Proceedings of the 5th Workshop on Mixed-Criticality Systems (WMC)

held in conjunction with the Real-Time Systems Symposium (RTSS)

Edited by Kunal Agrawal[1] and Arvind Easwaran[2]
[1]*Washington University in St. Louis, USA*
[2]*Nanyang Technological University, Singapore*

Paris, France
December 5, 2017

Message from the Chairs

It is our pleasure to welcome you to the 5^{th} International Workshop on Mixed-Criticality Systems (WMC) at the Real-Time Systems Symposium (RTSS) in Paris, France, on 5^{th} December 2017.

The purpose of WMC is to promote sharing of new ideas, experiences and information about research and development of mixed criticality real-time systems. The workshop aims to bring together researchers working in fields relating to real-time systems with a focus on the challenges brought about by the integration of mixed criticality applications onto singlecore, multicore and manycore architectures. These challenges are cross-cutting. To advance rapidly, closer interaction is needed between the sub-communities involved in real-time scheduling, real-time operating systems / runtime environments, and timing analysis. The workshop aims to promote understanding of the fundamental problems that affect Mixed Criticality Systems (MCS) at all levels in the software/hardware stack and crucially the interfaces between them.

For this fifth edition of the workshop a total of 7 submissions were received. In the review process, each submission received 4 reviews. We decided to accept 5 papers for presentation at the workshop. We sincerely thank all the Program Committee members for their time and effort in the review process.

As well as regular paper presentations, there is a keynote session on "Challenges in Applying Mixed-Criticality Systems to Aircraft Engine Control Systems", jointly given by Iain Bate (University of York, UK) and Stephen Law (Rolls-Royce Control Systems, UK). In addition, there is a special invited session comprising two talks: 1) Alan Burns (University of York, UK) will give a talk titled "Defining and Delivering Resilience in Mixed-Criticality Systems", providing a brief overview of MCS research over the past decade and introducing some new research directions currently being explored; 2) Borislav Nikolic (TU Braunschweig, Germany) will give a talk titled "Mixed Criticality Systems - A History of Misconceptions?", presenting some research challenges from the perspective of industrial MCS.

WMC 2017 would not be possible without the hard work of a number of people involved in the organization of RTSS 2017, including Frank Mueller, Isabelle Puaut, Liliana Cucu-Grosjean and Linda Buss. In particular, we would like to thank the RTSS 2017 workshops chair Song Han for his excellent organization of the overall workshops program. We also thank the WMC Steering Committee for their guidance.

Finally, we would like to thank all of the authors who submitted their work to WMC 2017; without them this workshop would not be possible. We wish you an interesting and exciting workshop and an enjoyable stay in Paris.

Kunal Agrawal (Washington University in St. Louis, St. Louis MO, USA)
Arvind Easwaran (Nanyang Technological University, Singapore)
WMC 2017 Program Chairs

WMC 2017 Technical Program Committee

Dirk Ziegenbein, Bosch Research, Germany

Zhishan Guo, Missouri University of Science and Technology, USA

Luca Santinelli, Onera, France

Jing Li, New Jersey Institute of Technology, USA

Leandro Soares Indrusiak, University of York, UK

Geoffrey Nelissen, ISEP, Portugal

Martina Maggio, Lund University, Sweden

Jaewoo Lee, KAIST, Korea

Xiaoting Li, ECE Paris, France

Tam Chantem, Virginia Tech, USA

Pontus Ekberg, Uppsala University, Sweden

WMC Steering Committee

Robert Davis, University of York, UK

Liliana Cucu-Grosjean, INRIA-Paris, France

Sanjoy Baruah, Washington University in St. Louis, USA

Claire Maiza, Verimag, France

Technical Program

Porting a Safety-Critical Industrial Application on a Mixed-Criticality Enabled Real-Time Operating System
Antonio Paolillo, Paul Rodriguez, Vladimir Svoboda, Olivier Desenfans, Joel Goossens, Ben Rodriguez, Sylvain Girbal, Madeleine Faugère and Philippe Bonnot 1

Selective Real-Time Data Emission in Mobile Intelligent Transport Systems
Laurent George, Damien Masson and Vincent Nelis 7

Response Time Analysis for Mixed Criticality Systems with Arbitrary Deadlines
Alan Burns and Robert Davis 13

Probabilistic Analysis of Low-Criticality Execution
Martin Küttler, Michael Roitzsch, Claude-Joachim Hamann and Marcus Völp 19

Revisiting the Computational Complexity of Mixed-Critical Scheduling
Rany Kahil, Peter Poplavko, Dario Socci and Saddek Bensalem 25

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

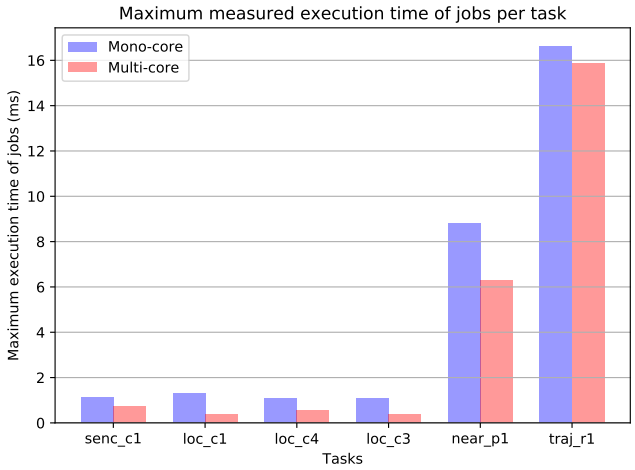

Fig. 1. Maximum measured execution times of the main recurring tasks of the Thales application on the SABRE Lite.

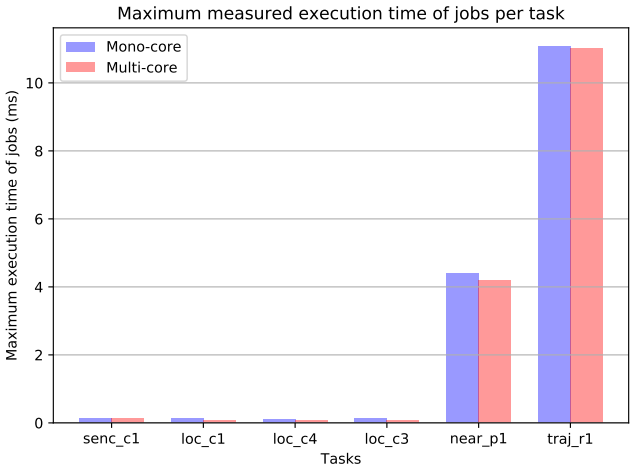

Fig. 2. Maximum measured execution times of the main recurring tasks of the Thales application on the T2080RDB.

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

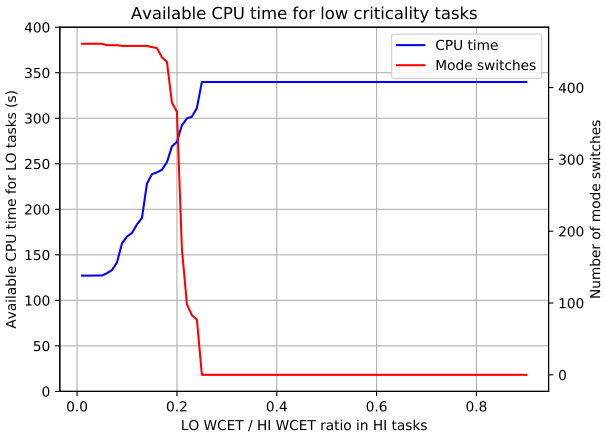

Fig. 3. Evolution of the available low criticality CPU time and number of mode switches over full runs of the Thales application when varying the LO WCET of HI tasks. Experiment on the SABRE Lite board.

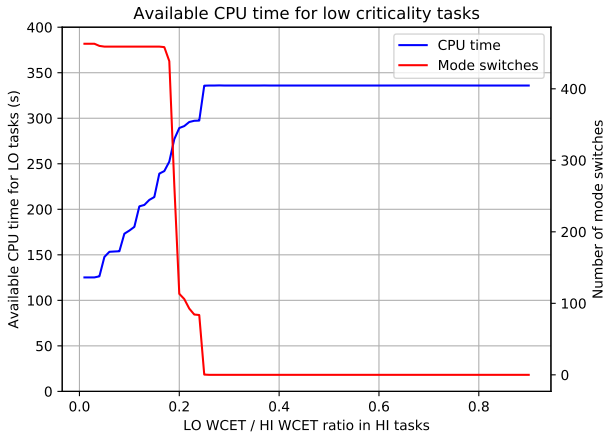

Fig. 5. Evolution of the available low criticality CPU time and number of mode switches over full runs of the Thales application when varying the LO WCET of HI tasks. Experiment on the T2080RDB.

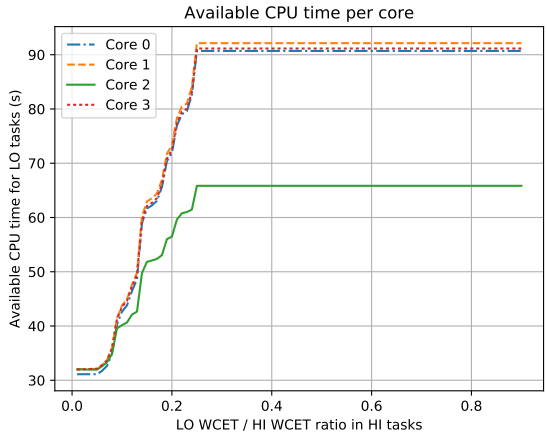

Fig. 4. Per-core available CPU time for low criticality tasks on the SABRE Lite. The value on core 2 is lower than others because the task that interacts with the host computer is very demanding and is partitioned on core 2.

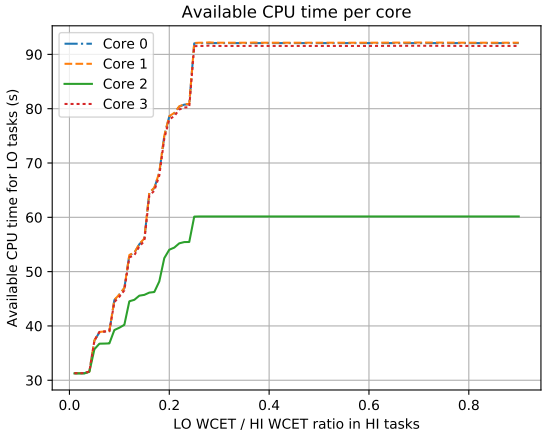

Fig. 6. Per-core available CPU time for low criticality tasks on the T2080RDB. The value on core 2 is lower than others because the task that interacts with the host computer is very demanding and is partitioned on core 2.

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.

Selective Real-Time Data Emission in Mobile Intelligent Transport Systems

Laurent George and Damien Masson
Université Paris-Est, LIGM (UMR 8049), CNRS, ENPC, ESIEE Paris, UPEM, F-93162, Noisy-le-Grand, France
Email: laurent.george@esiee.fr

Vincent Nelis
CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Portugal
Email: nelis@isep.ipp.pt

***Abstract*—Intelligent Transport Systems (ITS) gather information from their fleet vehicles to improve on their safety and quality of service. Information is sent through a wireless connection (e.g., 3G link) to a central unit responsible for controlling the activity of the vehicles. These data may include for instance the location of the vehicles, their speed, the time spent at each stop, an estimated number of passengers, the state of the doors (opened/closed), etc. In locations where the quality of the network is poor (low data rate), vehicles do not succeed in transmitting *all* the collected information and mechanisms must be implemented to select which information the network can afford to send. Assuming that the data has been labeled and classified beforehand according to its criticality, this paper identifies the exact minimum network speed to successfully send each (set of) data, defining a set of speed thresholds for each level of criticality. We assume that messages are sent according to a non-preemptive fixed priority scheduling. Based on the results of this paper, we expect to later develop algorithms to optimally send the data for arbitrary priority assignments and maximize the utility when choosing which data is discarded due to insufficient network bandwidth.**

1. Introduction

In Intelligent Transport Systems (ITS) [1] for public transports, vehicles like buses, tramways and trains are equipped with tracking devices that periodically send operational information to a central server. The server processes this information and sends feedback to the fleet of vehicles to eventually improve the security, safety, or the quality of the service in general. The tracking devices installed in each vehicle record, sample, and send a broad set of information such as the location of the vehicle (GPS coordinates), the time spent at each stop and station, the time between every two stops, the state of the doors (open/closed), the estimated number of passengers, the temperature in each car (in case of a train for example), fire-detectors information, consumption information, density of the traffic, watchdogs information or even the driver heartbeat in some cases. Note that the amount of data to be sent can be large if images must be forwarded to the central unit (from the in-vehicle security cameras or from the advanced driver-assistance systems for instance). The data is categorized according to its importance or criticality to the project objectives. For example, GPS coordinates can be seen as more important than an estimation of the air quality in the cars. Information is thus clustered and broken into categories and each category is labeled by a measure of its criticality.

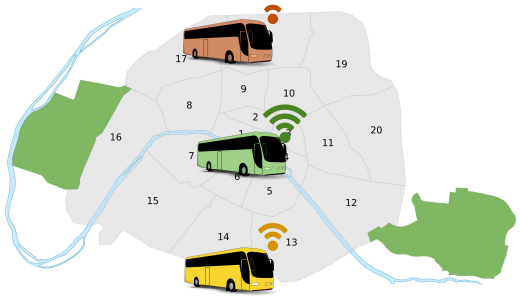

Figure 1. Example of network coverage in a city.

Every vehicle in the fleet sends its data periodically to the base central server over the city's 3G/4G network. However, many cities do not yet provide a full network coverage, and even when they do, there are always regions in which the signal strength drops and becomes insufficient to provide the network bandwidth required to send all the data at the desired frequency, as illustrated by Figure 1. In such regions, the system embedded in the vehicles has three options:

1) It can lower the emission rate by decreasing the frequency at which some of the information is sent. Note that, depending on the application constraints, this is not always feasible as some features are strongly time-dependent and become meaningless if the data is not processed at a specific rate, e.g. fire-detectors information or watchdogs information.
2) It can reduce the amount of data sent to match the capacity of the network that is currently available. That is, when the signal is too weak, the less critical data are no longer sent, which grants more bandwidth to the data that cannot alter their emission rate.
3) It can combine the first two approaches. That is, it can stop sending some of the (less critical) information *and* lowers the emission rate of some others.

The third option is most likely the most efficient technique as it combines the strengths of the first two options.

In this work though, we focus only on the second option in which the system temporarily stops sending data that are less critical to save bandwidth for the more critical ones. This difference of criticality between the data justifies the use of a fixed-priority scheduler for sending out the information. Due to the unpredictable nature of signal strength that can strongly affect the transmission time of the messages (up to the point where the transmission fails), it is arguably more appropriate to give a higher priority to the information of high criticality so that they are always sent before the less critical information. The transmission of data being a non-preemptible operation, it may happen that a data of low criticality is being sent while an information of high criticality becomes "ready-to-be-sent". This is the only case of priority inversion where the transmission of the higher critical data gets blocked by a lower critical data and this scenario can be easily factored in the schedulability analysis for non-preemptive fixed-priority arbitration.

The remainder of the paper is organized as follow. We present the model and notations in Section 2. Then the problem is formalized in Section 3. Section 4 recapitulates basics from the state of the art required to present our solution in Section 5 and to follow the proof of its correctness given in Section 6. Section 7 review existing related works and we conclude the paper in Section 8 with a discussion on future work directions.

2. Model of computation

The application is composed of a set τ of n periodic tasks denoted $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Every task models the process of periodically sending one information from the bus to the central station, such as the location of the vehicle (GPS coordinates) or the time spent at the last stop. Every task τ_i is activated periodically every T_i time units. We call each activation of a task a "job" and we say that the task "releases" a job when it gets activated. We denote the jobs by $\tau_{i,j}$, where $j \in \mathbb{N}_0$ is the job index. The time of the first activation of each task is not known. That is, once the first job $\tau_{i,0}$ is released (at any time $t > 0$), its subsequent jobs $\tau_{i,j}$, $\forall j > 0$, are released exactly T_i time units apart, starting from time t, i.e. $\tau_{i,j}$ is released at time $t + jT_i$, $\forall j$.

Every job of a same task sends a unique message of fixed length over the network. The transmission time of each message depends on the network's bandwidth available at the time of sending the message and the strength of the signal varies with the location of the vehicle. In some neighborhoods, the poor network coverage results in low-speed transmissions whereas at some other locations, the signal is strong and offers high-speed flawless communications. We denote by C_i the length of every message (in Megabits) to be delivered by the jobs of τ_i. At a given reference network speed of 1 Mbit/s (Megabit per second), it takes exactly C_i seconds to deliver each message of τ_i. From there, it trivially holds that at any speed of x Mbit/s, it takes exactly $\frac{C_i}{x}$ seconds to transfer each message of τ_i. The transmission time of every job $\tau_{i,j}$ of τ_i is thus linear in the network speed. A poor signal strength yields a long transmission time and inversely, a strong signal allows for faster transmissions.

It is important to note here that what we call *bandwidth* is the available bandwidth for the application, taking into account all optimizations made by the network protocols, the congestion on the network and so on. It is not in the scope of the paper to discuss how the available bandwidth is estimated, however, amongst other solutions, one could imagine that an acknowledgement is received after each transmissions to the central station, and that the bandwidth is computed from the measured delay to receive the acknowledgement.

Every task τ_i is assigned a relative temporal deadline denoted by D_i (relative to its activation time), with the interpretation that each of its jobs $\tau_{i,j}$ must have sent its message entirely before or at its absolute deadline set D_i time units after its release. We consider the constrained-deadline task model in which $D_i \leq T_i, \forall i$, meaning that every job must have entirely sent its message before the next job (from the same task) is released. Every τ_i is also assigned a *criticality* χ_i that qualifies the "importance" of the information that it sends to the central station. For example, χ_1 could represent *very important* messages, while χ_2 could be used for *moderately important* messages, χ_3 for *non crucial* information, etc. For convenience, we denote by $\tau(k)$ the set of tasks of criticality χ_k and we assume that $k \in [1, x]$, where x is the number of different criticality in the system (criticality 1 is the highest criticality level, then 2, 3, and so on till x, the least critical one).

