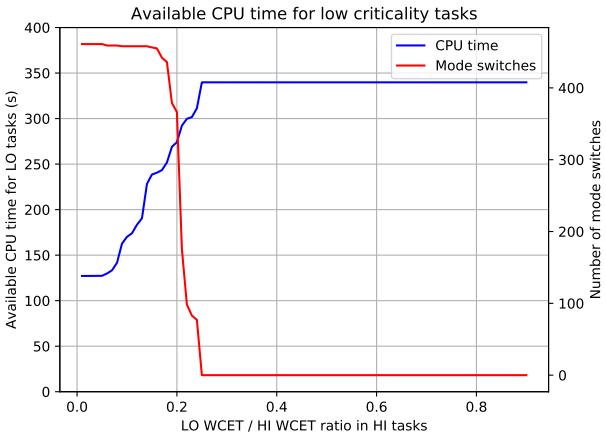

Fig. 3. Evolution of the available low criticality CPU time and number of mode switches over full runs of the Thales application when varying the LO WCET of HI tasks. Experiment on the SABRE Lite board.

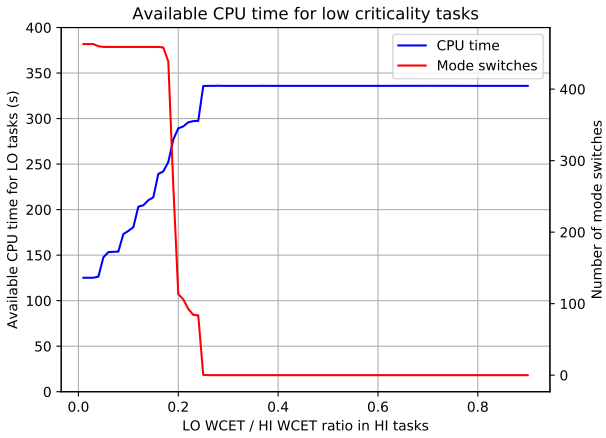

Fig. 5. Evolution of the available low criticality CPU time and number of mode switches over full runs of the Thales application when varying the LO WCET of HI tasks. Experiment on the T2080RDB.

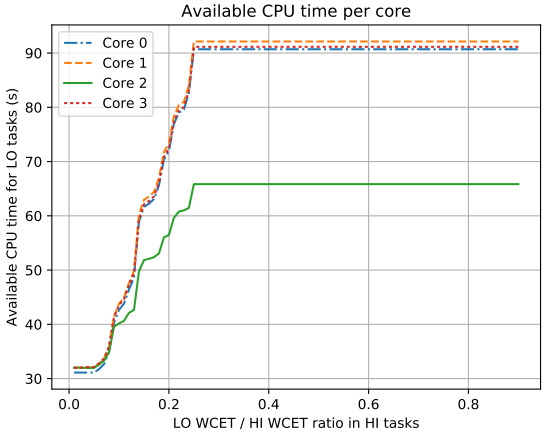

Fig. 4. Per-core available CPU time for low criticality tasks on the SABRE Lite. The value on core 2 is lower than others because the task that interacts with the host computer is very demanding and is partitioned on core 2.

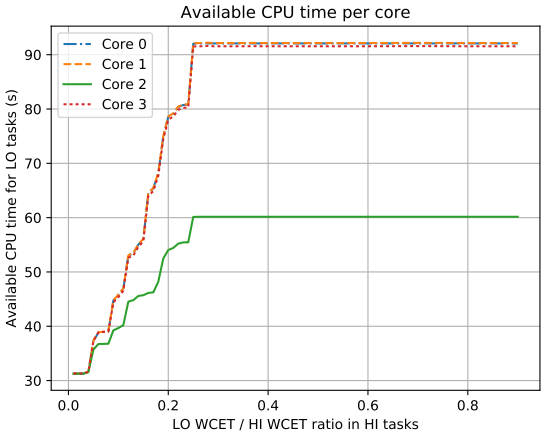

Fig. 6. Per-core available CPU time for low criticality tasks on the T2080RDB. The value on core 2 is lower than others because the task that interacts with the host computer is very demanding and is partitioned on core 2.

tasks for the experiments. Adding the total execution time of all low criticality dummy tasks, we obtained the available CPU time for low criticality tasks running alongside the Thales use case. The results of the multi-core experiments on the SABRE Lite board are shown in Figure 3, and in Figure 5 for the T2080RDB.

As can be seen on both graphs, the use of pessimistic WCET evaluation techniques leads to large potential gains through mixed-criticality mode switching. Indeed, setting the intermediate execution time limits of high criticality tasks (LO WCET) to a relatively small fraction (around 30%) of their WCET bounds generates no mode switches in our experiments. This suggests that the probability of a mode switch occurring with these values is very low. Because no mode switches occur, low criticality tasks are never suspended by the mixed-criticality mechanism, and are thus free to use the platform up to capacity. This is visible following the evolution of the CPU time curve. Notice that as the dummy tasks fill the remaining available time on the platform, these measurements allowed us to derive that the Thales use case is taking less than 10% utilisation on both platforms.

