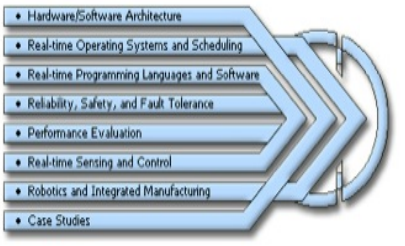
IEEE REAL-TIME SYSTEMS SYMPOSIUM

RTSS 2013 DECEMBER 3-6, VANCOUVER, CANADA



WMC

Proceedings of the 1st International Workshop on Mixed Criticality Systems

Edited by Liliana Cucu-Grosjean and Rob Davis

Program Chairs

Liliana Cucu-Grosjean
Rob Davis

Steering Committee

Sanjoy Baruah
Liliana Cucu-Grosjean
Rob Davis
Claire Maiza

Program Committee

Sanjoy Baruah
Arvind Easwaran
Sebastian Faucou
Laurent George
Raphael Guerra
Leandro Indrusiak
Karthik Lakshmanan
Giuseppe Lipari
Claire Maiza
Vincent Nelis
Moritz Neukirchner
Gabriel Parmer
Susan van der Ster
Wang Yi

Message from the Program Chairs

It is our pleasure to welcome you to the 1st International Workshop on Mixed Criticality Systems (WMC) at the Real-Time Systems Symposium (RTSS) in Vancouver Canada, 3rd December 2013.

The purpose of WMC is to share 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. The workshop will promote lively interaction, cross fertilisation of ideas, synergies, and closer collaboration across the breadth of the real-time community, as well as attracting industrialists from the aerospace, automotive and other industries with a specific interest in MCS.

For the first edition of the workshop a total of 18 submissions were received. The review process involved 14 Program Committee members, with each submission receiving at least 3 reviews. The overall standard of submissions was exceptionally high. In total, 14 papers were selected for presentation. Our thanks go to the WMC Program Committee for the time and effort they put into carefully reviewing the submissions, and for meeting the tight timescales set for reviews.

In addition to the regular papers, the workshop program also includes an invited talk from Steve Vestals whose seminal paper, "*Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance*" from RTSS 2007 underpins the line of research into the mixed critical systems that continues to this day.

WMC 2013 would not be possible without the hard work of a number of people involved in the organisation of RTSS. In particular, we would like to thank the RTSS 2013 Workshops Chair, Nathan Fisher (Wayne State University, USA) for his excellent organisation of the overall workshop program. We also thank James Harbin (University of York, UK) for the excellent website design, the WMC Steering Committee for their guidance, and the MCC (UK EPSRC EP/K011626/1), Proxima (EU FP7 IP 611085) and Departs (French BGLE O16526-405635) projects for their support.

Finally, we would like to thank *all* of the authors who submitted their work to WMC 2013, whether it was accepted or not; without them, this workshop would not be possible.

We wish you an interesting and exciting workshop and an enjoyable stay in Vancouver. We look forward to seeing you again at WMC 2014.

Liliana Cucu-Grosjean (INRIA, Paris-Rocquencourt, France)
Rob Davis (University of York, UK)
WMC 2013 Program Chairs

Table of Contents

Session 4

Towards A More Practical Model for Mixed Criticality Systems

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

S.K. Baruah
Department of Computer Science,
University of North Carolina, US.
Email: baruah@cs.unc.edu

***Abstract*—Mixed Criticality Systems (MCSs) have been the focus of considerable study over the last six years. This work has lead to the definition of a standard model that allows processors to be shared efficiently between tasks of different criticality levels. Key aspects of this model are that a system is deemed to execute in one of a small number of criticality modes; initially the system is in the lowest criticality mode, but if any task executes for more than its predefined budget for this criticality level then a mode change is made to a higher criticality mode and all tasks of the lowest criticality level are abandoned (aborted). The initial criticality level is never revisited. This model has been useful in defining key properties of MCSs, but it does not form a useful basis for an actual implementation of a MCS. In this paper we consider the tradeoffs stemming from a consideration of what systems engineers require at run-time and the actual properties of the model that scheduling analysis guarantees. Alternative models are defined that allow low criticality tasks to continue to execute after a criticality mode change. The paper also addresses robust priority assignment.**