All the messages are transmitted sequentially over the network in a non-preemptive way. This means that once a job has started to send its message, it cannot be interrupted or paused and continues emitting until its message is entirely transmitted. Since there may be several jobs in the system that are ready to send their message at the same time, the system uses a scheduling algorithm to define the order in which the jobs access the network. The scheduler works based on fixed task-level priorities. That is, every task τ_i is assigned a constant priority that is passed on to all its jobs. We denote by $\mathrm{hp}(\tau_i)$ (resp., $\mathrm{lp}(\tau_i)$) the set of tasks of higher (resp., lower) priority than τ_i and by $\mathrm{hep}(\tau_i)$ (resp., $\mathrm{lep}(\tau_i)$) the set $\mathrm{hp}(\tau_i) \cup \{\tau_i\}$ (resp., $\mathrm{lp}(\tau_i) \cup \{\tau_i\}$).

The scheduler is designed so that at any point in time, if two jobs are ready to send their message and compete for the access to the network, it is the job coming from the task with the highest priority that will be granted the access first (since the task priorities are passed on to the jobs). This implicitly assumes that all the tasks have distinct priorities and there is at most one job per task that is ready to send its message at any point in time. The latter is guaranteed by the model itself since $D_i \leq T_i$ for all τ_i and thus if two jobs of the same task ever compete for the access to the network at the same time, then one of them has failed to meet its deadline, which is not acceptable.

The relation between the priority of the tasks and their criticality is simple. As explained earlier, the tasks of a same criticality (say χ_k) are grouped together in $\tau(k)$. All the tasks in the system have a different priority, the only constraint imposed on the system is that all the tasks of a same group $\tau(k)$ have a higher priority than all the tasks of all the groups $\tau(j)$ for $j > k$. We believe that this is a reasonable assumption in a real-world scenario: even if important tasks may have longer deadlines and periods

than less important messages – and thus, in theory, the system could potentially serve the less important tasks first – by giving a higher priority to the more important tasks we force them to be served before the less important ones. This is highly recommended as the network speed can change at any moment and we do not want to delay the transmission of important messages (and take the risk of not being able to send them on time) to benefit less critical ones. Note that we do not impose any further restriction on the priority assignment, meaning that any fixed-priority assignment can be used to define the priority of the tasks within each group.

From this point onwards, we shall assume that every task in the system has been assigned a priority such that the above restriction is respected and the entire system is schedulable (i.e. all the deadline are met) at a network speed of 1 Mbit/s. To make sure that the second assumption is always verified, the data rate unit (here, Mbit/s) can be assumed to be Gbit/s or even Tbit/s provided that all the messages lengths C_i, $\forall i$, are converted and expressed in that new data rate. In the remainder of the paper, we shall assume Mbits/s to be the reference network speed and the unit in which the C_i values are expressed.

3. Problem statement

While vehicles move from one part of the city to another, the network speed naturally varies along with the strength of the network signal. As the signal gets weaker, it may become infeasible to send all the information and still verify the timing requirement of every task. To make sure that the critical information is still sent on time, we define x thresholds $\{\mathrm{sp}_1, \mathrm{sp}_2, \ldots, \mathrm{sp}_x\}$ within the speed domain such that, each time the current speed drops below one of these thresholds, say sp_k, all the tasks from the criticality groups $\tau(j)$, with $j > k$, are immediately suspended. They will no longer send messages until the network speed raises again above the threshold sp_k. If one of these tasks with a lower criticality than χ_k was sending a message when the network speed crossed the threshold, then it cannot be stopped (since sending is not preemptible) and it must continue until its entire message has been sent. After that, the task is suspended together with all the tasks of its criticality level and lower levels. The problem is thus to find the minimum speeds sp_k, for all $k \in [1, x]$, such that the groups of tasks $\tau(1)$, $\tau(2)$, ..., $\tau(k)$ is schedulable (i.e. meets all the deadlines). These minimum speeds will then be used as the thresholds described above.

4. Prerequisite

First, let us introduce some extra notations, properties, and basic concepts that we shall use in the computation of our solution and the proof of its optimiality. For a complete state of the art on the non preemptive scheduling problem, the interested reader can refer to [2], [3], [4], [5] amongst many other publications. We just recapitulate here the main results we need.

In the analysis of the optimality of our solution, the time windows of interest are the so-called level-i busy windows. When considering the scheduling of n tasks $\tau_1, \ldots, \tau_n$ indexed by decreasing order of priority, the level-i busy window (with $i \in [1, n]$) is the *longest* time window in which (1) only the jobs from tasks $\tau_1, \ldots, \tau_i$ send their messages (except the first job as we will discuss later) and (2), there is not a single instant at which no message is being sent (the network is never idle within that window). It has been proven in Lemma 6 of [4] (Section 4.3) that the job-release scenario that leads to the longest level-i busy window is the one at which all the tasks $\tau_1, \tau_2, \ldots, \tau_i$ release a job at a same time (say, t) *and* the task $\tau_k \in \mathrm{lp}(\tau_i)$ with the longest C_k releases a job at time $t - \epsilon$. Assuming that no job is sending a message at time $t - \epsilon$, that job from τ_k starts to send his immediately upon its release. That is, the level-i busy window begins at time t with the sending of the longest lower-priority message (here, that of τ_k). This contribution of a lower priority task to the level-i busy window is called the "blocking term" B_i and is calculated as

$$B_i = \max_{k \in \mathrm{lp}(\tau_i)} \{C_k - \epsilon\} \tag{1}$$

The ϵ time units subtracted from the C_k's account for the ϵ time units during which τ_k is already sending its message before the beginning of the busy window (i.e., between the instants $t - \epsilon$ and t). Starting with an initial length $L_i = B_i$, the length of the level-i busy window can be computed by iteratively adding to L_i the contribution of τ_i and all the higher priority tasks that release jobs during these L_i time units. Formally, it has been proven in Theorem 15 of [4] that the length L_i of the level-i busy window, for $i \in [1, n]$, is the first fixed-point solution of the equation

$$L_i = B_i + \sum_{j \in \mathrm{hep}(\tau_i)} \left(1 + \left\lfloor \frac{L_i}{T_j} \right\rfloor\right) \times C_j \tag{2}$$

It is also demonstrated in the same article that the maximum *response time*[1] of the jobs of τ_i is always observed within the level-i busy window. It is thus sufficient to verify that all the deadlines are met for the jobs of τ_i within that time window to conclude on the schedulability of τ_i. We also know that the number n_i of jobs of τ_i released in the level-i busy window is given by

$$n_i = 1 + \left\lfloor \frac{L_i}{T_i} \right\rfloor \tag{3}$$

If we re-index the n_i jobs of τ_i that are released in the level-i busy window from 0 to $n_i - 1$, then the latest time at which job $\tau_{i,j}$ with $j \in [0, n_i - 1]$ starts sending its message is given by

$$w_{i,j} = B_i + j \times C_i + \sum_{k \in \mathrm{hp}(\tau_i)} \left(1 + \left\lfloor \frac{w_{i,j}}{T_k} \right\rfloor\right) \times C_k \tag{4}$$

Finally, for each such job $\tau_{i,j}$, with $j \in [0, n_i - 1]$, released in the level-i busy window, we denote by $S_{i,j}$ the set of all the time-instants between its release at time jT_i and its deadline at time $jT_i + D_i$, at which the higher priority tasks release a job. Formally, for given task

1. The response time of a job is the time between its release and its completion.

and job indexes i and j ($j \in [0, n_i - 1]$), the set $S_{i,j}$ is computed as

$$S_{i,j} = \bigcup_{k \in \text{hp}(\tau_i)} \left\{ rT_k \mid r \in \mathbb{N}^+ \text{ and } jT_i < rT_k < jT_i + D_i \right\} \bigcup \left\{ jT_i, jT_i + D_i \right\} \tag{5}$$

It is important to note that the time-instants jT_i and $jT_i + D_i$ corresponding to the release and deadline of $\tau_{i,j}$ are also included in the set $S_{i,j}$ (in the second line of the equation).

5. Our solution

With the notations introduced in the previous section, we shall prove that for any set of n periodic tasks $\tau_1, \ldots, \tau_n$ (indexed in decreasing order of priority), the minimum speed sp that allows for all the deadlines to be met is given by

$$\text{sp} = \max_{i \in [1,n]} \left(\max_{j \in [0, n_i - 1]} \left(\min_{t \in S_{i,j}} \left(\max \left(\frac{w_{i,j}}{t - \epsilon}, \frac{w_{i,j} + C_i}{jT_i + D_i} \right) \right) \right) \right) \tag{6}$$

The rational behind that equation is that the minimum speed is, for each job, either constrained by its nececity to start on time ($\frac{w_{i,j}}{t-\epsilon}$) or to finish before its deadline ($\frac{w_{i,j}+C_i}{jT_i+D_i}$). The more constrained job has to be search amongst all the jobs in the set described in the previous section, for all higher priority tasks, and the process must be iterate for each task. A formal proof is given in the next section (see Lemma 1).

A solution to the problem described in Section 3 can easily be derived from Equ. 6. Given a set of n tasks $\tau_1, \tau_2, \ldots, \tau_n$ grouped in x groups $\{\tau(1), \tau(2), \ldots, \tau(x)\}$ such that

1) every task of the group $\tau(\ell)$, $\ell \in [1, x]$, has criticality χ_ℓ
2) every task of the group $\tau(\ell)$, $\ell \in [1, x-1]$, has a higher priority than all the tasks of $\tau(\ell + 1)$

the minimum speed sp_k, $k \in [1, x]$, such that all the groups $\tau(1)$, $\tau(2)$, ..., $\tau(k)$ meet their deadlines is given by the speed sp computed by Equ. 6, where the n tasks considered in the computation are the tasks in all the groups $\tau(1), \tau(2), \ldots, \tau(k)$. In the next section we demonstrate the correctness and optimality of Equ. 6.

6. Proof of optimality

We now show how to compute the exact minimum speed satisfying the deadlines of a given task set in the case of non-preemptive fixed priority scheduling. This computation was proposed in the state of the art by dichotomy (see section 7), we here explicit the exact formula.

Lemma 1. *For any set τ of n periodic tasks $\tau_1, \ldots, \tau_n$ (indexed in decreasing order of priority), the exact minimum speed that allows for all the deadlines to be met is given by* sp *as defined in Equ. 6.*

Proof. In this proof, not only we demonstrate that all the deadlines are met when running all the n tasks at speed sp as defined by Equ. 6, but also that any slower speed $\text{sp}^{\text{low}} < \text{sp}$ does *not* allow to meet all the deadlines.

Let p be any value of $i \in [1, n]$ that maximizes the outermost "max" operator of Equ. 6. Likewise, let q denote any value of j that maximizes the second outermost "max" operator. That is, Equ. 6 is maximized for $i = p$ and $j = q$ and we can rewrite it as

$$\text{sp} = \min_{t \in S_{p,q}} \left(\max \left(\frac{w_{p,q}}{t - \epsilon}, \frac{w_{p,q} + C_p}{qT_p + D_p} \right) \right) \tag{7}$$

Let $t_1, t_2, \ldots, t_x$ denote all the time-instants of $S_{p,q}$ sorted in increasing order, i.e., $t_1 < t_2 < \ldots < t_x$. We know from Equ. 5 that $x \geq 2$ as $S_{p,q}$ contains at least the two time-instants $t_1 = qT_p$ and $t_x = qT_p + D_p$. Now, let t_r be any of the time-instants $\in S_{p,q}$ that minimizes the outermost "min" operator of Equ. 7, we can then further simplify it as

$$\text{sp} = \max \left(\frac{w_{p,q}}{t_r - \epsilon}, \frac{w_{p,q} + C_p}{qT_p + D_p} \right) \tag{8}$$

From this simplified equation, two cases must be studied.

Case 1: $\dfrac{w_{p,q}}{t_r - \epsilon} \geq \dfrac{w_{p,q} + C_p}{qT_p + D_p}$

In this case, we get $\text{sp} = \frac{w_{p,q}}{t_r - \epsilon}$ and *at that speed*, the q^{th} job $\tau_{p,q}$ of task τ_p in the level-p busy window starts sending its message at time $\frac{w_{p,q}}{\text{sp}} = \frac{w_{p,q}}{\frac{w_{p,q}}{t_r - \epsilon}} = t_r - \epsilon$ and thus slightly before time t_r. At this time, $\tau_{p,q}$ starts to send its message in a non-preemptive manner until it finishes and since $\text{sp} \geq \frac{w_{p,q}+C_p}{t_x}$ (from the case), we know that it will finish before its deadline at time t_x. Note that by Equ. 6, we know from the first two maximum operators that sp is higher than (or equal to) the minimum speed required to execute every job from every other task on time.

Now, we show that in this case, any speed $\text{sp}_{\min}$ lower than sp does not allow all the deadlines to be met. By contradiction, suppose that $\text{sp}_{\min} < \text{sp}$ does allow to meet all the deadlines. Since by assumption $\text{sp}_{\min} < \text{sp} = \frac{w_{p,q}}{t_r - \epsilon}$, it holds that at speed $\text{sp}_{\min}$ the job $\tau_{p,q}$ starts to send its message later than time $t_r - \epsilon$ but before t_x (because we assumed that all the deadlines are met at speed $\text{sp}_{\min}$). Let t_{u-1} and t_u be the two consecutive time-instants in $S_{p,q}$ such that $t_r \leq t_{u-1} \leq \frac{w_{p,q}}{\text{sp}_{\min}} < t_u$. We know that those two instants exist since in the worst-case scenario, we have $t_u = t_x$ and we know that $\frac{w_{p,q}}{\text{sp}_{\min}} < \frac{w_{p,q}+C_p}{\text{sp}_{\min}} < t_x$ (because we assumed that $\text{sp}_{\min}$ allows to meet all the deadlines). Let sp_u be the speed calculated by Equ. 7 when considering $t = t_u \in S_{p,q}$ in the "min" operator, i.e. $\text{sp}_u = \max \left(\frac{w_{p,q}}{t_u - \epsilon}, \frac{w_{p,q}+C_p}{t_x} \right)$. Because of that minimum, we know that the speed sp computed previously at time t_r is $\leq \text{sp}_u$ and again, two cases may arise for sp_u.

Case 1.1: $\dfrac{w_{p,q}}{t_u - \epsilon} \geq \dfrac{w_{p,q} + C_p}{t_x}$

From the case, it holds that $\text{sp}_u = \frac{w_{p,q}}{t_u - \epsilon}$ and thus at that speed, the job $\tau_{p,q}$ starts to send its message at time $t_u - \epsilon$. Therefore, since $\text{sp}_{\min} < \text{sp} \leq \text{sp}_u$, $\tau_{p,q}$ finishes *after* time t_u if processed at speed $\text{sp}_{\min}$. This

contradicts our definition of t_u that imposes $\frac{w_{p,q}}{\mathrm{sp}_{\min}} < t_u$.

Case 1.2: $\dfrac{w_{p,q}}{t_u - \epsilon} < \dfrac{w_{p,q} + C_p}{t_x}$

Here, $\mathrm{sp}_{\min} < \mathrm{sp}_u = \frac{w_{p,q}+C_p}{t_x}$. If $\mathrm{sp}_{\min} \leq \frac{w_{p,q}}{t_u-\epsilon}$ then we obtain the same contradiction as in Case 1.1. Therefore, it must hold that $\frac{w_{p,q}}{t_u-\epsilon} < \mathrm{sp}_{\min} < \frac{w_{p,q}+C_p}{t_x}$, which also leads to a contradiction because at that speed, (1) the job $\tau_{p,q}$ starts sending its message after time $t_u - \epsilon$ and (2), $\frac{w_{p,q}+C_p}{t_x}$ is the lowest speed at which $\tau_{p,q}$ can possibly send its message before time t_x. Therefore, all the deadlines cannot be met at speed $\mathrm{sp}_{\min} < \frac{w_{p,q}+C_p}{t_x}$.

Case 2: $\dfrac{w_{p,q}}{t_r - \epsilon} < \dfrac{w_{p,q} + C_p}{qT_p + D_p}$

In this case, we have $\frac{w_{p,q}}{t_r-\epsilon} < \mathrm{sp} = \frac{w_{p,q}+C_p}{qT_p+D_p}$. Let us first show that that speed sp allows to meet all the deadlines. At that speed, the job $\tau_{p,q}$ starts sending its message before time $t_r - \epsilon$ (since $\mathrm{sp} > \frac{w_{p,q}}{t_r-\epsilon}$). Then, since $\mathrm{sp} = \frac{w_{p,q}+C_p}{t_x}$ (from the case), we know it finishes sending its message before its deadline at time t_x. Note that by Equ. 6, we know from the two outermost "max" operators that sp is higher than (or equal to) the minimum speed required to execute every job from every other task on time.

Now, let us show that similarly to case 1, any speed $\mathrm{sp}_{\min}$ lower than sp does not allow all the deadlines to be met. By contradiction, suppose that there exists $\mathrm{sp}_{\min} < \mathrm{sp}$ that *does* allow to meet all the deadlines. Two cases may arise:

Case 2.1: $\dfrac{w_{p,q}}{t_r - \epsilon} \leq \mathrm{sp}_{\min} < \mathrm{sp} = \dfrac{w_{p,q} + C_p}{t_x}$

In this case, $\mathrm{sp}_{\min} \geq \frac{w_{p,q}}{t_r-\epsilon}$ and thus at speed $\mathrm{sp}_{\min}$ the job $\tau_{p,q}$ starts sending its message at time $t_r - \epsilon$. Since $\mathrm{sp}_{\min} < \frac{w_{p,q}+C_p}{t_x}$ (from the case), it cannot finish sending the message by its deadline at time t_x.

Case 2.2: $\mathrm{sp}_{\min} \leq \dfrac{w_{p,q}}{t_r - \epsilon} < \mathrm{sp} = \dfrac{w_{p,q} + C_p}{t_x}$

At that speed $\mathrm{sp}_{\min}$, the job $w_{p,q}$ starts to send its message after time $t_r - \epsilon$ (say at time $t^* > t_r - \epsilon$). At that time t^*, all the higher priority jobs that have arrived at every time-instant $t_u \in S_{p,q}$, with $t_r \leq t_u \leq t^*$, will have priority over $\tau_{p,q}$. However, knowing that $\mathrm{sp}_{\min} < \frac{w_{p,q}+C_p}{t_x}$, even without these extra jobs arrived within $[t_r, t^*]$ the speed $\mathrm{sp}_{\min}$ is already to slow for $\tau_{p,q}$ to finish sending its message by the deadline at time t_x.