As the LO WCET is set to lower values, criticality mode switches start to occur and the CPU time available for low criticality tasks declines rapidly. As the pessimistic WCET bounds are approximately 4 times the maximum observed execution time (see Figures 1 and 2) and that the observed execution times are very stable, it is not surprising that mode switches start to occur only below 30%. The number of mode switches changes very rapidly when LO WCET varies between 30% and 15% on both boards, but stays roughly stable outside of that range. This suggests that around 15% and lower is the range of values at which almost all possible mode changes occur systematically. Available low criticality

CPU time decreases as soon as the first mode changes happen, however the decrease is much smoother. It is also worth noting that low criticality CPU time never drops below around 120 seconds. This suggests that even when all high criticality tasks overrun their LO WCET, there is still time in-between them where CPU time can be used by low criticality tasks. This is a sort of accidental time-slicing. At 0% LO WCET, low and high criticality tasks never execute simultaneously.

Below 15%, CPU time continues to decrease while the number of mode switches stays stable. This means that these mode switches arrive earlier in the system execution and therefore the system stays in critical mode for a longer time.

E. Per-core utilisation

We measured the CPU time available to LO tasks on each core separately on both boards. The results are shown on Figure 4 and Figure 6 for the SABRE Lite and the T2080RDB respectively. As can be seen on the graphs, the Thales use case application only uses a small fraction of the computing power available on both boards. Only core 2 is used significantly, due to the heavy task that handles communication with the host computer being partitioned on core 2. This is due to the usage of relatively slow device for the application to communicate with the external world (the UART/serial port communication). Note that this task is not part of the Thales use case, we added it ourselves for the needs of our experiments.

VII. Conclusions

In this research, we ported an existing industrial use case application from Thales Research & Technology to the HIPPEROS RTOS. We developed the mixed-criticality extension to the existing HIPPEROS kernel adding support for the elastic mixed-criticality model. We ported an industrial use case on two different boards based on different architectures, the SABRE Lite and the T2080RDB. We used the mixed-criticality system of HIPPEROS and actual executions of the Thales use case to show how much CPU time is available on the platform for low criticality tasks.

Our data illustrates that the pessimistic WCET bound evaluation techniques applied on highly critical tasks can be mitigated efficiently. A significant load of low criticality tasks can be added at little cost and no OS-level risk to the original application. The OS does not guarantee that there is no cache interference between low criticality and high criticality tasks, but it does guarantee that low criticality tasks are suspended when high criticality tasks execute for longer than their typical duration, as chosen by the user.

Note that we only measured the total available low criticality CPU time and number of criticality switches. Adding real-time guarantees to low criticality tasks requires low LO WCETs for high criticality tasks. Therefore, there is a true trade-off between choosing high to minimise the risk of mode switches and choosing low to maximise the time to be allocated to low criticality tasks. We expect the optimal trade-off to be found by setting the LO WCET to the smallest value such that no mode switch occurs within acceptable probabilities (domain-defined).

For future work, we consider the implementation of actual mixed-criticality scheduling algorithms (such as EDF-VD [13]). It is the logical next step in the development of the HIPPEROS mixed-criticality solution. On the experiment side, we also consider implementing non-trivial low-criticality applications (with memory accesses, file systems and other I/O operations) that are both compressible and incompressible. This would allow to observe the mixed-criticality system behaviour in the presence of more shared hardware resources, making the experiments more representative of typical industrial applications.

Acknowledgements This work was supported by EU funding as part of the IMICRASAR Industrial Experiment under the umbrella of the EuroCPS H2020-ICT-2014-1 project in partnership with Thales Research & Technology.

References

[1] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*. IEEE, 2007, pp. 239–243.

[2] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*. IEEE, 2010, pp. 13–22.

[3] S. K. Baruah, A. Burns, and R. I. Davis, "Response-time analysis for mixed criticality systems," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*. IEEE, 2011, pp. 34–43.

[4] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, 2013.

[5] M. Lemerre, E. Ohayon, D. Chabrol, M. Jan, and M.-B. Jacques, "Method and tools for mixed-criticality real-time applications within pharos," in *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on*. IEEE, 2011, pp. 41–48.

[6] D. Eyles, "Tales from the lunar module guidance computer," in *27th annual Guidance and Control Conference*. American Astronautical Society, 2004.

[7] "The imicrasar eurocps project," accessed 25-September-2017. [Online]. Available: https://www.eurocps.org/innovators-projects/ongoing-projects/imicrasar-isolated-mixed-criticality-avionics-system-architecture/

[8] A. Paolillo, O. Desenfans, V. Svoboda, J. Goossens, and B. Rodriguez, "A new configurable and parallel embedded real-time micro-kernel for multi-core platforms," in *ECRTS-OSPERT*, July 2015.

[9] G. Durrieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch, "Predictable Flight Management System Implementation on a Multicore Processor," in *Embedded Real Time Software (ERTS'14)*, Toulouse, France, Feb. 2014. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01121700

[10] H. Su and D. Zhu, "An elastic mixed-criticality task model and its scheduling algorithm," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2013, pp. 147–152.

[11] F. Santy, L. George, P. Thierry, and J. Goossens, "Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE, 2012, pp. 155–165.

[12] A. Crespo, A. Soriano, P. Balbastre, J. Coronel, D. Gracia, and P. Bonnot, "Hypervisor feedback control of mixed critical systems: the xtratum approach," in *ECRTS-OSPERT*, June 2017.

[13] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie, "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE, 2012, pp. 145–154.