I. Introduction

Although the formal study of mixed criticality systems (MCSs) is a relatively new endeavor, starting with the paper by Vestal (of Honeywell Aerospace) in 2007 [24], a standard model has emerged (see for example [4], [5], [13], [14], [19]). For dual criticality systems this standard model has the following properties:

- A mixed criticality system is defined to execute in either of two modes: a HI-crit mode and a LO-crit mode.
- Each task is characterised by the minimum inter-arrival time of its jobs (period denoted by T), deadline (relative to the release of each job, denoted by D) and worst-case execution time (one per criticality level), denoted by $C(HI)$ and $C(LO)$. A key aspect of the standard MCS model is that $C(HI) \geq C(LO)$.
- The system starts in the LO-crit mode, and remains in that mode as long as all jobs execute within their low criticality computation times ($C(LO)$).
- If any job executes for its $C(LO)$ execution time without completing then the system immediately moves to the HI-crit mode.
- As the system moves to the HI-crit mode all LO-crit tasks are abandoned. No further LO-crit jobs are executed.
- The system remains in the HI-crit mode.
- Tasks are assumed to be independent of each other (they do not share any resource other than the processor).

This abstract behavioural model has been very useful in allowing key properties of mixed criticality systems to be derived, but it has met with some criticism from systems engineers[1] that it does not match their expectations. In particular:

- In the HI-crit mode LO-crit tasks should not be abandoned but be allowed to make some progress, as long as they are not interfering with HI-crit tasks.
- For systems which operate for long periods of time it should be possible for the system to return to the LO-crit mode when the conditions are appropriate.
- Whereas a HI-crit job executing for more than its $C(LO)$ execution time must induce a mode change, a LO-crit job should be constrained so that it cannot execute for more than $C(LO)$ (so no a mode change).

Some of these criticisms are partly misplaced as any high integrity system should remain in the LO-crit mode for its entire execution: the transition to HI-crit mode is only a theoretical possibility that the scheduling analysis can exploit [4]. Nevertheless, in less critical applications (such as those envisaged in the automotive industry) actual criticality mode changes may be experienced during operation and the above criticisms should be addressed.

Our contributions. In this paper we address all of the concerns listed above. First, we present alternative implementation models that (a) do not abandon LO-crit tasks upon transitioning to HI-crit mode; and (b) define conditions for the system to transition back to LO-crit mode. And second, we propose priority-assignment techniques for fixed-priority mixed-crit schemes that are more *robust* than previously-proposed techniques in the sense that systems assigned priorities according to these robust priority-assignment schemes are less likely to undergo a mode change to HI-crit mode. In other work [9] we have discussed (and removed) a further criticism of the standard model – that tasks are independent. This was done by revisiting and adapting the original priority ceiling protocol. Other proposals comes from Lakshmanan et al. [15] They define two protocols: PCIP (Priority and Criticality Inheritance Protocol) and PCCP (Priority and Criticality Ceiling Protocol). Both of these contain the notion of criticality inheritance. This

[1]For example at the tutorial presented at the 2012 Embedded System Week http://www.esweek.org/ and at the workshop that was part of the 2013 HiPEAC conference http://www.hipeac.net/conference/berlin/workshop/integration-mixed-criticality-subsystems-multi-core-processors.

notion is also used by Zhao et al. [25] in their HLC-PCP (Highest-Locker Criticality Priority Ceiling Protocol).

Related work. Background material on MCS research can be obtained from the following papers [2], [4], [5], [12]–[14], [24]). A survey on MCS research is available from the MCC (Mixed Criticality Systems on Many-core Platforms) project's web site[2] Santy et al. [19] attempt to remove some of the strictness of the standard model; however, they largely focus on a different set of issues.

II. Limitations Imposed by Scheduling Analysis

The standard model requires an immediate change to the HI-crit mode and the consequent abandonment of all active LO-crit jobs upon a change to HI-crit mode. Despite this simplifying assumption, it has been shown [2] that the mixed criticality schedulability problem is strongly NP-hard even if there are only two criticality levels. Hence only sufficient rather than exact analysis is computationally feasible. One of the consequences of this intractability is that a significant proportion of the available (sufficient rather than exact) analysis that has been produced for MCSs actually assumes that any LO-crit job that has been released at the time of the mode change will complete, rather than being aborted – this is the case with, e.g., the analysis in [2], [4], [24] (although not the analysis of the EDF-VD algorithm [3]).