The proof has successfully covered all possible cases and showed for each one that the speed sp as defined by Equ. 6 is the *lowest* speed at which the n tasks $\tau_1, \tau_2, \ldots, \tau_n$ can meet their deadlines.

□

7. Related Work

A common performance measurement tool for scheduling algorithms is the so called speedup factor. This speedup factor represents the minimum factor by which the processor speed should be increased in order for a given algorithm α to schedule any task-set which is schedulable by an optimal algorithm. In that sense, it gives a metric to measure the gap between α and an optimal algorithm. In order to compute the value, or bounds, of this speedup factor, one has also to compute the critical scaling factor (the maximal factor by which each tasks cost can be increase in order to keep the task-set feasible), but not for any kind of task-set, only for special-case task-sets which model the limit cases. This concept is so not directly linked to our present work, but the interested reader can refer to [6].

Previous works have investigated on the maximal slowdown factor of a processor, particularly in the context of energy-aware scheduling or in the critical scaling factor, in the context of sensitivity analysis. These problems are similar: slowdown the processor is equivalent to increase the tasks length. However, this problem is mainly studied in the context of preemptive tasks. For example, Lehoczky et al. give a threshold value to compute the critical scaling factor under Rate Monotonic analysis in [7]. For non preemptive tasksets, the slowdown factor can be compute for EDF as explained in [8], but to the best of our knowledge, works addressing this issue for fixed priority non preemptive task-sets use binary search methods to approximate the slowdown/critical-scaling factor, as in [9].

In the case of a system where tasks can experience different execution durations, Mixed Criticality (MC) was introduced to arbitrate between critical and non critical tasks when the schedulability cannot be satisfied if both critical and non critical tasks experience their maximum WCET.

Mixed Criticality (MC) was first introduced by Vestal in [10] for periodic tasks and dual criticality systems HI and LO on a uniprocessor system.

A task can be of HI or LO criticality and is characterized by: a minimum inter-arrival time T (period), a relative deadline denoted D and a Worst Case Execution Time (WCET) denoted $C(LO)$ (respectively $C(HI)$) associated to LO (respectively HI) mode representing the budget of time given to a task for its execution. In the classical MC model, $C(HI) \geq C(LO)$ for HI tasks and the system starts in LO mode where both HI and LO tasks can execute and then can switch to HI mode as soon as any HI job executes for its $C(LO)$ without completing. In HI mode, only HI tasks can be executed, LO tasks are suspended with no more guarantee. The condition that triggers a mode change was only based on a WCET overrun w.r.t. the current criticality level [10]. Several approaches have been considered for constrained deadline tasks: EDF-VD [11], AMC [12].

This classical model has been extended to other ones:

- With two Fixed Priorities for a LO-crit task in LO and HI modes. The idea is to reduce the priority of LO-crit tasks in HI mode such that LO-crit tasks do not interfere with HI-crit tasks in HI mode [13].
- By reducing the budget of some of the LO-crit tasks in HI mode [13].
- By increasing the period of LO-crit tasks in HI mode. This approach has been investigated with the elastic model [14] applied to EDF scheduled tasks in the context of MC [15]. The aim of the elastic model was to adapt the Quality–of–Service of a system or handle overloaded situations by modifying the periods of the tasks during the execution of the system. The period of a LO task

is higher when the system criticality is HI than in LO mode.

In this paper, we consider the case were the criticality of a task is such that any task τ_i of arbitrary criticality has a higher priority than any task of lower criticality than τ_i.

8. Conclusion and Future work

In this paper, we considered the problem of sending information from a mobile ITS vehicle to a central entity according their importance. We supposed a wireless link whose quality may vary according to the location of the vehicle. We characterized the exact speed at which all messages of the same or higher importance can be sent. This leads to define a set of speed thresholds where the transmisison of less important messages should be stopped. We assumed a non-preemptive fixed priority scheduling where less important messages all have a lower priority than any higher priority messages. Messages of the same importance can have distinct priorities. The speed thresholds we obtain are optimal in that context.

As a further work, we will explore optimal fixed priority assignement strategies when no restriction is imposed on the priority of messages. In [4], it is showed that Ausley's priority assignment is optimal for non-preemptive fixed priority scheduling. We would like to explore a robust priority assignement minimizing the speed of the thresholds at which less important messages should be stopped. We then would like to propose an algorithm minimizing the number of speed thresholds while satisfying the deadlines of messages.

References

[1] EU, "on the framework for the deployment of intelligent transport systems in the field of road transport and for interfaces with other modes of transport," *Official Journal of the European Union*, Jul. 2010. [Online]. Available: https://goo.gl/XwoFHX

[2] K. W. Tindell, H. Hansson, and A. J. Wellings, "Analysing real-time communications: controller area network (can)," in *1994 Proceedings Real-Time Systems Symposium*, Dec 1994, pp. 259–263.

[3] K. Tindell, A. Burns, and A. Wellings, "Analysis of hard real-time communications," *Real-Time Systems*, vol. 9, pp. 147–171, 1994.

[4] L. George, N. Rivierre, and M. Spuri, "Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling," INRIA, Research Report RR-2966, 1996, projet REFLECS. [Online]. Available: https://hal.inria.fr/inria-00073732

[5] R. J. Bril, J. J. Lukkien, and W. F. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption," *Real-Time Syst.*, vol. 42, no. 1-3, pp. 63–119, Aug. 2009. [Online]. Available: http://dx.doi.org/10.1007/s11241-009-9071-z

[6] R. I. Davis, A. Burns, S. Baruah, T. Rothvoß, L. George, and O. Gettings, "Exact comparison of fixed priority and edf scheduling based on speedup factors for both pre-emptive and non-pre-emptive paradigms," *Real-Time Systems*, vol. 51, no. 5, pp. 566–601, Sep 2015. [Online]. Available: https://doi.org/10.1007/s11241-015-9233-0

[7] J. Lehoczky, L. Sha, and Y. Ding, *Rate monotonic scheduling algorithm: Exact characterization and average case behavior*. Publ by IEEE, 1989, pp. 166–171.

[8] R. Jejurikar and R. Gupta, "Energy aware non-preemptive scheduling for hard real-time systems," in *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, July 2005, pp. 21–30.

[9] S. Punnekkat, R. Davis, and A. Burns, *Sensitivity analysis of real-time task sets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 72–82.

[10] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Real-Time Systems Symposium*, Dec 2007, pp. 239–243.

[11] S. Baruah, V. Bonifaci, G. D'Angelo, 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 *Euromicro Conference on Real-Time Systems*, July 2012, pp. 145–154.

[12] S. K. Baruah, A. Burns, and R. I. Davis, "Response-time analysis for mixed criticality systems," in *Real-Time Systems Symposium*, Nov 2011, pp. 34–43.

[13] A. Burns and S. Baruah, "Towards a more practical model for mixed criticality systems," in *Workshop on Mixed Criticality Systems, Real-Time Systems Symposium*, december 2013, pp. 1–6.

[14] G. C. Buttazzo, G. Lipari, and L. Abeni, "Elastic task model for adaptive rate control," in *Real-Time Systems Symposium*, Dec 1998, pp. 286–295.

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

Response Time Analysis for Mixed Criticality Systems with Arbitrary Deadlines

Alan Burns
Department of Computer Science,
University of York, UK.
Email: burns@cs.york.ac.uk

Robert I. Davis
Department of Computer Science,
University of York, UK.
Email: rob.davis@cs.york.ac.uk

***Abstract*—This paper extends analysis of the Adaptive Mixed Criticality (AMC) scheme for fixed-priority preemptive scheduling of mixed-criticality systems to include tasks with arbitrary deadlines. Both of the previously published schedulability tests, AMC-rtb and AMC-max are extended to cater for tasks with deadlines that may be greater than their periods. Evaluations show that the simpler method, AMC-rtb-Arb, remains a viable approach that performs almost as well as the more complex alternative, AMC-max-Arb, when tasks with arbitrary deadlines are considered.**

I. Introduction

Since the publication of Vestal's model [14] there has been a significant number of papers on the scheduling of Mixed Criticality Systems (MCS) (see [8] for a survey). Somewhat surprisingly none of the papers on fixed priority scheduling of MCS have addressed tasks with so called *arbitrary deadlines* that may be greater than their periods. In this paper we consider this issue within the context of fixed-priority preemptive systems scheduled according to the Static Mixed Criticality (SMC) [4] and Adaptive Mixed Criticality (AMC) [5], [1] schemes.

Arbitrary-deadline tasks cater for situations where there is some leeway in when a task must execute. For example, a consumer task that reads items from a buffer must, over a long time interval, consume items at the same rate as they are produced; however, the task can have response times that are longer than its period or some multiple of its period, provided that buffer has sufficient space to store unread items. A task set may be unschedulable if its deadlines are constrained (i.e. less than or equal to their periods), but may meet all of its time constraints if a few tasks are allowed to have arbitrary deadlines.

The remainder of the paper is organized as follows. In Section II, we outline the standard MCS model. In Section III, we recap on the extended form of response-time analysis for arbitrary-deadline tasks. Section V then reviews existing schedulability analysis for SMC and AMC, assuming tasks with constrained deadlines. The main contribution of the paper is the new response-time analyses derived in Section VI for the AMC scheme, catering for arbitrary-deadline task sets. Section VII discusses priority assignment, while Section VIII evaluates the performance of the various schedulability tests. Section IX concludes.

II. System Model, Terminology and Notation

In this paper, we are interested in the Fixed Priority Preemptive Scheduling (FPPS) of a single processor MCS comprising a static set of n sporadic tasks. Each task, τ_i, is defined by its period (or minimum inter-arrival time), relative deadline, worst-case execution time (WCET), criticality level, and unique priority: (T_i, D_i, C_i, L_i, P_i). Task deadlines may be arbitrary, i.e. less than, equal to or greater than their periods.

We assume that each task τ_i gives rise to a potentially unbounded sequence of jobs, with the release of each job separated by at least the minimum inter-arrival time from the release of the previous job of the same task. The worst-case response time of task τ_i is denoted by R_i and corresponds to the longest response time from release to completion for any of its jobs. For tasks with arbitrary deadlines, more than one job of the same task may be active at any given time. Among jobs of the same task, those released earlier are executed first.

The system is assumed to be defined over two criticality levels (HI and LO). Each LO-criticality task τ_j is assumed to have a single estimate of its WCET: $C_j(LO)$, while each HI-criticality task τ_k has two estimates: $C_k(HI)$ and $C_k(LO)$, with $C_k(HI) \geq C_k(LO)$. (Note we drop the task index when using these and other terms in a generic way, but include the index when referring to the parameters of a specific task). Most scheduling approaches for MCS identify different modes of behavior. In the LO-criticality (or normal) mode, all tasks execute within their $C(LO)$ bounds and all deadlines are required to be met. At all times LO-criticality tasks are constrained by run-time monitoring to execute for no more than their $C(LO)$ bound. In contrast, if a HI-criticality task executes for $C(LO)$ without signaling completion then the system enters HI-criticality mode. In this mode only HI-criticality tasks are required to meet their deadlines. HI-criticality tasks are assumed to execute for no more than $C(HI)$. The response time of a task τ_i in LO-criticality mode is denoted by $R_i(LO)$ and in HI-criticality mode by $R_i(HI)$.

III. Existing Analysis for FPPS

For constrained-deadline tasks ($D_i \leq T_i$) in a single criticality level system, the response time of task τ_i can be computed as follows (see [11], [3] for a full derivation):

$$R_i = C_i + \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad (1)$$

where $\mathbf{hp(i)}$ is the set of tasks with higher priority than τ_i. This and all subsequent response time equations can be solved via fixed point iteration. Iteration starts with a value of $R_i = 0$ and continues either until convergence or until the computed response time exceeds the task's deadline.

For tasks with arbitrary deadlines ($D_i \geq T_i$) then it is possible in a schedulable system that $R_i \geq T_i$. It follows that there can be more than one job of task τ_i active within the same priority level-i busy period; any of these jobs can give rise to the worst-case response time for the task. We use q as an index to denote each job within the busy period, with $q = 0$ indicating the first job. The completion time of each job of the task, $r_i(q)$ ($0 \leq q \leq p$), as measured from the start of the busy period, can be computed as follows (see [13] for a full derivation):

$$r_i(q) \;=\; (q+1)C_i \;+\; \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{r_i(q)}{T_j} \right\rceil C_j \qquad (2)$$

The last job in the busy period is denoted by p, which is the first value where completion of the job occurs before the next release of the task, i.e. where $r_i(p) \leq (p+1)T_i$. The response-time of each job q is calculated as follows:

$$\forall q_{(0 \leq q \leq p)} : \quad R_i(q) = r_i(q) - qT_i \qquad (3)$$

With the worst-case response time of the task given by:

$$R_i \;=\; \max_{\forall q_{(0 \leq q \leq p)}} \{R_i(q)\} \qquad (4)$$

This analysis for FPPS scheduling of arbitrary-deadline tasks can be applied to MCS by simply assuming that all HI-criticality tasks have a single execution time of $C(HI)$ and all LO-criticality tasks have a single execution time of $C(LO)$.

IV. Existing Analysis for SMC

Static Mixed Criticality (SMC) [4] is a simple scheme that extends Vestal's original approach [14] with run-time monitoring. Under SMC, LO-criticality tasks continue to be released and to execute in HI-criticality mode; however, they are not required to meet their deadlines in that mode. (Recall that all tasks are required to meet their deadlines in LO-criticality mode).

Under SMC, the worst-case response times for constrained-deadline tasks may be computed as follows:

$$\begin{aligned} R_i(L_i) \;&=\; C_i(L_i) \\ &+\; \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i(L_i)}{T_j} \right\rceil C_j(\min(L_i, L_j)) \end{aligned} \qquad (5)$$

We note that the arbitrary-deadline analysis for FPPS given by (2), (3), and (4) can be adapted to SMC by changing C_i to $C_i(L_i)$ and C_j to $C_j(\min(L_i, L_j))$.

V. Existing Analyses for AMC

With Adaptive Mixed Criticality (AMC) [5], if a HI-criticality task executes for $C(LO)$ without completing, then the system enters HI-criticality mode. AMC differs from SMC in that in HI-criticality mode, previously released jobs of LO-criticality tasks may be completed, but subsequent releases of LO-criticality tasks are not started. Similar to SMC, only HI-criticality tasks are required to be schedulable in HI-criticality mode. Also, in common with SMC, LO-criticality mode may be re-entered when the processor becomes idle.

In the original paper [5] two sufficient schedulability tests were developed for AMC. The first approach, called AMC-rtb (where rtb denotes 'response time bound') takes account of a bound on the duration over which LO-criticality tasks can interfere. The second, more complex approach is called AMC-max; it determines the maximum response time assuming all possible times at which the criticality mode change could occur.

A. AMC-rtb Analysis

The AMC-rtb analysis first computes the worst-case response time for each task τ_i in the LO-criticality mode:

$$R_i(LO) = C_i(LO) + \sum_{\tau_j \in \mathbf{hp}(i)} \left\lceil \frac{R_i(LO)}{T_j} \right\rceil C_j(LO) \qquad (6)$$

During the mode change and subsequent HI-criticality mode only HI-criticality tasks are required to be schedulable. The worst-case response time of a HI-criticality task τ_i can be computed as follows:

$$\begin{aligned} R_i(HI) \;=\; C_i(HI) \;+\; \sum_{\tau_j \in \mathbf{hpH}(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(HI) \;+ \\ \sum_{\tau_k \in \mathbf{hpL}(i)} \left\lceil \frac{R_i(LO)}{T_k} \right\rceil C_k(LO) \end{aligned} \qquad (7)$$

where $\mathbf{hpH(i)}$ is the set of HI-criticality tasks with higher priority than τ_i, and $\mathbf{hpL(i)}$ is the set of LO-criticality tasks with higher priority than τ_i. Note that higher priority LO-criticality tasks released after $R_i(LO)$ cannot cause interference, since if task τ_i has not completed by $R_i(LO)$ then the system must have switched to HI-criticality mode.

B. AMC-max analysis

The AMC-rtb analysis given in (7) is potentially pessimistic in that it assumes all jobs of higher priority HI-criticality tasks contribute $C(HI)$, and yet also assumes that the mode change takes place as late as possible at $R(LO)$ allowing jobs of higher priority LO-criticality tasks to contribute interference of $C(LO)$ until that point. This pessimism is avoided by the AMC-max analysis, which assumes that a mode change takes place at some time s, measured from the start of the busy period for task τ_i. All jobs of higher priority tasks that complete before that time can cause interference equating to at most their $C(LO)$ values, and after that time, no new jobs of LO-criticality tasks can be released and contribute interference. (Note that the AMC-max analysis for tasks in LO-criticality mode is the same as that for AMC-rtb, i.e. (6)).

AMC-max computes the worst-case response time $R_i^s(HI)$ of HI-criticality task τ_i, assuming a mode change at time s, and then takes the maximum of these values over all possible values of s.

$$R_i^s(HI) = C_i(HI) + I_L(s) + I_H(s) \qquad (8)$$

where $I_L(s)$ is the interference from higher priority LO-criticality tasks, and $I_H(s)$ is the interference from higher priority HI-criticality tasks.

As jobs of higher priority LO-criticality tasks are prevented from being released after the mode change at time s, their worst-case interference is upper bounded by:

$$I_L(s) = \sum_{j \in \mathbf{hpL}(i)} \left(\left\lfloor \frac{s}{T_j} \right\rfloor + 1 \right) C_j(LO) \qquad (9)$$

$I_H(s)$ is defined by considering the number of jobs of each higher priority HI-criticality task τ_k that can execute in

a busy period of length t, with the mode change taking place at time $s < t$. Only those jobs that may have some part of their execution after time s can contribute interference of $C(HI)$, with the remainder contributing $C(LO)$.

The maximum number of jobs of τ_k with $D_k \leq T_k$ that can fit into an interval of length $t - s$ is bounded by:

$$\left\lceil \frac{t - s - (T_k - D_k)}{T_k} \right\rceil + 1 \tag{10}$$

Equation (10) can be pessimistic; including more jobs than can actually be present in an interval of length t. This is taken into account by defining:

$$M(k, s, t) = \min \left\{ \left\lceil \frac{t - s - (T_k - D_k)}{T_k} \right\rceil + 1, \left\lceil \frac{t}{T_k} \right\rceil \right\} \tag{11}$$

where $M(k, s, t)$ is the maximum number of jobs of τ_k that can exhibit HI-criticality behavior in a busy period of length t with a transition to HI-criticality mode at time s. The interference term for higher priority HI-criticality tasks thus becomes:

$$I_H(s) = \sum_{k \in \mathbf{hpH}(i)} \Big\{ M(k, s, t)\ C_k(HI) + \left(\left\lceil \frac{t}{T_k} \right\rceil - M(k, s, t) \right) C_k(LO) \Big\} \tag{12}$$

Hence the worst-case response time with a mode change at time s is given by:

$$R_i^s(HI) = C_i(HI) + \sum_{j \in \mathbf{hpL}(i)} \left(\left\lfloor \frac{s}{T_j} \right\rfloor + 1 \right)\ C_j(LO) + \sum_{k \in \mathbf{hpH}(i)} \Big\{ M(k, s, R_i^s) C_k(HI) + \left(\left\lceil \frac{t}{T_k} \right\rceil - M(k, s, R_i^s) \right)\ C_k(LO) \Big\} \tag{13}$$

The worst-case response time of the task is then the maximum over all possible values of s:

$$R_i^*(HI) = \max_{\forall s} (R_i^s(HI)) \tag{14}$$

Finally, it is necessary to limit the number of values of s that that are considered from the range of possible values $[0, R_i^{LO}]$. In (13), the $\mathbf{hpL}$ term increases as a step function with increasing values of s, while the $\mathbf{hpH}$ term decreases. It follows that $R_i^s(HI)$ can only increase at values of s corresponding to multiples of the periods of LO-criticality tasks – hence these are the only values of s that need to be considered. Note that a LO-criticality job released at exactly R_i^{LO} will not be allowed to execute, and hence s is restricted to the interval $[0, R_i^{LO})$.

VI. Arbitrary-Deadline Analysis for AMC

In this section, we present new analysis for arbitrary-deadline tasks scheduled according to AMC. The two new methods are referred to as AMC-rtb-Arb and AMC-max-Arb ("Arb" meaning arbitrary deadline). They build on the existing analysis for AMC-rtb, AMC-max [5] and the classical arbitrary-deadline analysis for FPPS [13].

A. AMC-rtb-Arb Analysis

The strategy behind AMC-rtb is to count interference from each job of a higher priority HI-criticality task τ_k as $C_k(HI)$, and to assume that each job of a higher priority LO-criticality task τ_j released up to the LO-criticality response time of the task under analysis τ_i causes interference of $C_j(LO)$. We adapt this strategy to the case of arbitrary deadlines.

First we consider each task τ_i in LO-criticality mode. The length $r_i^L(q)$ of the busy period in LO-criticality mode up to completion of job q of τ_i is given by:

$$r_i^L(q) \ = \ (q+1)C_i(LO) \ + \ \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{r_i^L(q)}{T_j} \right\rceil C_j(LO) \tag{15}$$

The worst-case response time of each job is therefore:

$$\forall q_{(0 \leq q \leq p)} : \quad R_i(LO)(q) = r_i^L(q) - qT_i \tag{16}$$

and the worst-case response time of the task in LO-criticality mode is given by:

$$R_i(LO) \ = \ \max_{\forall q_{(0 \leq q \leq p)}} \{R_i(LO)(q)\} \tag{17}$$

As with the classical analysis for FPPS, iteration over the values of q ends at p, the smallest value such that $r_i^L(p) \leq (p+1)T_i$, indicating that $r_i^L(p)$ corresponds to the end of the priority level-i busy period in LO-criticality mode.

Analysis for each HI-criticality task τ_i considering the mode switch and HI-criticality behavior proceeds in a similar fashion. Compared to LO-criticality mode, due to the larger amounts of interference, it is possible that the busy period extends over more releases of jobs of task τ_i. Let v be the index of the last job of τ_i in this longer priority level-i busy period. Note that $v \geq p$, where p is the last release considered in LO-criticality mode.

When analyzing HI-criticality behavior, for each job q, we consider the longest time for which the system can remain in LO-criticality mode, and so LO-criticality tasks can be released. This is given by $r_i^L(q)$ provided that $q \leq p$. In the case that $q > p$ then it is not possible for LO-criticality mode to extend as far as the release of job q, since releases of LO-criticality jobs cannot take place beyond $r_i^L(p)$. The relevant equations become:

$$r_i^H(q) \ = \ (q+1)C_i(HI) \ + \ \sum_{j \in \mathbf{hpH}(i)} \left\lceil \frac{r_i^H(q)}{T_j} \right\rceil C_j(HI) \ + \ \sum_{k \in \mathbf{hpL}(i)} \left\lceil \frac{r_i^L(\min(q,p))}{T_k} \right\rceil C_k(LO) \tag{18}$$

$$\forall q (0 \leq q \leq v) : \quad R_i(HI)(q) = r_i^H(q) - qT_i \tag{19}$$

$$R_i(HI) \ = \ \max_{\forall q_{(0 \leq q \leq v)}} \{R_i(HI)(q)\} \tag{20}$$

This formulation shares similar pessimism to the constrained-deadline analysis on which it is based. As with AMC-rtb, it assumes that jobs of higher priority HI-criticality tasks cause interference of $C(HI)$ over the entire busy period, while higher priority LO-criticality tasks can interfere over the duration of a maximum length LO-criticality busy period. Often these two cases cannot occur together leading to pessimism. The following analysis, building on AMC-max, seeks to remove this pessimism.

B. AMC-max-Arb analysis for arbitrary-deadline MC tasks

We now extend the ARM-max analysis to arbitrary-deadline tasks. The analysis for all tasks in LO-criticality mode is the same as AMC-rtb-Arb, i.e. (15), (16), and (17). We therefore only consider the schedulability of HI-criticality tasks across the mode switch and in the subsequent HI-criticality mode. It turns out that the derivation, described in Section V-B for the constrained deadline case, applies with simple adaptations.

We compute the completion time of the qth job of task τ_i when the mode switch occurs at time s, as follows:

$$r_i^s(q) = XC_i(HI) + YC_i(LO) + I_L(s) + I_H(s) \qquad (21)$$

where $X+Y = q+1$. (The values of X and Y are determined below). Note (9) still provides an upper bound $I_L(s)$ on the interference from LO-criticality tasks.

Following the same argument as the AMC-max analysis, we need to determine an upper bound $I_H(s)$ on the interference that HI-criticality tasks such as τ_k can cause in a busy period of length t if the mode change occurs at time s, with $s < t$. To do so, we maximize the number of jobs of τ_k still potentially active at time s as all of these jobs can contribute interference of $C_k(HI)$, while all other jobs of τ_k contribute $C_k(LO)$. In an interval of length $t - s$ there can be at most

$$\left\lceil \frac{t - s + (D_k - T_k)}{T_k} \right\rceil + 1 \qquad (22)$$

active jobs of τ_k.

Equation (22) is identical to (10) that caters for $D \leq T$; we observe that it is also applicable when $D > T$. Since (11) and (12) remain unchanged, (21) can be used to compute $r_i^s(q)$, with $I_H(s)$ given by (12) and $I_L(s)$ given by (9).

The number X of jobs of the task under analysis τ_i which contribute $C_i(HI)$ can be derived in a similar way to (22). Accounting for the fact that there are at most $q+1$ active jobs in total, we have:

$$X = \min\left(\left\lceil \frac{t - s + (D_i - T_i)}{T_i} \right\rceil + 1, q + 1 \right)$$

The remaining steps are:

$$r_i^*(q) \;=\; \max_{\forall s} \left(r_i^s(q)\right) \qquad (23)$$

where s takes values corresponding to the release times of jobs of higher priority LO-criticality tasks in the interval $[0, r_i^L(q))$ with $r_i^L(q)$ given by (15).

$$\forall q_{(0 \leq q \leq p)} : \quad R_i(HI)(q) = r_i^*(q) - qT_i \qquad (24)$$

where p is the smallest value such that $r_i^*(p) \leq (p+1)T_i$. Finally, the worst-case response time is given by:

$$R_i(HI) \;=\; \max_{\forall q (0 \leq q \leq p)} \{R_i(HI)(q)\} \qquad (25)$$

VII. Priority Assignment

To maximize schedulability it is necessary to assign task priorities in an optimal way [10]. For arbitrary-deadline task sets scheduled under FPPS, and for constrained-deadline mixed-criticality task sets scheduled under SMC or AMC, an optimal priority ordering can be obtained via Audlsey's Optimal Priority Assignment (OPA) algorithm [2].

Davis and Burns [9] proved that it is both sufficient and necessary to show that a schedulability test meets three simple conditions in order for Audlsey's OPA algorithm to be applicable. These three conditions require that schedulability of a task according to the test is (i) independent of the relative priority order of higher priority tasks, (ii) independent of the relative priority order of lower priority tasks, (iii) cannot get worse if the task is moved up one place in the priority order (i.e. its priority is swapped with that of the task immediately above it in the priority order). We observe that these three conditions hold for all of the analyses derived in this paper, and thus Audsley's OPA algorithm is applicable.

VIII. Evaluation

In this section, we present an empirical evaluation of the schedulability tests introduced for mixed-criticality tasks with arbitrary deadlines.

A. Task set parameter generation

The task set parameters used in our experiments were randomly generated as follows:

- Task utilisations ($U_i = C_i/T_i$) were generated using the UUnifast algorithm [7], giving an unbiased uniform distribution of values.
- Task periods T_i were generated according to a log-uniform distribution with a factor of 100 difference between the minimum and maximum possible period.
- Task deadlines D_i were generated according to a log-uniform distribution in the range $[0.25, 4.0]T_i$.
- The LO-criticality execution time of each task was given by: $C_i(LO) = U_i \cdot T_i$.
- The HI-criticality execution time of each task was given by: $C_i(HI) = CF \cdot C_i(LO)$ where $CF = 2.0$.
- The probability that a generated task was of HI-criticality was given by the parameter $CP = 0.5$.

B. Schedulability tests investigated

We investigated the performance of the following schedulability tests. In all cases, we made use of Audsley's Optimal Priority Assignment (OPA) algorithm [2].

- UB-H&L-Arb: This is a *necessary* test which checks if all of the tasks are schedulable in LO-criticality mode and if the HI-criticality tasks are schedulable in HI-criticality mode (with no LO-criticality tasks executing). It ignores the mode switch. UB-H&L-Arb thus provides an upper bound on the performance of any fixed-priority fully-preemptive scheme for scheduling mixed-criticality, arbitrary-deadline tasks.
- AMC-max-Arb: described in Section VI-B.
- AMC-rtb-Arb: described in Section VI-A.
- SMC-Arb: Analysis for Static Mixed Criticality [4] using a straightforward adaptation of the standard arbitrary-deadline analysis for FPPS [13].
- FPPS-Arb: The standard arbitrary-deadline analysis for FPPS [13] described in Section III. This test requires that both LO- and HI-criticality tasks must be schedulable in both modes.

In addition to the above tests for arbitrary-deadline tasks, we also explored the performance that could be obtained by utilizing existing schedulability tests (e.g. AMC-max [5], AMC-rtb [5], SMC [4], FPPS [11], [3]) designed for constrained-deadline task sets. These methods can be used to provide sufficient tests for arbitrary-deadline task sets via the simple expedient of *constraining* any deadline that is greater than the task's period to be equal to that period. In the

figures, these methods are denoted by "(Suff.)" indicating a sufficient test. They are shown using dotted lines, with the same markers and line colors as the equivalent arbitrary-deadline tests. Note that UB-H&L-Arb (Suff.) is an upper bound on the schedulability of all fixed-priority fully-preemptive methods for constrained-deadline task sets.

C. Experiments

In our experiments, the task set utilization was varied from 0.025 to 0.975[1]. For each utilization value, 1000 task sets were generated (100 for weighted schedulability experiments) and the schedulability of those task sets determined using the schedulability tests listed above. The graphs are best viewed on-line in color.

Figure 1 shows the percentage of task sets generated that were deemed schedulable for a system of 20 tasks with the defaults parameters as described in section VIII-A. We observe that AMC-max-Arb outperforms AMC-rtb-Arb by a small but significant margin. The performance of both these tests is relatively close to the theoretical upper bound given by UB-H&L-Arb; closer than the equivalent tests for constrained deadlines (dotted lines) are to their bound. This is due to the fact that longer (arbitrary) deadlines can compensate for the effects of the overload that occurs on a criticality mode switch over a longer time interval.

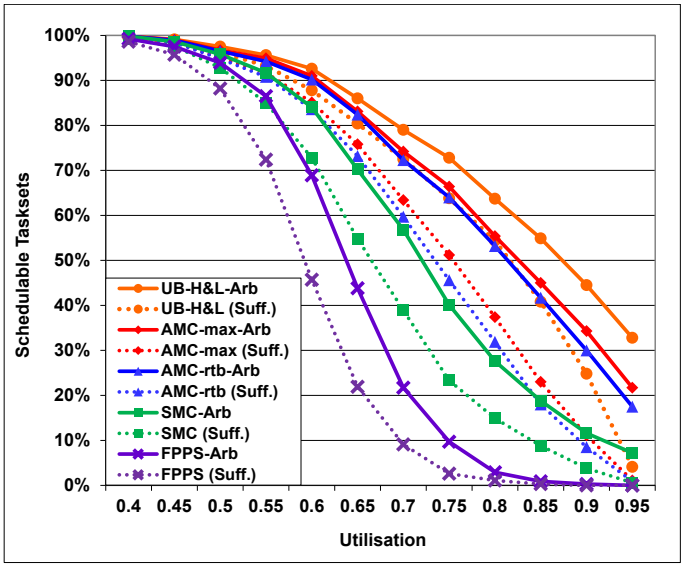

Fig. 1. Percentage of schedulable task sets

Figure 1 also illustrates that the performance of AMC-max-Arb and AMC-rtb-Arb is significantly better than that of SMC-Arb, which in turn is significantly better than FPPS-Arb. Further, each of these methods provides performance that is significantly better than the equivalent constrained-deadline test adapted to provide sufficient analysis for arbitrary-deadline tasks. Stated otherwise, for mixed criticality systems scheduled using AMC, SMC, and FPPS, increasing task deadlines beyond their periods can provide a substantial increase in guaranteed real-time performance when the schedulability tests derived in this paper are employed.

In the following figures we show the weighted schedulability measure [6] for each schedulability test as a function of some other parameter w which is varied. For each value of w, this measure combines results for the task sets generated for all of a set of equally spaced utilization levels (0.025 to 0.975 in steps of 0.025) weighted by utilisation level. (See [6] for further details of the weighted schedulability measure).

Figure 2 shows how the weighted schedulability measure for each schedulability test changes as the range of task deadlines is varied from $[0.25, 0.25]T_i$ to $[0.25, 5.66]T_i$ (each step on the x-axis increases the upper limit of the range by a factor of $\sqrt[4]{2}$, and hence every 4 steps it increases by a factor of 2). Note in each case the deadlines are chosen at random according to a log-uniform distribution. As expected, in all cases schedulability improves as the range of possible deadlines is expanded.

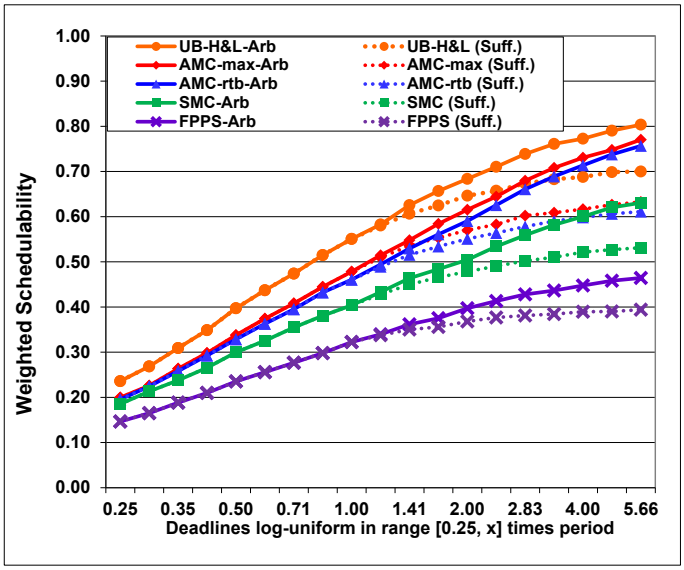

Fig. 2. Varying the range of task deadlines

We observe that while the range is no greater than $[0.25, 1.0]T_i$, then each of the arbitrary-deadline methods provides exactly the same results as its constrained-deadline counterpart (i.e. the solid and the dotted lines precisely overlap). Beyond that point, the arbitrary-deadline analysis confers increasingly superior performance. We note that the relative performance of the various schemes (AMC, SMC, and FPPS) remains broadly similar to that shown in the baseline experiment.

Figure 3 shows how the weighted schedulability measure changes as the range of task periods (ratio of max/min possible task period) is expanded from $10^{0.5} \approx 3$ to $10^4 = 10,000$. (In this experiment, task deadlines D_i were chosen according to the default settings from the range $[0.25, 4.0]T_i$). When the range of tasks periods is small, then the performance advantage of AMC-max-Arb over AMC-rtb-Arb is negated. This happens because when the task periods are similar, AMC-max-Arb counts, via (22), almost all of the jobs of each higher priority HI-criticality task as contributing interference of $C(HI)$. In this case, the analysis effectively reduces to that of AMC-rtb-Arb; all jobs of higher priority HI-criticality tasks contribute $C(HI)$, and the maximum response time occurs when the transition to HI-criticality mode is as late as possible allowing the maximum interference from LO-criticality tasks. As the

[1]Utilization here is computed from the $C(LO)$ values only.

range of task periods increases, so AMC-max-Arb begins to confer an advantage.