For example, for constrained deadlines tasks, the Adaptive Mixed Criticality (AMC Method 1 or AMC-rtb) approach presented at RTSS in 2011 [4] first computes the worst-case response times for all tasks in the LO-crit mode (denoted by $R_i(LO)$). This is accomplished by solving, via fixed point iteration, the following response-time equation for each task:

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

where $\mathbf{hp(i)}$ is the set of all tasks with priority higher than that of task τ_i.

During the criticality change we are only concerned with HI-crit tasks, so for these tasks:

$$R_i(HI) \;=\; C_i(HI) \;+\; \sum_{\tau_j \in \mathbf{hpH}(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(HI) \;+\; \sum_{\tau_k \in \mathbf{hpL}(i)} \left\lceil \frac{R_i(HI)}{T_k} \right\rceil C_k(LO) \quad (2)$$

where $\mathbf{hpH(i)}$ is the set of HI-crit tasks with priority higher than that of task τ_i and $\mathbf{hpL(i)}$ is the set of LO-crit tasks with priority higher than that of task τ_i. So $\mathbf{hp(i)}$ is the union of $\mathbf{hpH(i)}$ and $\mathbf{hpL(i)}$. Note $R_i(HI)$ is only defined for HI-crit tasks.

This equation is conservative for AMC as it does not take into account the fact that LO-crit tasks cannot execute for the entire busy period of a high criticality task in the HI-crit mode.

[2]http://www.cs.york.ac.uk/research/research-groups/rts/mcc/.

A change to HI-crit must occur before $R_i(LO)$ and hence (2) can be modified as follows:

$$R_i(HI) \;=\; C_i(HI) \;+\; \sum_{\tau_j \in \mathbf{hpH}(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(HI) \;+\; \sum_{\tau_k \in \mathbf{hpL}(i)} \left\lceil \frac{R_i(LO)}{T_k} \right\rceil C_k(LO) \quad (3)$$

which 'caps' the interference from LO-crit tasks as $R_i(HI)$ must be greater than $R_i(LO)$.

The cap is however at its maximum level. The maximum number of LO-crit jobs are assumed to interfere and each of these jobs is assumed to complete – each inducing the maximum interference of $C_k(LO)$.

Finally in this section we note that if, for any HI-crit task, $R_i(HI) \leq D_i$ during the transition to the HI-crit mode then this task will remain schedulable once the HI-critically mode is fully established, and there is no execution from LO-crit tasks.

III. Alternative Models

Notice that one of the consequences of the limitations imposed by the schedulability analysis that we described in Section II above is that any LO-crit job, once released, is assumed to complete. This "limitation" can be exploited in an implementation model by only allowing the status of LO-crit tasks to be changed when they are suspended between jobs.

To accommodate the requirement to allow some progress for LO-crit tasks after the move to HI-crit mode, and to allow a mode change back to LO-crit mode, LO-crit tasks must not be abandoned. Rather they must remain runnable but in a way that cannot impact on HI-crit jobs. For a fixed priority system this means: (i) changing the priority of these tasks to be below the lowest priority of any HI-crit task, or (ii) reducing the execution time requirements of these tasks so that both HI and LO criticality tasks can execute successfully in the HI-crit mode, or (iii) extending the period of the LO-crit tasks to achieve the same result.

In the rest of this paper we will concentrate on single processor systems scheduled using fixed priorities. We consider three complimentary schemes. In the first LO-crit tasks have their priorities reduced, in the second they have their execution-time requirements reduced, and in the third they have their periods extended.

A. Reducing LO-crit Tasks' Priorities

Although with fixed priority scheduling priorities are 'fixed', to accommodate the requirements identified above, it must be possible to dynamically change the priority of a LO-crit task. Such tasks will have two priorities, $P_i(LO)$ and $P_i(HI)$; with the constraints that $P_i(HI) \leq P_i(LO)$ and $P_i(HI) < \min_{j \in \mathbf{HI}}(P_j)$ where $\mathbf{HI}$ is the set of all HI-crit tasks. We note that most RTOSs and programming languages allow the base priority of a task to be altered. For optimal performance the relative ordering of the priorities of LO-crit tasks will be the same in both criticality modes.