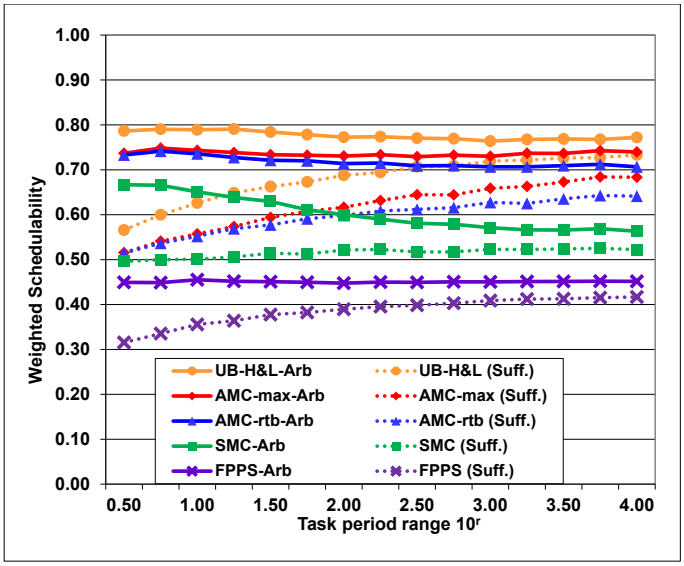

Fig. 3. Varying the range of task periods

The performance of SMC-Arb is substantially better, weighted schedulability ≈ 0.67, when the range of task periods is small, declining to ≈ 0.56 as the range of task periods increases. This effect is due to the influence of optimal priority assignment. If Deadline Monotonic Priority Order is assumed for SMC-Arb, then the result is a nearly horizontal line, with weighted schedulability ≈ 0.47 (this is not shown to avoid cluttering the graph). With arbitrary deadlines and a small range of task periods, the performance of SMC-Arb can be enhanced by the OPA algorithm placing LO-criticality tasks at low priority levels. These tasks are schedulable in LO-criticality mode, assuming interference of $C(LO)$ from higher priority HI-criticality tasks. (Note such an arrangement would not be feasible under FPPS-Arb where LO-criticality tasks also have to be schedulable in HI-criticality mode, where they are subject to interference of $C(HI)$ due to higher priority HI-criticality tasks). As a consequence, HI-criticality tasks can be placed at higher priority levels and thus have fewer higher priority LO-criticality tasks that contribute interference when HI-criticality mode is considered, improving overall schedulability. The amount of flexibility available via priority assignment reduces as the range of task periods becomes larger and the deadlines of the tasks become more widely separated. This reduces the performance of SMC-Arb.

Finally, observe that in the case of the constrained-deadline methods, when the range of task periods is small, then once constrained to the range $[0.25, 1.0]T_i$ all of the task deadlines are fairly similar and hence schedulability is low compared to the situation with a much larger range of periods (and deadlines). This is a well-known property of FPPS. It happens when the total interference from higher priority tasks in a given interval is considerably higher than that implied by their utilization [12]. With longer, arbitrary deadlines or with a larger range of task periods, this excess interference reduces and so schedulability improves.

We also explored the effects of varying the number of tasks, the criticality factor (CF), and the criticality percentage (CP). These results were broadly in line with those reported for the baseline experiment. The graphs are not shown here due to space limitations.

IX. Conclusions

In this paper, we considered the problem of scheduling mixed-criticality systems on a single processor. We studied the Adaptive Mixed Criticality (AMC) scheme that requires additional run-time support, but is able to provide superior schedulability guarantees to Static Mixed Criticality (SMC) or Fixed Priority Preemptive Scheduling (FPPS). Previous studies of AMC have restricted the task model to constrained-deadline tasks. In this paper we lifted this restriction, allowing tasks to have arbitrary deadlines.

There are two main published schedulability tests for the AMC scheme: AMC-rtb and AMC-max. In this paper we extended both tests to allow tasks to have deadlines greater than their periods. Our evaluation demonstrates that the simpler AMC-rtb-Arb form of analysis remains effective for tasks with arbitrary deadlines. The AMC-max-Arb analysis delivers improved schedulability, but at a cost in terms of increased complexity of the method.

Acknowledgements

The research in this paper is partially funded by the ESPRC grant, MCCps (EP/K011626/1). EPSRC Research Data Management: No new primary data was created during this study.

References

[1] S. Asyaban and M. Kargahi. An exact schedulability test for fixed-priority preemptive mixed-criticality real-time systems. *Real-Time Systems*, Aug 2017.

[2] N. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

[3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[4] S. Baruah and A. Burns. Implementing mixed criticality systems in Ada. In A. Romanovsky, editor, *Proc. of Reliable Software Technologies - Ada-Europe 2011*, pages 174–188. Springer, 2011.

[5] S. Baruah, A. Burns, and R. I. Davis. Response-time analysis for mixed criticality systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 34–43, 2011.

[6] A. Bastoni, B. Brandenburg, and J. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *Proc. of Sixth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 33–44, 2010.

[7] E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Journal of Real-Time Systems*, 30(1-2):129–154, 2005.

[8] A. Burns and R. I. Davis. A survey of research into mixed criticality systems. *ACM Computing Surveys*, To appear, 2017.

[9] R. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *Real-Time Systems, Volume 47, Issue 1*, pages 1–40, 2010.

[10] R. I. Davis, L. Cucu-Grosjean, M. Bertogna, and A. Burns. A review of priority assignment in real-time systems. *Journal of systems architecture*, 65:64–82, 2016.

[11] M. Joseph and P. Pandya. Finding response times in a real-time system. *BCS Computer Journal*, 29(5):390–395, 1986.

[12] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–61, 1973.

[13] K. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Journal of Real-Time Systems*, 6(2):133–151, 1994.

[14] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS)*, pages 239–243, 2007.

Probabilistic Analysis of Low-Criticality Execution

Martin Küttler, Michael Roitzsch, Claude-Joachim Hamann
Operating Systems Group
Department of Computer Science
Technische Universität Dresden
01062 Dresden, Germany
Email: {mkuettle,mroi,hamann}@os.inf.tu-dresden.de

Marcus Völp
CritiX: Critical and Extreme Security and Dependability
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg
L-2721 Luxembourg
Email: marcus.voelp@uni.lu

***Abstract*—The mixed-criticality toolbox promises system architects a powerful framework for consolidating real-time tasks with different safety properties on a single computing platform. Thanks to the research efforts in the mixed-criticality field, guarantees provided to the highest criticality level are well understood. However, lower-criticality job execution depends on the condition that all high-criticality jobs complete within their more optimistic low-criticality execution time bounds. Otherwise, no guarantees are made. In this paper, we add to the mixed-criticality toolbox by providing a probabilistic analysis method for low-criticality tasks. While deterministic models reduce task behavior to constant numbers, probabilistic analysis captures varying runtime behavior. We introduce a novel algorithmic approach for probabilistic timing analysis, which we call *symbolic scheduling*. For restricted task sets, we also present an analytical solution. We use this method to calculate per-job success probabilities for low-criticality tasks, in order to quantify, how low-criticality tasks behave in case of high-criticality jobs overrunning their optimistic low-criticality reservation.**

I. Introduction

Mixed-criticality systems [1] promise size, weight and power savings by consolidating safety-critical tasks with different certification requirements on a single computing platform. Examples can be found in many traditional and emerging application scenarios. To formalize such systems and to reason about their behavioral properties, mixed-criticality was invented. It allows designers to rank tasks by criticality levels, which expresses the confidence in task parameters such as worst-case execution times and also indicates, which tasks to drop in case a subset of these task parameters are violated at runtime. Highly critical tasks with high parameter confidence can then be isolated from lower-criticality tasks with relaxed parameter confidence. System designers can use this formalism to meet two otherwise competing goals: criticality levels provide the *separation* needed for safe operation, while the admission of low-criticality tasks according to more optimistic task parameters provides the *resource sharing* needed for efficient operation.

A large body of research has explored many aspects of the mixed-criticality toolbox [2]. In this paper, we contribute new analysis results for the following mixed-criticality scheduling discipline: When a high-criticality job overruns its more optimistic low-criticality execution time bound, all low-criticality tasks drop in priority, so any task with higher criticality takes precedence. Demoting the priority of low-criticality tasks constitutes a straightforward extension of the classical Adaptive Mixed-Criticality (AMC) [3] scheduling discipline, which drops low-criticality jobs altogether after an execution time overrun of a high-criticality job.

Classical mixed-criticality scheduling is based on deterministic worst-case assumptions, which are pessimistic, because at runtime job parameters are rarely constant, but follow a distribution. We present a probabilistic analysis method to capture such varying job behavior. Our task model expresses job execution using a pWCET distribution. We describe the task model in Section III.

In this paper, we make the following contributions:

1) We present an algorithmic approach for probabilistic timing analysis of per-job behavior in mixed-criticality systems, which we call *symbolic scheduling* (Section IV). For restricted task sets with criticality-monotonic priority assignment and harmonic periods, we also present an analytical solution (Section V).
2) We use this analysis to quantify how low-criticality tasks behave in case of high-criticality jobs overrunning their optimistic low-criticality reservation. We calculate probabilities p for jobs of low-criticality tasks meeting their deadline when the system operates in high criticality mode.

In Section VI, we evaluate our analysis using randomly generated task sets. We show that our probabilistic analysis can quantify the low-criticality task execution. These success probabilities can, for example, be used for a more permissive admission test that requires only a given percentage q of jobs to succeed in low criticality mode.

II. Related Work

Lehoczky [4] was first to characterize execution times as random variables and to describe them through probability density functions. However, as realized by Griffin and Burns [5], modern processor architectures often violate the independence assumptions required for scheduling based on probabilistic execution times to remain mathematically tractable. Recent works on probabilistic worst-case execution times (pWCET) [6], [7], [8], [9], [10], [11], [12] thus describe the confidence in WCET estimates of a task as the exceedance probability derived from the generalized extreme-value distribution of observed maxima.

Different methods to derive probabilistic representations of execution time have been explored. For example, Yue et al. [13] present a technique based on random sampling for determining

pWCETs. Iverson et al. [14] suggest a purely statistical analysis, whereas David and Puaut [15] propose a combined static and measurement-based analysis.

Statistical and probabilistic techniques have been used for real-time analysis in the past. Atlas and Bestavros [16] calculate task success rates when exact execution times are known at the release instant. Guo et al. [17] perform admission tests under given failure probabilities. Hamann et al. [18], [19] extend imprecise computations [20] to derive budgets for optional parts that guarantee a certain completion probability.

Recent work on probabilistic mixed-criticality analysis by Maxim et al. [21] calculates worst-case response times (WCRT) for static and adaptive mixed-criticality scheduling based on critical instant analysis. However, WCRT alone is an inadequate quality metric for a mixed-criticality schedule. After a criticality switch to high-criticality mode, the low-criticality job following the critical instant may never execute, whereas all following jobs of the same task could always be successful. Therefore, a success probability based on WCRT is arbitrarily pessimistic.

Our work extends this analysis by calculating success probabilities for every job individually and aggregating them to a per-task value that is less pessimistic. We propose *symbolic scheduling,* which can be viewed as a specialized form of probabilistic model checking.

III. Task Model

In his seminal work, Vestal proposed to describe tasks with different certification requirements through vectors of increasingly pessimistic scheduling parameters [1]. Admission and scheduling must ensure that a higher-criticality task failing to meet the more optimistic requirements from a lower certification level can still meet its deadline when it adheres to the more pessimistic parameters at its high certification level. Baruah et al. coined the term certification-cognizant scheduling [22] for this interpretation of the mixed-criticality framework. Our paper follows this interpretation.

In the standard deterministic model, a task $\tau = (T, D, L, C(\ell))$ is assumed strictly periodic with period T, a relative deadline D, a criticality level L and a worst-case execution time (WCET) $C(\ell)$ for each criticality level $\ell \leqslant L$. These WCETs must satisfy $C(\ell_1) \leqslant C(\ell_2)$ for $\ell_1 \leqslant \ell_2$. Our analysis does not restrict the number of criticality levels or the relation of T and D, so $D \geqslant T$ is possible. To simplify the presentation, we only discuss the dual criticality case, with the two criticality levels named LO and HI. Other research also considers criticality-dependent interrelease times [23] or deadlines [24]. We limit ourselves to criticality-dependent execution times.

We extend the standard task model to a probabilistic one similar to Maxim et al. [21]. In addition to the above parameters, each task is described by a probabilistic worst case execution time (pWCET) X. The CDF of this random variable describes the probability for a job to not exceed a given execution time bound, which allows us to compute probabilistic guarantees for low-criticality jobs. We write $X = [2 : 0.8, 5 : 0.2]$, e.g., meaning that with a probability of 0.8 the job will have finished executing until 2 time units, and with a probability of 0.2 it may exceed 2 time units and take up to 5 time units.

We assume an active enforcement of execution times by the runtime system. Whenever an execution time bound or deadline is reached, the system aborts jobs or drops them in priority. Whenever a LO-criticality job executes beyond $C(LO)$, it is aborted. $C(HI)$ is therefore not meaningful for LO-jobs. Whenever a HI-criticality job exceeds its $C(LO)$ execution time, the system switches to HI-criticality mode. We call this situation *criticality miss.* As part of this mode switch, the priorities of all current and future LO-jobs are changed such that all HI jobs dominate all LO-jobs. We allow switching back from HI mode to LO mode only at a simultaneous release instant at the beginning of a hyperperiod. Protocols allowing earlier recovery have been presented [25], but are not considered here.

IV. Symbolic Scheduling

We propose the concept of *symbolic scheduling,* which we present here and which we have implemented[1]. Symbolic scheduling draws on ideas from actual runtime scheduling, in that it tracks runs of jobs along a time axis to figure out which meet their deadline. Unlike a scheduler, though, it does not observe concrete execution times, but it keeps track of possible executions and their probabilities.

Given a set of tasks as described in Section III, there is a conceptually simple but computationally expensive way to analyze the behavior of each job: Try every possible combination of execution times, and keep track of the respective probabilities. This leads to a tree, where each path from the root to a leaf is a possible execution trace of the system, and each node branches into as many subtrees as the respective job has possible values for its execution. The obvious disadvantage is that this tree will grow huge, and that many paths through it will be equivalent for practical purposes. Such equivalent paths may differ in execution times, but agree in the succession of jobs and them finishing before their deadline. Symbolic scheduling takes advantage of these equivalences by trying to merge branches that only differ in timings, but not in the jobs' order and success. More precisely, it does not branch unless there is an immediate and important difference. However, depending on taskset parameters our current implementation can take multiple hours to produce a result.

A. Example

Consider the following example with two tasks:

$$\tau_1 : T = D = 8, X = [2 : 0.8, 5 : 0.2]$$
$$\tau_2 : T = D = 16, X = [1 : 0.6, 11 : 0.4]$$

We ignore criticality for now, so we can ignore L and C. The two Jobs of task τ_1 that run in the first hyperperiod are called J_{11} and J_{12}, the job of τ_2 is called J_{21}.

Assuming EDF scheduling, the symbolic scheduler starts at time $t = 0$ and first select job J_{11}. Since it is a job of the task of highest priority, it will run until completion at either time

[1] https://github.com/mkuettler/symbolic-scheduler

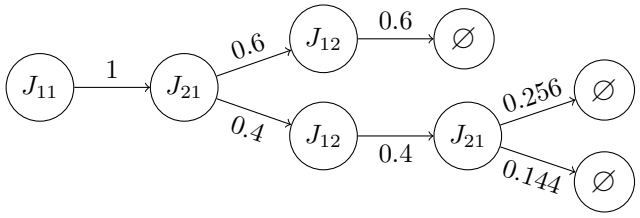

Figure 1. Event tree for the example in IV-A

2 or 5. Note that these two cases only differ in their time of occurrence and probability — the ready jobs and completed jobs are exactly the same. Thus we do not want to distinguish these cases as different timelines, because the full combinatorial tree that would ensue is prohibitively huge. Instead, the symbolic scheduler represents the current time using (partial) distributions: after scheduling J_{11} we are at time $t = [2 : 0.8, 5 : 0.2]$. These distributions describe the probability of the currently investigated situation. Note that the accumulated probability of such a *partial distribution* can be less than 1.

Next, job J_{21} runs. The total runtime of these first two jobs can be 3, 6, 13, or 16. Only the first two results are possible times for J_{21} to complete, because J_{12} becomes ready at time 8 and will interrupt J_{21} if it is still running. Thus, the symbolic scheduler needs to branch into two different possible timelines.

Like before, we do not want to branch needlessly. We only need to distinguish whether J_{21} finishes before J_{12} arrives. If it does — i.e., when J_{21} executes only for 1 time unit — we are at time $t = [3 : 0.48, 6 : 0.12]$, with J_{12} as the only job left. The scheduler will wait until the arrival of J_{12} at 8, which leaves us at $t = [8 : 0.6]$, since the total probability is $0.6 = 0.48 + 0.12$. Then J_{12} can run, and finishes at $t = [10 : 0.48, 13 : 0.12]$. This trace corresponds to the topmost branches in Figure 1.

If J_{21} does not finish before J_{12} arrives, it is interrupted at $t = [8 : 0.4]$, because J_{12} has higher priority. But J_{21} is not done yet and still remains in the list of ready jobs. It ran for $[6 : 0.8, 3 : 0.2]$ time units already, so the remaining time is $[5 : 0.8, 8 : 0.2]$. Now J_{12} runs to completion at $t = [10 : 0.32, 13 : 0.08]$. After that the remaining part of J_{21} is scheduled, and runs until $[15 : 0.256, 18 : 0.128, 21 : 0.016]$. But since the deadline of J_{21} is at 16, the job will finish successfully with a probability of 0.256, and miss its deadline with probability $0.128 + 0.016 = 0.144$. This simplified example illustrates the main concept of symbolic scheduling. To give a formal description we need to introduce some notation.

B. Notation

Let d, d_1, d_2 be potentially partial distributions, and x be a number.

- $d_1 + d_2$ denotes the convolution of d_1 and d_2.
- $d \leftarrow x$ is the part of d that lies to the left of x, including x. Conversely, $d \rightarrow x$ is the part of d that lies to the right of x, excluding x.
- $\mathrm{sum}(d)$ denotes the total probability of all values of d. Thus $\mathrm{sum}(d) = \mathrm{sum}(d \leftarrow x) + \mathrm{sum}(d \rightarrow x)$ for all d and x.
- $d_1 \cup d_2$ is defined to be the distribution that contains all the points in d_1 and d_2, with their respective probabilities. Hence $\mathrm{sum}(d_1 \cup d_2) = \mathrm{sum}(d_1) + \mathrm{sum}(d_2)$, which must be $\leqslant 1$ for this operation to make sense.
- $d \rightarrowtail x := (d \rightarrow x) \cup [x : \mathrm{sum}(d \leftarrow x)]$

With this notation, we can formally describe symbolic scheduling for the simplified scenario where criticality misses do not change job priorities, i.e., priorities are criticality monotonic. The general formulation is more complex and due to spatial constraints, we refer the reader to our technical report [26].

Algorithm 1 Symbolic Scheduling without criticality misses

```
 1  function sym_sched(t, jobs) {
 2      J = next_job(jobs)
 3      if J is None: return
 4      t = t ↣ J.release
 5      s = next_sched_event(jobs)
 6      t1 = (t + J.X) ↰ s
 7      t2 = (t + J.X) ↱ s
 8      if not empty(t1) {
 9          J.success += sum(t1)
10          new_jobs = jobs.remove(J)
11          sym_sched(t1, new_jobs)
12      }
13      if not empty(t2) {
14          diff = sum(t2)-sum(t ↱ s)
15          t_next = (t ↱ s) ∪ [s: diff]
16          new_jobs = jobs.remove(J)
17          if s ≠ J.deadline {
18              elapsed = (s-time) ↣ 0
19              J.X = (J.X - elapsed) ↱ 0
20              J.X = normalize(J.X)
21              new_jobs.insert(J)
22          }
23          sym_sched(t_next, new_jobs)
24      }
25  }
```

C. Simplified Formal Description

Let t be the current time distribution, and J the next job, i.e., the ready job with the highest priority. Let s be the time of the next scheduling event, i.e., either the deadline of J or the release of a job with higher priority. Note that s is not a distribution, but a scalar value. To calculate the next time point t_{next}, there are two cases to consider:

- J finishes before s: $t_{\text{next}} = (t + X(J)) \leftarrow s$.
- J does not finish before s. Intuitively, t_{next} should be $[s : \mathrm{sum}((t + X(J)) \rightarrow s)]$, like in the example above. But that only works if $t \leqslant s$ (i.e. $t \rightarrow s$ is empty), otherwise the next time point would lie in the past. Branching into two different timelines would be an option, but for performance reasons we want to reduce branches. Instead, we can handle this case as follows:

$$t_{\text{next}} = t \rightarrow s \cup \left[s : \mathrm{sum}\big((t + X(J)) \rightarrow s\big) - \mathrm{sum}(t \rightarrow s)\right].$$

In this case J is not done, so unless s is its deadline it must be kept in the list of ready jobs. But to account for the time it did run, its remaining execution time must be set to $(X(J) - ((s - t) \rightarrowtail 0)) \rightarrow 0$, normalized to total probability of 1.

The algorithm is shown in Algorithm 1. Starting at $t = 0$, it selects the ready job with the highest priority, and, for each of

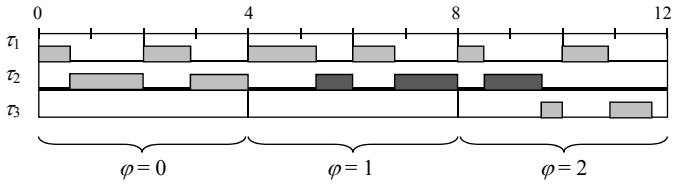

Figure 2. Job Phases and Remaining Execution Time

the two cases, updates the time and ready list, and repeats. In a general mixed-critically system there is one additional case to consider: A job may trigger a critically miss, i.e., it may overrun its $C(LO)$, thus causing the system to switch to HI-mode. This situation can be covered by a third branch to be followed and analyzed. We present more details in the technical report [26].

V. Analytical Solution

For determining per-job success probabilities analytically, we first restrict the task model to allow reasonable calculation effort and formula complexity. We consider systems with implicit deadlines and harmonic periods, consisting of m HI-tasks τ_i and n LO-tasks τ_{m+j}. The scheduling algorithm uses criticality-monotonic priority assignment: Tasks are assigned a static priority, such that all HI-tasks dominate all LO-tasks, thus forming two priority bands. Within each band, rate monotonic priority assignment is used. A criticality miss is thus inconsequential. The typical approach to switch back from HI to LO-mode at processor idle time results in LO-jobs always being executed as soon as the processor is done executing HI jobs.

Due to limited space, we again only sketch the central case, which would entail further special cases described in the technical report [26]. The central case is characterized by all LO-tasks having period lengths at most the length of the hyperperiod of the HI-tasks, hence $T_{m+n} \leqslant T_m$. Clearly, the success probability p_j of a job of task τ_{m+j} depends on its position within this hyperperiod. We call this position the job's phase φ, counting from 0 as shown in Figure 2 for two HI-tasks τ_1, τ_2 and one LO-task τ_3, as well as $T_1 = 2$, $T_2 = 12$, $T_3 = 4$.

Crucial for determining p_j is the calculation of the time demanded by all HI-tasks within one phase as well as the remaining execution times of those HI jobs, whose period exceeds that of the currently considered LO job. According to Figure 2, those HI jobs start earlier, but extend into the phase of the considered LO job. Any such remaining execution time is also a random variable, so a concrete realization may already finish in phase $\varphi = 1$.

Analyzing the highest priority LO-task τ_{m+1} is comparably simple: The calculation is performed in *schedule slices* of length T_{m+1}. Unfortunately, applying the same idea to all lower priority LO-tasks is dangerously misleading, because remaining execution times or remaining processor capacity across multiple phases are not stochastically independent. A formal description must be based on conditional probabilities, which can be calculated by distinction of multiple cases. Similar to the symbolic scheduler, we employ event trees to solve this problem. We continue in two steps.

In step one, we construct the event tree $\tilde{A}_0$ describing all HI-tasks in schedule slices of length T_{m+1}. Nodes of this tree are random variables A with the intuitive meaning 'total execution time of all HI-tasks minus period length'. Formally, the semantics is as follows: For $A = a$ with $a \geqslant 0$, the highest priority LO-task receives a processor capacity of a within the respective phase and no remaining execution time of HI-jobs occurs. In case $a < 0$, the processor is completely occupied by the HI-tasks, which further contribute a remaining execution time of $-a$. Edges within the tree are annotated with the probabilities of the respective case.

We denote the resulting random variables $A_{0\varphi}^{r_1 \dots r_\varphi}$, with φ being the phase of τ_{m+1} and $r_1, \dots, r_\varphi$ the remaining execution times occurring up until this phase. We call this sequence the history H of $A_{0\varphi}$. For the HI-tasks within phase $\varphi = 0$ of τ_{m+1} we have $A_{00} = T_{m+1} - \sum_{i=1}^{n} X_i$, which determines the root node of the event tree. For the general case we observe that a remaining execution time $r < T_{m+1}$ leaves us with a remaining processor capacity of $T_{m+1} - r$. This amount is reduced by the time demand of all HI-jobs that are released at the beginning of this phase. This conclusion also applies to $r \geqslant T_{m+1}$, which can anyways lead to further remaining execution time which is carried onward into the next phase.

Lemma. *After execution of all HI-jobs, the highest priority LO-task receives processor capacity in phase $\varphi = 0, \dots, {}^{T_m}/_{T_{m+1}}$ depending on remaining execution times $r_1, \dots r_{\varphi-1}, r$ of the preceding phases:*

$$A_{0\varphi}^{r_1 \dots r_{\varphi-1} r} = A_{0\varphi}^{r_1 \dots r_{\varphi-1} 0} - r \quad \textit{with}$$
$$A_{0\varphi}^{r_1 \dots r_{\varphi-1} 0} = T_{m+1} - \sum_{i:\, \frac{\varphi T_{m+1}}{T_i} \in \mathbb{N}} X_i$$

In the second step, we calculate the corresponding random variables for all other LO-tasks, resulting in success probabilities for each LO-job. We begin with the case of A_{00} assuming a negative value, meaning that the CPU is already fully loaded with an available capacity of 0. The first job of LO-task τ_{m+1} is therefore not running, thus having success probability $p_{10} = 0$. Remaining execution times represented by A_{00} remain unchanged. In the other case, the value a of $A_{00} - X_{m+1}$ describes, whether the LO-job is successful ($a \geqslant 0$) or unsuccessful ($a < 0$). Unsuccessful execution does not add to the value of p_{10} and no more processor time is available. Hence, those values are replaced with 0. Because LO-jobs are discarded at the end of their period, they do not contribute remaining execution time and the event tree $\tilde{A}_0$ remains unchanged. To consider cases with remaining execution being carried into the current phase, the event tree needs to be transformed for the highest-priority LO-task and aggregation for all lower-priority LO-tasks as well as modification of node random variables are required. We summarize those consequences in the following:

- For a discrete random variable Z with negative values, we form a partial distribution: Let $Z(0)$ be the partial

distribution containing all non-negative values of Z. Further, let $Z(r)$ for $r > 0$ be the partial distribution solely containing the value r if Z contains the value $-r$. In both cases, the respective probabilities are copied from Z unchanged.

- The tree $\tilde{A}_0$ is transformed into the tree A_0 by splitting the root node A_{00} into nodes $A_{00}(r)$, analogously for all non-root nodes.
- For the highest-priority LO-task τ_{m+1}, partial distributions are calculated:

$$B^H_{1\varphi}(r) = A^H_{0\varphi}(r) - X_{m+1}$$

Summation leads to the success probability of a LO-job in phase φ. Now $B^H_{1\varphi}(r)$ is transformed into the final partial distribution $A^H_{1\varphi}(r)$ by replacing negative values in $B^H_{1\varphi}(r)$ with 0 and accumulating the accompanying probabilities.

We end up with a tree A_1, which is structurally identical to A_0, but the distributions kept at the nodes have changed. For every phase of τ_{m+1} we receive a probability based on the corresponding history H. Determining the success probabilities $p_{1\varphi}$ requires weighing those values with the product of the probabilities along the path from the root node to the considered leaf node.

For all further LO-tasks, we can continue in the same manner. The stochastic independence of execution times in successive phases is guaranteed by the separation into distinct cases in the event tree. Therefore, connected random variables can be added, causing related phases of A_1 to be aggregated, which leads to a new tree A_2.

For a generalized formulation, let $q^H_{j\varphi}(r)$ be the probability along the path from the root of the tree A_j to the considered leaf node and let the parameter r of $A^H_{j\varphi}(r)$ be called the state of the job. Then:

Proposition. *In case $T_{m+n} \leqslant T_m$, the success probability $p^H_{j\varphi}(r)$ of a job from LO-task τ_{m+j} in state r within phase φ and with history H for $\varphi = 0, \ldots, {}^{T_m}\!/_{T_{m+j}} - 1$, $j = 1, \ldots, n$ is*

$$p^H_{j\varphi}(r) = \Pr(B^H_{j\varphi}(r) \geqslant 0)$$

with

$$B^H_{1\varphi}(r) = A^H_{0\varphi}(r) - X_{m+1},$$

$$B^H_{j\varphi}(r) = \sum_{k=0}^{q_j} A^H_{j,q_j\varphi+k}(r) - X_{m+j},$$

$$\text{for } q_j = {}^{T_{m+j}}\!/_{T_{m+j-1}}.$$

The final success probability of a job from LO-task τ_{m+j} in phase φ follows:

$$p_{j\varphi} = \sum_{H,r} q^H_{j\varphi}(r) \cdot p^H_{j\varphi}(r).$$

Further,

$$A^H_{j\varphi}(r) = \sum_{k=0}^{q_j} A^H_{j,q_j\varphi+k}(r) \dot{-} X_{m+j}$$

(operator $\dot{-}$ denotes non-negative subtraction).

An aggregated success probability p_j for LO-task τ_{m+j} can be given as

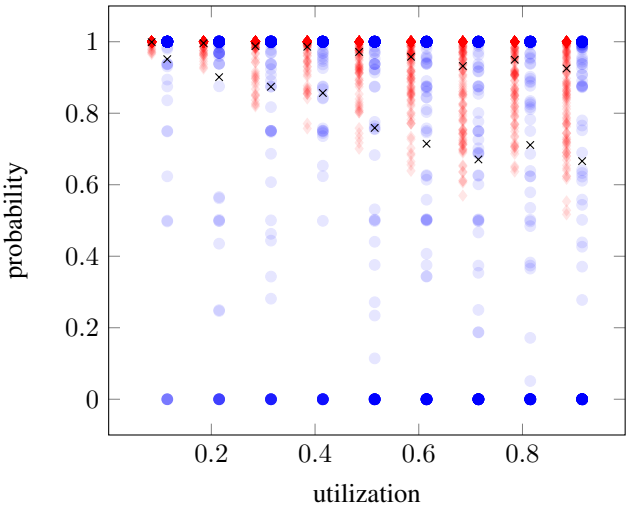

Figure 3. Average job success probability (red) and first job success probability (blue) for RMS in bands

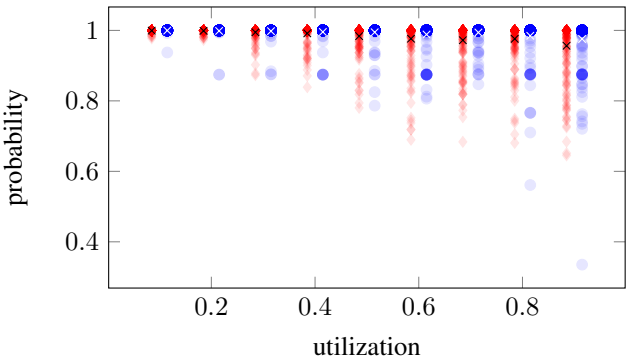

Figure 4. Average job success probability (red) and first job success probability (blue) for EDF

$$p_j = \frac{1}{{}^{T_m}\!/_{T_{m+j}}} \sum_{\varphi=0}^{{}^{T_m}\!/_{T_{m+j}}-1} p_{j\varphi}$$

using the assumption that phases occur uniformly distributed.

VI. Evaluation

In this section we show preliminary results of our per-job analysis of LO-tasks in mixed criticality systems. We determine success probabilities of LO-job execution. We compare our analysis results to critical instant response time analysis.

A. Task Generation

We chose $D = T$ from a list of 7 values that roughly follow a log-uniform distribution between 15 and 1000. $C(LO)$ is determined from a utilization generated by the UUnifast algorithm [27]. We vary the total utilization (see below). Tasksets have two criticality levels LO and HI, the chance for a task to have HI criticality is 50%. In this case $C(HI) = 1.6\,C(LO)$.

The pWCET distribution of each task can take values between $0.6\,C(LO)$ and $C(LO)$ (for LO-tasks) or $C(HI)$ (for HI-tasks). Values are uniformly spaced with a distance of $0.2\,C(LO)$, thus there are 3 values in LO-task distributions

and 6 values in HI-task distributions. The corresponding probabilities start at 0.5 for $0.6\,C(LO)$ and are halved at each step except the last (so that the sum is 1). This way the pWCET distributions approximates an exponential tail distribution.

B. Results

For each utilization in $0.1, 0.2, \ldots, 0.9$ we generated 100 tasksets. They were scheduled twice, once with criticality-monotonic priorities and rate-monotonic ordering within each criticality band (Figure 3), and once with bands determined by the system criticality level (jobs of at least that criticality in the upper, all other in the lower band) and EDF priorities within the bands (Figure 4). In both figures, red marks denote aggregate task success probabilities, i.e., the average success probability across all jobs of a task, and blue marks denote the success probability at the critical instant (at time 0 in these examples). Marks are transparent to illustrate their distribution, crosses show the average within each column. Only LO jobs were considered in both plots, as high jobs must always finish in a valid schedule.

In Figure 3, where priorities are static, critical instant probabilities are a — sometimes very pessimistic — lower bound of the average probability. When priorities depend on the system criticality however, as in Figure 4, the critical instant with standard criticality does not provide a lower bound.

VII. Conclusion

In this work, we propose a new method for probabilistic analysis of low-criticality tasks. We implement our approach using *symbolic scheduling*. For restricted task sets, we also present an analytical solution. We use our analysis to show first results on success probabilities of low-criticality tasks after a criticality miss. We believe our analysis provides a useful new tool to designers of mixed-criticality systems.

Acknowledgments

This research was supported in part by the German Research Council (DFG) through the Cluster of Excellence *Center for Advancing Electronics Dresden* and through the German priority program 1648 *Software for Exascale Computing* via the research project *FFMK: A Fast and Fault-Tolerant Microkernel,* as well as by the Fonds National de la Recherche Luxembourg (FNR) through PEARL grant FNR/P14/8149128.

References

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

[2] A. Burns and R. I. Davis, "Mixed-criticality systems: A review," University of York, York, UK, Tech. Rep. 5th Edition, February 2015. [Online]. Available: http://www-users.cs.york.ac.uk/burns/review.pdf

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

[4] J. P. Lehoczky, "Real-time queueing theory," in *17th IEEE Real-Time Systems Symposium (RTSS)*. IEEE, December 1996, pp. 186–195.

[5] D. Griffin and A. Burns, "Realism in statistical analysis of worst case execution times," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*. OASIcs, July 2010, pp. 44–53.

[6] L. Cucu-Grosjean, "Independence: A misunderstood property of and for probabilistic real-time systems," in *Real-Time Systems: The Past, the Present and the Future*, N. Audsley and S. Baruah, Eds., March 2013, pp. 29–37.

[7] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiones, and F. J. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs," in *24th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, July 2012, pp. 91–101.

[8] R. I. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean, "Analysis of probabilistic cache related pre-emption delays," in *25th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, July 2013, pp. 168–179.

[9] S. Edgar and A. Burns, "Statistical analysis of WCET for scheduling," in *22nd IEEE Real-Time Systems Symposium (RTSS)*. IEEE, December 2001, pp. 215–224.

[10] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real-time systems," in *23rd IEEE Real-Time Systems Symposium (RTSS)*. IEEE, December 2002, pp. 279–288.

[11] S. Altmeyer, L. Cucu-Grosjean, and R. I. Davis, "Static probabilistic timing analysis for real-time systems using random replacement caches," *Real-Time Systems*, vol. 51, no. 1, pp. 77–123, January 2015.

[12] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim, "PROARTIS: Probabilistically analyzable real-time systems," *ACM Transactions on Embedded Computing Systems*, vol. 12, no. 2s, pp. 94:1–94:26, May 2013.

[13] Y. Lu, T. Nolte, I. Bate, and L. Cucu-Grosjean, "A new way about using statistical analysis of worst-case execution times," *SIGBED Review*, vol. 8, no. 3, pp. 11–14, September 2011.