At run-time the overhead cost of changing the priority of a runnable task can be relatively high as the task must be taken out of the run queue and then reinserted at the place appropriate for its new priority. Fortunately due to the limitations of the analysis, which not only assumes all LO-crit jobs complete but also assumes they do so at their current priority, it is acceptable to only modify the priority of a task (from $P_i(LO)$ to $P_i(HI)$) when the task is suspended.

To return the system from HI-crit to LO-crit requires that a further mode change is undertaken. This is a more extensive mode change as new work (the LO-crit tasks) needs to be reintegrated with the HI-crit work. Although there is analysis that attempts to deal with complex mode change protocols (eg. [17]), the most straightforward and easily verified protocol to use is one that simply waits for a system idle tick and then makes the mode change [23]. As the system is idle at that point there can be no behavioural impact on the new LO-crit mode from the previous HI-crit mode. This issue has been further investigated in two recent papers [16], [20].

All but the simplest models of MCS require that the execution times of all jobs are monitored. Most RTOSs will allow this to be done for single processor systems (although the problem for multi-core platforms with shared buses remains an open issue). For LO-crit tasks, an adequate behaviour is for them to be prevented from executing for more than their $C(LO)$ budget. For HI-crit jobs there is no run-time benefit to be gained from capping their execution times, but the criticality mode switch must be made if any HI-crit job executes for its $C(LO)$ value without signaling completion.

B. Reducing LO-crit Tasks' Execution Time Budgets

One possible criticism of the above scheme is that if LO-crit tasks can have their priorities changed, this capability could be exploited in a security breach to undermine the assurances given to the HI criticality tasks. Another problem with the approach is that no guarantees can be given to short deadline LO-crit jobs executing after the mode change. In practice there is likely to be spare capacity available (once the HI-crit tasks have been scheduled in the high criticality mode) and this capacity could be used to guarantee at least some level of service to some of the LO-crit tasks. In this section we introduce a different mixed criticality model that explicitly retains (some) LO-crit work in the HI-crit mode.

We first introduce the model under the assumption that there is a specific level of LO-crit work that must be guaranteed; i.e. there is a schedulability test that will either accept, or not, a given task set. This test is then used, in the context of sensitivity analysis, to explore what levels of service are possible whilst retaining schedulability.

The *modified system model* is as follows. Each task, τ_i is defined by the parameters: T_i, D_i, L_i, $C_i(LO)$ and $C_i(HI)$. For HI-crit tasks $C_i(HI) \geq C_i(LO)$, for LO-crit tasks $C_i(LO) \geq C_i(HI)$. Note, for some LO-crit tasks $C_i(HI)$ may be zero meaning that no jobs of such tasks start their execution once the system is in the HI-crit mode. Static priorities (P_i) are assigned to the tasks according to Audsley's Optimal Priority Assignment algorithm [1] (as explained in [4], [24]). The system model is now defined by the following behaviours:

- The system starts in the LO-crit mode, and remains in that mode as long as all HI-crit tasks execute within their low criticality computation times ($C(LO)$).
- If any job of a HI-crit task τ_i executes for its $C_i(LO)$ value without completing then the system immediately moves to the HI-crit mode.
- The priorities of tasks are never modified.
- No job of a LO-crit task τ_k is allowed to execute for more than its $C_k(LO)$ parameter in the LO-crit mode or its $C_k(HI)$ parameter in the HI-crit mode; any attempt to do so will result in the task being suspended until its next release (at least T_k after its last release).
- There is no bound on the execution time of HI-crit tasks.
- If the system is in the HI-crit mode and there is an idle tick then the system can move back to the LO-crit mode and all LO-crit tasks can have their execution time budgets restored to their original $C(LO)$ values.

Note that a LO-crit job released in the LO-crit mode is not guaranteed to receive processing time of $C(LO)$ by its deadline, if there is a mode change during its execution and its budget is reduced. It is however guaranteed to receive processing time of $C(HI)$ by its deadline.