[14] M. A. Iverson, F. Özgüner, and L. C. Potter, "Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment," *IEEE Transactions on Computers*, vol. 48, pp. 1374–1379, December 1999.

[15] L. David and I. Puaut, "Static determination of probabilistic execution times," in *16th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, June 2004, pp. 223–230.

[16] A. K. Atlas and A. Bestavros, "Statistical rate monotonic scheduling," in *19th IEEE Real-Time Systems Symposium (RTSS)*. IEEE, December 1998, pp. 123–132.

[17] Z. Guo, L. Santinelli, and K. Yang, "EDF schedulability analysis on mixed-criticality systems with permitted failure probability," in *21st International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, August 2015, pp. 187–196.

[18] C.-J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig, "Quality-assuring scheduling: Using stochastic behavior to improve resource utilization," in *22nd IEEE Real-Time Systems Symposium (RTSS)*. IEEE, December 2001, pp. 119–128.

[19] C.-J. Hamann, L. Reuther, J. Wolter, and H. Härtig, "Quality-assuring scheduling," TU Dresden, Dresden, Germany, Tech. Rep. TUD-FI06-09, December 2006.

[20] K.-J. Lin, S. Natarajan, and J. W. S. Liu, "Imprecise results: Utilizing partial comptuations in real-time systems," in *8th IEEE Real-Time Systems Symposium (RTSS)*. IEEE, December 1987, pp. 210–217.

[21] D. Maxim, R. I. Davis, L. Cucu-Grosjean, and A. Easwaran, "Probabilistic analysis for mixed criticality scheduling with SMC and AMC," in *4th International Workshop on Mixed Criticality Systems (WMC)*, November 2016.

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

[23] S. Baruah, "Certification-cognizant scheduling of tasks with pessimistic frequency specification," in *7th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE, June 2012, pp. 31–38.

[24] P. Huang, G. Giannopoulou, N. Stoimenov, and L. Thiele, "Service adaptions for mixed-criticality systems," in *19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, January 2014, pp. 125–130.

[25] I. Bate, A. Burns, and R. I. Davis, "A bailout protocol for mixed criticality systems," in *27th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, July 2015, pp. 259–268.

[26] M. Küttler, M. Roitzsch, C.-J. Hamann, and M. Völp, "Probabilistic analysis of low-criticality execution," TU Dresden, Dresden, Germany, Tech. Rep. TUD-FI17-02, November 2017. [Online]. Available: https://os.inf.tu-dresden.de/papers_ps/wmc2017-probmcs-tr.pdf

[27] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, p. 129–154, May 2005.

Revisiting the Computational Complexity of Mixed-Critical Scheduling

Rany Kahil[1], Peter Poplavko[2], Dario Socci[2], Saddek Bensalem[1]
1. Université Grenoble Alpes (UGA), VERIMAG, F-38000 Grenoble, France
2. Mentor® A Siemens Business, F-38334 Inovallée Montbonnot, France

***Abstract*— In this paper we revisit, refute, and give missing proofs for some previous results on mixed critical scheduling. We find a counter-example to the proof that this optimisation problem belongs to the class NP, which reopens the question on its computational complexity upper bound. Further, we consider a restricted problem formulation, the 'fixed priority per mode' (FPM) scheduling, to show that it is in NP. To do this, one has to demonstrate that FPM is sustainable. We prove (in the extended version, [1]) that is true in the single-processor case. We also show that FPM is not sustainable in the multiprocessor case.**

I. Introduction

When defining and solving optimisation problems care should be taken to provide a rigorous procedure to test the correctness of solutions. For mixed critical scheduling in general it has been believed proven in [2] that the testing can be done by a "canonical" algorithm in time polynomial on the number of jobs. In [2] this has served a proof that the computational complexity of this problem is *at most* of class NP (for a fixed number of criticality levels). By reduction from an NP-Hard problem to a mixed criticality problem, it was shown that the problem is also *at least* NP-Hard.

However, in this paper we present a counterexample that refutes the proof in [2] of polynomial complexity of the "canonical" algorithm. Consequently, the *upper bound* cannot be anymore considered proven, and the problem may, in fact have a complexity beyond the class NP. The question of complexity upper bound is thus re-opened.

We also have considered a restricted problem formulation where the polynomial complexity of the canonical algorithm is true by construction. In this case, the above mentioned refutation poses no problem to NP complexity claim. The considered restriction is well-known fixed-priority per mode (FPM) scheduling policy. For this problem the NP-Hard complexity lower bound from [2] still holds, and it would be useful to establish the class NP as an upper bound.

It turns out that there is another obstacle in doing it, unrelated to the refuted polynomial complexity proof. One can reapply the proof NP complexity given in [2] to a concrete scheduling policy only if that policy is *sustainable*, in a generalized mixed-critical sense. In the previous work, *e.g.,* [3], it was taken for granted that FPM is sustainable, but a closer study has revealed that it can only be the case for single processor but not multiple processors. Even for single processor case, currently we prove the result only for dual-critical instances. Therefore, so far only dual-critical single-processor FPM policy can be demonstrated to be in class NP, the other cases is an open problem.

II. Problem Formulation

Since our topic is problem complexity results, in this paper we will focus on simplest – dual-criticality – problem, as in this particular case it is the easiest to understand and revisit these results. The dual-criticality systems are systems that have only two levels of criticality, the high level, being denoted as 'HI', and the low (normal) level, denoted as 'LO'. Every job gets a pair of WCET values: the LO WCET and the HI WCET. The former one is for normal safety assurance, used to assess the sharing of processor with the LO jobs, and the other one, a higher value, is used to ensure certification.

A *job* J_j is characterized by a 5-tuple $J_j = (j, A_j, D_j, \chi_j, C_j)$, where:

- $j \in \mathbb{N}_+$ is a unique index
- $A_j \in \mathbb{N}$ is the arrival time, $A_j \geq 0$
- $D_j \in \mathbb{N}$ is the deadline, $D_j \geq A_j$
- $\chi_j \in \{\text{LO}, \text{HI}\}$ is the job's criticality level
- $C_j \in \mathbb{N}_+^2$ is a vector $(C_j(\text{LO}), C_j(\text{HI}))$ where $C_j(\chi)$ is the WCET at criticality level χ.

The index j is technically necessary to distinguish between jobs with the same parameters. The timing parameters A_j, D_j, C_j are integers that correspond to time resolution units (*e.g.,* clock cycles). We assume that [2]: $C_j(\text{LO}) \leq C_j(\text{HI})$ The latter makes sense, since $C_j(\text{HI})$ is a more pessimistic estimation of the WCET than $C_j(\text{LO})$. We also assume that the LO jobs are forced to terminate after $C_j(\text{LO})$ time units of execution, so: $(\chi_j = \text{LO}) \Rightarrow C_j(\text{LO}) = C_j(\text{HI})$.

An *instance* of the scheduling problem is a set of jobs $\mathbf{J}$. A *scenario* of an instance $\mathbf{J}$ is a vector of execution times of all jobs: $c = (c_1, c_2, \ldots, c_K)$, where K is the number of jobs. We only consider scenarios where no c_j exceeds $C_j(\text{HI})$. The *criticality of scenario* $c = (c_1, c_2, \ldots, c_K)$ is LO if $c_j \leq C_j(\text{LO}), \quad \forall j \in [1, K]$, is HI otherwise. A scenario c is *basic* if:

$$\forall j = 1, \ldots, K \quad c_j = C_j(\text{LO}) \vee c_j = C_j(\text{HI})$$

A *schedule* $\mathcal{S}$ of a given scenario c is a mapping: $\mathcal{S} : T \mapsto \widehat{\mathbf{J}}_m$, where T is the physical time and $\widehat{\mathbf{J}}_m$ is the family of subsets of $\mathbf{J}$ that contains all subsets $\mathbf{J}'$ of $\mathbf{J}$ such that $|\mathbf{J}'| \leq m$, where m is the number of processors. Every job J_j should start at time A_j or later and run for no more than c_j time units. We assume that the schedule is *preemptive* and that job migration is possible, *i.e.,* that any job run can be interrupted and resumed later on the same or different processor. Note that in this definition we do not include the mapping of jobs to processors, but a valid mapping, if needed, can be easily

obtained from a simulation which assumes that a job can be scheduled at any available processor at any time.

A job J is said to be *ready* at time t if at that time or earlier it has already arrived and has not yet terminated. The online state of a run-time scheduler at every time instance consists of the set of terminated jobs, the set of *ready jobs*, the remaining workload of ready jobs, *i.e.,* for how much they should still execute in future, and the current *criticality mode*, χ_{mode}, initialized as $\chi_{mode} = \text{LO}$ and switched to 'HI' as soon as a HI job exceeds $C_j(\text{LO})$. A scheduling policy is *correct* for the given problem instance if the following conditions are respected in any possible scenario:

Condition 1: If all jobs run at most for their LO WCET, then both critical (HI) and non-critical (LO) jobs must terminate before their deadline.

Condition 2: If at least one job runs for more than its LO WCET, then all critical (HI) jobs must terminate before their deadline, whereas non-critical (LO) jobs may be even dropped.

Based on the online state, a *scheduling policy* deterministically decides which ready jobs are scheduled at every time instant on m processors. A policy is said to be *work-conserving* if it never idles the processor if there is pending workload.

An instance **J** is *MC-schedulable* if there exists a correct scheduling policy for it.

One can restrict the general MC-schedulability problem to the problem of finding solutions for certain classes of scheduling policies or to the instances that have special properties. In this paper we show that the HI jobs with $C(\text{LO}) = C(\text{HI})$ create a certain complication for checking the correctness of MC-scheduling solutions, which can be easily avoided by incrementing their $C(\text{HI})$ by a small δC. Therefore the following optional restriction is worth considering.
Restriction (i) $\chi(J_i) = \text{HI} \implies C_i(\text{LO}) < C_i(\text{HI})$.

The dual-criticality MC-scheduling problem is NP-hard and is also claimed to be in class NP [2], but in this paper we refute their proof for the latter claim. One can restrict the MC-scheduling problem to finding optimal solutions for *fixed priority* (FP) and *fixed-priority-per-mode* (FPM) scheduling policies, which we define in a moment. The former restricted problem is in class P (polynomially solvable), the latter is NP-hard, just as the general problem [2], while we show in this paper that certain cases of this problem are in class NP.

FP is a scheduling policy that can be defined by a priority table PT, which is a K-sized vector specifying all jobs in a certain order. The position of a job in PT is its *priority*, the earlier a job is to occur in PT the higher the priority it has. Among all ready jobs, the fixed-priority scheduling policy always schedules the m highest-priority jobs in PT.

Fixed priority per mode (FPM), a natural extension of fixed-priority for mixed critical systems. FPM is mode-switched policy with two tables: PT_{LO} and PT_{HI}. The former includes all jobs. The latter needs to include only the HI jobs. As long as the current criticality mode χ_{mode} is LO, this policy performs the fixed priority scheduling according to PT_{LO}. After a switch to the HI mode, this policy drops all pending LO jobs and applies priority table PT_{HI}. Suppose that after removing the LO jobs from PT_{LO} while keeping the same relative order of the HI jobs we obtain the PT_{HI} table. In this case one can just keep using the same priority table, PT_{LO}, after a switch to the HI mode with exactly the same result. Therefore in this particular case we say that we have *FPM-equivalent* tables: '$PT_{\text{LO}} \sim PT_{\text{HI}}$'. The below optional restriction of FPM scheduling problem allows to ensure certain useful properties:
Restriction (ii) Generate only solutions where: $PT_{\text{LO}} \sim PT_{\text{HI}}$

III. Correctness Test and Complexity

A. The Mixed Criticality Notion of Sustainability

To test the correctness of a scheduling policy one usually evaluates it for the scenario with maximal execution times for all jobs, which in our case corresponds to HI WCET's. However, to justify this test a scheduling policy must be *sustainable*, which means that increasing the execution time of any job A – while keeping all other execution times the same – may not make any other job B terminate earlier [4]. In other words, sustainability means that the termination times must be *monotonically non-decreasing* functions of execution times.

For mixed-critical scheduling the usual sustainability definition is too restrictive, as it does not take into account that an increase of an execution time of a HI job to a level that exceeds its LO WCET may lead to a mode switch and hence to dropping the LO jobs, which, in turn may lead to an earlier termination of another HI job, and hence non-monotonic dependency of termination times. Therefore, a weaker definition of sustainability is adopted for mixed criticality problems.

The new definition poses almost the same requirement of non-decreased termination time of any job B when we increase the execution time of a job A – while keeping all other execution times the same. However, now this property required to hold only when the increase of execution time of A leads neither to a change of the criticality mode in which B terminates nor to a switch of criticality mode by A. The second condition can only be violated if before the increase A executed for at most LO WCET and after the increase it exceeds the LO WCET. If at least one of these two conditions is violated then B may terminate earlier.

The adaptation of the notion of sustainability to mixed criticality raises the problem of how to adapt the policy correctness test to this new definition, as we cannot anymore rely on the traditional method of just testing the scheduling policy using just one maximal scenario: the "plain WCET".

B. Correctness Test and Computational Complexity

A general correctness test for a solution of a mixed-critical scheduling problem with a fixed set of jobs was systematically treated in [2], with the goal to study the *computational complexity* of the problem. The point is that the algorithmic complexity of the correctness test determines the complexity class of the problem. *If the correctness test is demonstrated to be simple enough so that it has at most polynomial complexity then the problem can be demonstrated to be in class NP.* To make the test algorithmically as simple as possible the test may "require" that a general-case solution be "*preprocessed*" into another solution such that the new solution is simpler to test. This permits to reduce the complexity of the subsequent test to

the minimum, thus maximizing the chances that the test can be demonstrated polynomial and the problem NP. Note also that *the preprocessing should be applicable to any correct solution* in the set of solutions of the given problem and it should produce at the output a solution that is correct if the input solution is correct.

In [2] a correctness test is proposed that generalizes the ordinary plain-WCET scenario testing to testing a polynomial number of basic scenarios. For the test to be applicable to a given scheduling policy the minimal requirement is that it must be sustainable (in the sense we defined earlier). We come back to this test in a moment.

To build the argument that the problem is in NP, Lemmas 1 and 2 in [2], in fact, define two preprocessing steps:
Step (1) – we call it "*preprocessing for sustainability*" – ensures several useful properties of the output solution at the same time. Firstly, it ensures that (a) the output policy is sustainable (even if the input policy is not), (b) that the testing requires to construct only a polynomial number of schedules, and (c) that every event in a schedule (job arrival, preemption and termination) contributes only a polynomial time to the total cost of the test.
Step (2)– we call it "*preprocessing for polynomially-sized schedules*" is supposed to ensure that the schedules have polynomial (in fact, linear) size, *i.e.,* perform only polynomial number of preemptions.

In Section IV we will refute Step (2) proposed in [2], *i.e.,* their Lemma 2, which breaks their argument for mixed critical scheduling be in class NP (but not the argument that it is NP-hard). We also show in Section V that Step (1) as well can pose complications for the claim of NP complexity, for the case of FPM scheduling.

C. Canonical Algorithm for Correctness Testing

In this subsection we describe the correctness testing algorithm and Step (1).

Definition 1: An online scheduling policy is basically correct for instance **J** if for any basic scenario of **J** the policy generates a feasible schedule.

Lemma 1: If a scheduling policy is sustainable and Restriction (i) applies to the problem instance then the policy correctness follows immediately from its basic correctness. In other words, if the policy gives a feasible schedule in all basic scenarios then this is also the case for the non-basic scenarios as well.

For proof see [1] Later on we show an single-processor example that violates Restriction (i) and that is only basically but not correctly schedulable by FPM with certain priority tables.

Lemma 2: An instance **J** is MC-schedulable if it admits a basically correct scheduling policy.

The above lemma is Lemma 1 from [2]. At the first glance it seems to be contradicting to a claim we have just made to show an FPM counterexample, but it should be noted that the lemma only claims that a correct policy exists, not that this policy is necessarily the same by which basically correct solution is constructed. In the proof given in [2] they show a

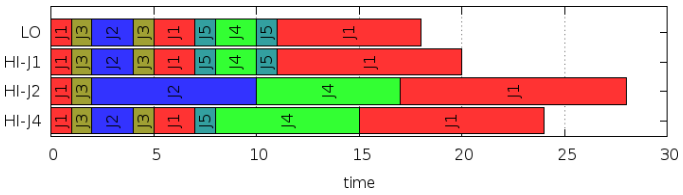

Fig. 1. The job-specific scenario schedules for Example 1 obtained with priority table $PT = (2, 4, 3, 5, 1)$

simple procedure to transform any basically correct policy into a similar policy that is, in addition, also sustainable, thus, by Lemma 1, yielding correct schedules in non-basic scenarios as well. This procedure corresponds to Step (1) of preprocessing for computational complexity, we come back to it in detail at the end of this subsection.

In fact, the above lemma implies that a complete correctness test can be reduced to testing all basic scenarios. However, this could not yield a polynomial testing algorithm, as there are exponential number of basic scenarios.

Fortunately, testing in all basic scenarios is redundant. Suppose that we have a sustainable scheduling policy. It turns out that to test the policy correctness for a dual-critical instance it suffices to simulate $H + 1$ basic scenarios, where H is the total count of HI jobs in the problem instance.

Consider a LO basic scenario schedule $\mathcal{S}^{LO}$ and select an arbitrary HI job J_h. Let us modify this schedule by assuming that at time t_h when job J_h reaches its LO WCET (C_h(LO)) it has not yet signalled its termination, thus provoking a mode switch. Then, by Condition 2, we should ensure that J_h and all the other HI jobs that did not terminate strictly before time t_h will meet their deadlines even when continuing to execute until their maximal execution time – the HI WCET. Note that in multiprocessor scheduling multiple jobs may also terminate *exactly* at time t_h in $\mathcal{S}^{LO}$, and they are conservatively assumed to also continue their execution after time t_h in the modified schedule. The behavior described above is formalized to a basic scenario where all HI jobs that execute after time t_h have HI WCET.

Definition 2: For a given problem instance, LO basic-scenario schedule S^{LO} and HI job J_h, the basic scenario defined above is called 'specific' for job J_h and is denoted HI-J_h, whereas its schedule is denoted $\mathcal{S}^{HI\text{-}J_h}$.

Note that $\mathcal{S}^{HI\text{-}J_h}$ coincides with $\mathcal{S}^{LO}$ up to the time when job J_h switches, and after the switching time it starts using HI execution times for the jobs that did not terminate before the switch.

Example 1: Fig. 1 shows Gantt charts for the job-specific scenarios of the following single-processor problem instance:

Job	A	D	χ	C(LO)	C(HI)
1	0	30	HI	10	12
2	2	10	HI	2	8
3	1	8	LO	2	2
4	8	17	HI	2	7
5	7	11	LO	2	2

We see, for example that in the LO scenario job J_2 terminates at time 4, but in the HI-$J2$ scenario job J_2 switches at time 4 and continues to execute, because, apparently, it has a HI WCET larger than the LO WCET. In fact, these schedules

are obtained from FPM policy and demonstrate that this policy is correct for the given problem instance, as explained later in Example 3.

Theorem 1: Under Restriction (i), to ensure correctness of a scheduling policy that is sustainable (in the mixed criticality sense) it is enough to test it for the LO scenario and the scenarios HI-J_h of all HI jobs J_h.

For proof see [1]

The above theorem, in fact, defines – for dual-criticality case – what we call *canonical correctness test* algorithm. It can be directly derived from the correctness test procedure described in [2], which is, however, more complex and more general, as it applies a number criticality levels more than two. Though that procedure, for efficiency reasons, would organize the schedules of basic scenarios in a tree structure and use backtracking, our less efficient formulation has only polynomially higher complexity, which does not impact on the reasoning on NP complexity.

Step (1) formulation. To prove NP complexity along the lines of reasoning given in [2] one has to demonstrate that any correct scheduling policy can be transformed into another one that is sustainable and basically correct and then apply Lemmas 1,2. For this, [2] specifies a procedure for this transformation, which can be reformulated as follows. As the output policy we use a mode-switched time-triggered policy which we call *Static Time-Triggered per Basic Scenario* (STTBS). This policy specifies a static time-triggered table for the LO scenario and all job-specific scenarios. The Gantt charts in Figure 1, in fact, specify such tables for the given example. The total length of all slots attributed to a given job in a given table should be at least equal to the job's execution time in the given basic scenario. The tables are obtained by simulation of the input policy in the given scenario. The execution of STTBS policy starts in the LO static table. If a job finishes earlier than the allocated time, the processors are idled in the remaining slots of that job. A job is allowed to continue execution for longer than its LO WCET, in that case the STTBS policy switches to the static table specific for that job. One can show that one can bypass Restriction (i) requirement of Lemma 1 and Theorem 1 when they are applied to STTBS solutions.

It should be noted that STTBS policy is radically different from what is usually referred to as "time-triggered scheduling" in mixed critical systems, which is *static time triggered table per mode* (STTM) [5], where there are only two time-triggered tables: one per mode. We will see an example of that policy in the next section.

Clearly, if we present as input to the canonical correctness testing algorithm the STTBS policy with $H+1$ static time-triggered tables then the properties (a),(b) and (c) that should be guaranteed by Step (1) are, in fact guaranteed. In particular, the simulation of one preemption takes a polynomial time, because the job that preempts another job is pre-specified in the STTBS table.

In fact, what remains to be shown for proving the NP upper bound on computational complexity is that the tested solution can, in addition, be presented in the form where only a polynomial number of preemptions occur in each job-specific scenario, this is what we call Step (2). Unfortunately, we have to observe that Step (2) proposed in [2] does not work in general case and hence is incorrect. We demonstrate it by a counterexample in the next section.

IV. Refuting the Proof of Polynomial Size

In this section we refute a lemma given [2]. This lemma was used as a cornerstone to prove that the canonical correctness test algorithm, when applied in general case, can have polynomial complexity, because the schedules can be restricted to have only a polynomial number of preemptions.

The lemma is copied below for convenience.

In the lemma $C_j(i)$ is the WCET estimate for job j at criticality level i and L is the number of criticality levels in the system. In our usual notations, level 1 is LO, level 2 is HI, $C_j(1)$ is C_j(LO), $C_j(2)$ is C_j(HI).

Lemma 3 (Refuted Lemma): If an instance is MC-schedulable, then there exists an optimal online scheduling policy that preempts each job j only at time points t such that at time t either some other job is released, or j has executed for exactly $C_j(i)$ units of time for some $1 \leq i \leq L$.

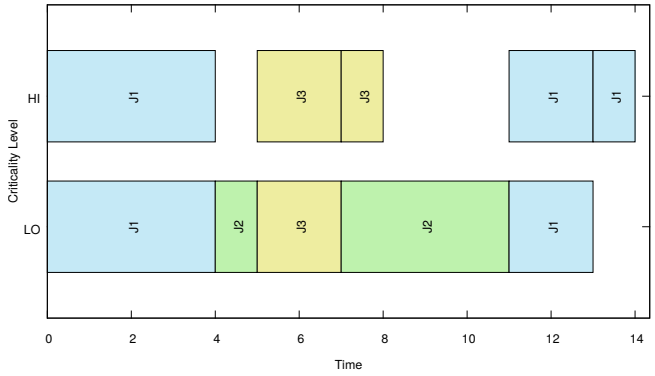

Fig. 2. A Valid Scheduling Policy for the Instance in Example 2

Example 2: Consider the following problem instance:

Job	A	D	χ	$C(1)$	$C(2)$
1	0	14	HI	6	7
2	0	11	LO	5	5
3	5	10	HI	2	3

Let us check if it is MC-schedulable according to Lemma 3. At $t=0$ we can execute either job J_1 or job J_2, whichever job we choose it should not be preempted before $t=5$.

1) If job J_1 is to be executed in the time interval $[0,5)$, then in the interval $[5,11)$, which is 6 time units, we will have to execute jobs J_2 and J_3 which combined need $7{=}(5+2)$ units of execution in the LO scenario. Thus we cannot execute J_1 in $[0,5)$.
2) Suppose that we execute job J_2 in $[0,5)$. What is then left to execute is the two high criticality jobs. We take the scenario that they both execute for their C(HI). Then we need a total of $10{=}(7+3)$ units in the execution window $[5,14)$, which has space for only 9 units.

Thus, according to Lemma 3 this instance is not MC-schedulable.

Figure 2 shows a Gantt chart representing an STTM scheduling policy [5] that correctly schedules that instance, contradicting Lemma 3. This policy starts execution in static table 'LO' and keeps using this table as long as there is no switch to the HI criticality mode $\chi = HI$, in which case it switches to static table 'HI'. This example shows that an instance can be MC-schedulable but no optimal online scheduling policy exists that preempts a job j only at time points where another job is released or j has executed for exactly $C_j(i)$ units.

V. Rehabilitation for Fixed Priority per Mode

In this section, for dual-criticality case, we "rehabilitate" the NP complexity argument established in [2] for the case of FPM – fixed priority per mode – policy. This is important, because this policy is popular in the literature, see *e.g.,* EDF-VD [6]. The "NP-Hard" classification as a "lower bound" on complexity, established in [2], remains valid when we restrict ourselves to FPM, therefore it is important to establish "NP" classification as an "upper bound". The refutation of Lemma 3 is not a problem for FPM, because this policy satisfies the statement of that lemma by construction.

However when we go from general-case MC-scheduling problem to a particular case of FPM scheduling problem and want to prove that it is in class NP, now Step (1) described in Section III encounters an obstacle. The solutions presented to the correctness test algorithm *should belong to the set of solutions of the problem for which we prove that it belongs to NP*. In the general problem formulation, the set of solutions includes all possible policies, and therefore presenting STTBS policies at the input of the correctness test was legal. Unlike the general MC-scheduling case, discussed in Section III, the set of solutions of the FPM scheduling problem consists *exclusively* of applications of FPM policy with different priority tables. Therefore, to prove that FPM is in NP we have to present FPM policies at the input of the correctness test algorithm, so Step (1) cannot be applied.

Therefore, we investigate whether Step (1) can be skipped and whether the canonical test algorithm can be applied directly to FPM policy. For this, the FPM policy should guarantee all the properties (a), (b), (c) ensured by Step (1). In fact, only property (a) – "*sustainability*" (in mixed-criticality sense) is not trivial and needs investigation and proof. Therefore, *we focus on the conditions under which FPM is sustainable*. Under these conditions it is also in class NP. We will only focus on dual-criticality case, leaving generalisation to more levels of criticality to future work.

The following theorem from [7] states a very useful property, for which we formulate a corollary:

Theorem 2: Fixed-priority policy is sustainable (in the default strict sense), for single- and multi-processor scheduling.

Corollary 3: For *single-processor* dual-criticality instances FPM is sustainable (in the mixed criticality sense). If, in addition, Restriction (i) is satisfied then the canonical correctness test is applicable to test FPM correctness for such instances. However, when Restriction (ii) is satisfied then Restriction (i) is not necessary and the scope of dual-critical FPM sustainability extends from single-processor to multiprocessor instances, whereby the canonical test is applicable as well.

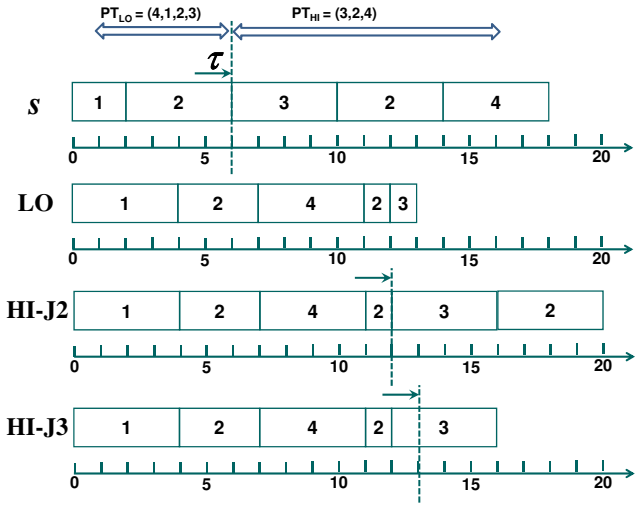

Fig. 3. Gantt charts (indices 1,2,...identify J_1, J_2, ...) of scenario $s = (c_1 = 1, c_2 = 8, c_3 = 4, c_4 = 4)$ in Example 4. Mode switch time of scenario s is $\tau = 6$. Also the LO and job-specific basic scenarios are shown. Note that job HI-J_4 is not given because J_4 cannot switch. Job J_4 misses deadline, but this is not captured in any basic scenario. This is possible because Restriction (i) requirement of Corollary 3 is not respected.

For proof see [1]. The corollary implies that under the specified conditions the FPM scheduling is in the class NP.

Note that Restriction (i) is quite general, as it can be ensured by an arbitrarily small increase of C_{HI} if $C_{\text{HI}} = C_{\text{LO}}$. Unfortunately, the sustainability of FPM cannot be asserted for multiprocessor case such general conditions, in this case we can only propose a quite restrictive Restriction (ii).

Example 3: Consider single-processor independent-job problem instance **J** defined in Example 1. For a certain priority table $PT_{\text{LO}} = PT_{\text{HI}}$, the Gantt chart in Figure 1 shows the execution of FPM policy on single processor in all scenarios required by the canonical correctness test. In those scenarios all jobs meet their deadlines. Since for this instance Restriction (ii) holds, FPM is sustainable and the canonical test is indeed applicable. Therefore the FPM policy with given priority table is correct for the given problem instance.

Example 4: To illustrate that Restriction (i) may be necessary let us consider the following single-processor problem instance **J**:

Job	A	D	χ	C(LO)	C(HI)
1	0	20	LO	4	4
2	0	20	HI	4	8
3	0	20	HI	1	4
4	7	11	HI	4	4

This problem instance violates Restriction (i) and hence it is not guaranteed that the canonical correctness test is applicable to FPM solutions directly. Figure 3 shows the Gantt chart of FPM policy for a specified non-basic scenario s and specified priority tables. In scenario s job J_4 has a deadline miss, but this is not visible to the canonical test, which checks in the LO and HI-job specific scenarios, whose Gantt charts are also shown. Note that job J_4 has $C(\text{LO}) = C(\text{HI})$, so it cannot switch and therefore we do not include the respective Gantt chart for HI-J_4, but even if we included that scenario it would not reveal the deadline miss either.

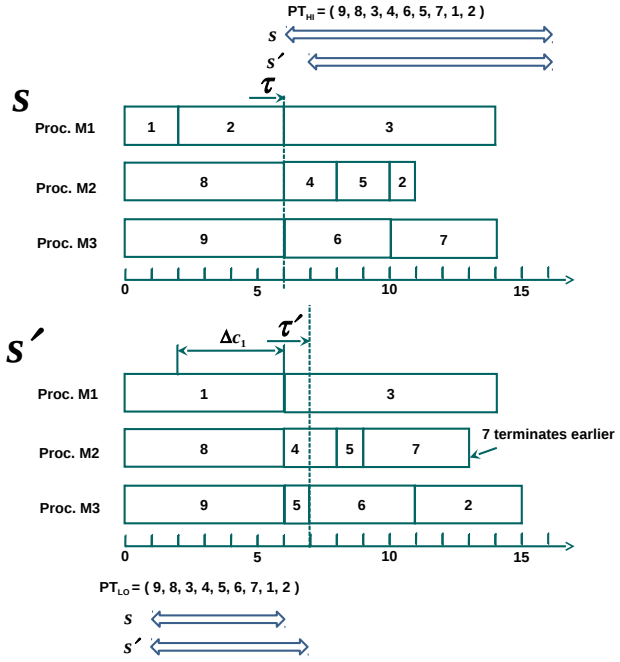

Fig. 4. FPM non-sustainability demonstration on multiprocessor case, using Example 5. Gantt charts (indices 1,2,... identify J_1, J_2, ...) of two scenarios: s and s'. Mode switch times are τ and τ', resp. Scenario s is defined by $(c_1 = 2, c_2 = 5, c_3 = 8, c_4 = 2, c_5 = 2, c_6 = 4, c_7 = 4, c_8 = c_9 = 6)$. Scenario s' differs from s by $c'_1 = c_1 + \Delta c_1 = 2 + 4$. Job J_7 violates sustainability by terminating in s' earlier than in s, while terminating in the same mode (HI) in both scenarios.

VI. Complication for Multiprocessor Case

In this section we give a counterexample for sustainability of FPM on multiple processors and discuss the consequences.

Example 5: Consider the following problem 3-processor problem instance **J** be defined by:

Job	A	D	χ	C(LO)	C(HI)
1	0	6	LO	6	6
2	0	14	HI	4	5
3	6	15	HI	7	8
4	6	8	HI	1	2
5	6	9	HI	1	2
6	6	11	HI	3	4
7	6	13	HI	3	4
8	0	6	LO	6	6
9	0	7	LO	6	6

The Gantt chart in Figure 4 shows execution in two scenarios: s and s' for the priority tables specified in the figure, whereby Restriction (ii) $PT_{\text{LO}} \sim PT_{\text{HI}}$ is not satisfied and hence sustainability is not guaranteed. Scenario s' differs from scenario s only by a larger execution time of J_1. Nevertheless we see that job J_7 (as well as J_5) terminates in scenario s' earlier than in scenario s. This behavior contradicts the requirements of sustainability.

Note that in scenario s jobs J_5 and J_7 miss their deadlines, whereas in s' they do not. If correction test algorithm were used for this case, it would not check for scenario s, because it is not basic. It would check in s', which is HI-J_4, in the other job-specific scenarios and in the LO scenario. Since in these scenarios the FPM policy would not miss the deadlines, it would come to conclusion that the proposed priority tables are correct, whereas, as we see this is not true. The test algorithm would come to a wrong conclusion because the condition on sustainability of the policy is not satisfied.

Unlike Example 4, this example illustrates not just an exceptional case but well-known common properties of multiprocessor scheduling, differentiating them from single-processor case. Changing the order of job execution leads to a change of load distribution of different jobs between processors, which leads to different interference *w.r.t.* lower priority jobs. In our case, in window $[\tau, \tau']$ swapping the priority order between J_5 and J_6 has perturbed the load balance between the processors, such that a smaller priority job J_7 terminates earlier. Note that in both priority tables the set of jobs that have higher priority than J_7 is the same and all of them arrive no later than J_7. Under the same conditions on single processor these jobs would inevitably have the same total interference on J_7 in the two scenarios, but not on multiple processors.

From the above it follows that FPM cannot be shown to be in NP or PSPACE for multiprocessors by following the same line of reasoning as proposed in [2].

VII. Conclusions

In this paper we have reconsidered the results concerning the computational complexity of mixed critical scheduling of a fixed set of jobs. We have refuted the proof that mixed critical schedules can be restricted to have size polynomial on the number of jobs without loosing optimality. This can mean that the problem may have a complexity beyond NP and PSPACE.

We have also studied the computational complexity of of the special case of fixed-priority per mode scheduling. We have discovered that this problem can be shown in class NP only in the single-processor case, since this policy turned out to be non-sustainable in multiprocessor case. In the extended version [1], we also give the sustainability proof for single-processor case.

References

[1] R. Kahil, P. Poplavko, D. Socci, and S. Bensalem, "Revisiting the computational complexity of mixed critical scheduling," Tech. Rep. TR-2017-7, Verimag Research Report, 2017. http://www-verimag.imag.fr/Technical-Reports,264.html?lang=en&number=TR-2017-7.

[2] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *IEEE Trans. Comput.*, vol. 61, pp. 1140 –1152, aug. 2012.

[3] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Multiprocessor scheduling of precedence-constrained mixed-critical jobs," in *IEEE ISORC 2015*, 2015.

[4] S. K. Baruah and A. Burns, "Sustainable scheduling analysis," in *Real-Time Systems Symposium (RTSS 2006)*, pp. 159–168, 2006.

[5] S. Baruah and G. Fohler, "Certification-cognizant time-triggered scheduling of mixed-criticality systems," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pp. 3–12, 2011.

[6] S. Baruah, V. Bonifaci, G. D'Angelo, 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 *Euromicro Conf. on Real-Time Systems*, ECRTS'12, pp. 145–154, IEEE, 2012.

[7] R. Ha and J. W. S. Liu, "Validating timing constraints in multiprocessor and distributed real-time systems," in *Proc. Int. Conf. Distributed Computing Systems*, pp. 162–171, Jun 1994.