We can now give the schedulability test for this model, using the AMC-rtb (Method 1) approach. Equation (1) is again used to compute the worst-case response time of all tasks in the LO-crit mode. Considering the HI-crit mode, the starting point is (3). This equation assumes that no LO-crit job released after $R_i(LO)$ can interfere with a HI-crit task τ_i. Now we allow interference, but at a lower level. This is easily accommodated; firstly for HI-crit tasks:

$$R_i(HI) = C_i(HI) + \sum_{\tau_j \in \mathbf{hpH}(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(HI) + \sum_{\tau_k \in \mathbf{hpL}(i)} \left\lceil \frac{R_i(LO)}{T_k} \right\rceil C_k(LO) + \sum_{\tau_k \in \mathbf{hpL}(i)} \left(\left\lceil \frac{R_i(HI)}{T_k} \right\rceil - \left\lceil \frac{R_i(LO)}{T_k} \right\rceil \right) C_k(HI) \quad (4)$$

Note that as $R_i(HI)$ is always greater than or equal to $R_i(LO)$ the final term in (4) is never negative. Equation (4) thus assumes the maximum possible number of releases of LO-crit tasks with the higher execution time.

An alternative form for (4) is available by simply rearranging the terms:

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

$$\sum_{\tau_k \in \mathbf{hpL}(i)} \left\lceil \frac{R_i(HI)}{T_k} \right\rceil C_k(HI)$$

to give:

$$R_i(HI) = C_i(HI) + \sum_{\tau_j \in \mathbf{hp}(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(HI) + \sum_{\tau_k \in \mathbf{hpL}(i)} \left\lceil \frac{R_i(LO)}{T_k} \right\rceil (C_k(LO) - C_k(HI)). \quad (5)$$

For a LO-crit task τ_i, which now continues after the mode change, but with a reduced execution time requirement of $C_i(HI)$, the above equation also applies; however, a tighter formulation is also possible. We note that if the task has already executed for $C_i(HI)$ in the LO-crit mode, then it has trivially met its requirements in the HI-crit mode. Therefore we need only consider the case where the mode change occurs before it has executed for $C_i(HI)$. Equation (1) computes the worst-case response-time for LO-crit tasks assuming that the task's own requirement is $C_i(LO)$, but here the only scenario of interest is when the mode change occurs before it has executed for $C_i(HI)$ which must be earlier as $C_i(HI) \leq C_i(LO)$. Hence (1) can be modified for LO-crit tasks to give:

$$R_i^*(LO) = C_i(HI) + \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{R_i(LO)}{T_j} \right\rceil C_j(LO) \quad (6)$$

and (4) then becomes:

$$R_i(HI) = C_i(HI) + \sum_{\tau_j \in \mathbf{hpH}(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(HI) + \sum_{\tau_k \in \mathbf{hpL}(i)} \left\lceil \frac{R_i^*(LO)}{T_k} \right\rceil C_k(LO) + \sum_{\tau_k \in \mathbf{hpL}(i)} \left(\left\lceil \frac{R_i(HI)}{T_k} \right\rceil - \left\lceil \frac{R_i^*(LO)}{T_k} \right\rceil \right) C_k(HI) \quad (7)$$

again this can be rearranged to give:

$$R_i(HI) = C_i(HI) + \sum_{\tau_j \in \mathbf{hp}(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(HI) + \sum_{\tau_k \in \mathbf{hpL}(i)} \left\lceil \frac{R_i^*(LO)}{T_k} \right\rceil (C_k(LO) - C_k(HI)). \quad (8)$$

As indicated earlier, (4) and (7) (or (5) and (8)) could be used to test a specific application's requirements. More practically, they would be used to explore the design space – how much guaranteed capacity is available once all HI-crit tasks are validated? Sensitivity analysis [8], [18] could be used to explore this space. Possible questions to consider are:

- How many LO-crit tasks can have their full capacity (i.e. $C(HI) = C(LO)$) with the rest having $C(HI) = 0$?
- By how much must all the $C(LO)$ of LO-crit tasks be reduced (i.e. $C(HI) = \alpha \cdot C(LO)$) with $\alpha < 1$ to give a schedulable system?
- Can some or all LO-crit tasks employ alternative versions that take less resources?

With this implementation model the mode change back to LO-crit is straightforward. Again a low priority 'background' task can be used to implement the change back to LO-crit mode; however, now the only action of this task is to return the budgets for each LO-crit task to their larger LO-crit values ($C(LO)$ rather than $C(HI)$). This should be done atomically to avoid any potential race condition.

C. Increasing LO-crit Tasks' Periods

The elastic scheduling model [10] has been applied [22] to EDF scheduled mixed criticality systems to allow a LO-crit task to have its period extended after a mode change. We can apply this idea to fixed priority systems by allowing a LO-crit task to have two period values: $T_i(LO)$ and $T_i(HI)$ with $T_i(LO) \leq T_i(HI)$. After a mode change the task can be released again but with an extended period (and perhaps also a reduced budget). The equations of the previous section can be extended to include a revised period after the mode change. But as space is restricted this is left as an exercise for the reader.

D. Capacity Inheritance with the Budget Reduction Scheme

The previous subsections have introduced three different schemes for dealing with LO-crit tasks following a mode change to HI-crit. One reduces LO-crit tasks to, in effect, the background level; here they can utilise all available spare capacity, but the schedulability of LO-crit tasks after the criticality mode change is seriously undermined. The other schemes guarantees some level of service, but does not utilise spare capacity. This is a significant drawback as there is likely to be considerable spare capacity in the HI-crit mode. A HI-crit job may execute for more than $C(LO)$, but will most likely complete well before it has used its full $C(HI)$ budget – most of $\big(C(HI) - C(LO)\big)$ could be available for LO-crit jobs.

To improve the effectiveness of the second scheme two strategies are possible:

1) Use the schemes together – once a low criticality job has used up its $C(HI)$ budget its priority is lowered to its background level ($P_i(HI)$) where it can continue to execute.
2) The spare capacity from HI-crit jobs is directly assigned to LO-crit jobs.

The second strategy can exploit previously published techniques such as Extended Priority Exchange [21], Capacity Sharing [7] and History Rewriting [6] that were developed for combined hard and soft real-time fixed priority task sets. Within the context of mixed criticality systems these techniques would work as follows. Assume the system is in the HI-crit mode.

- All HI-crit tasks have a budget equal to their maximum, guaranteed, execution time $C(HI)$.
- At run-time the actual execution time of each HI-crit job is monitored;
- When a job with release time s and absolute deadline d (with $d = s + D$) completes, its actual execution time is noted (e) and its *gain* time (g) is computed ($g = C(HI) - e$); g is assumed to be non-negative.
- The gain time is available to be allocated to lower priority jobs.
- The gain time must be used by d (its expiry time).

The three techniques referenced above allocate the gain time in different ways. For Extended Priority Exchange [21]:

- The gain time is allocated to the budget of the next highest priority task that is ready to execute. (Note if there is no ready task, then the gain time is lost).

For Capacity Sharing [7]:

- An executing job first uses its own budget of $C(HI)$.
- When this budget is exhausted it 'pulls down' extra capacity from any available higher priority gain time.
- The LO-crit job is said to be *plugged* to the budget of the *host* HI-crit job.
- The plug is broken when the expiry time is reached, the capacity is exhausted or the job completes.

If there are a number of higher priority gain times available then any can be chosen and indeed more than one can be utilised, though only one at a time. Useful heuristics to use are: use the biggest g first, or use the earliest d first. An analysis of these heuristics and the implementation efficiency of the Capacity Sharing scheme is described in [7].

For History Rewriting [6] a retrospective reallocation of budgets is undertaken. The problem with Capacity Sharing is that the gain time has to be used before its expiry time or it is lost. This is not the case with History Rewriting which has the following characteristics:

- At the deadline (d) of a job the gain time is noted (g).
- A lower priority task that has executed for e before d is chosen and its budget is increased by $\max(g, e - g)$, g is reduced by this amount.
- Any remaining gain time is further allocated to lower priority tasks.

In effect, a job that was executing from its own budget is deemed to have been executing from the gain time of a higher priority job, and hence its own budget is intact and can be used to further the execution of the job. Further details of this approach are given in [6].

Job	Crit	C(LO)	C(HI)	s	d	e	g
τ_1	HI	1	4	0	8	2	2
τ_2	LO	6	4	4	12	-	-

TABLE I
TWO JOB EXAMPLE

For an example of History Rewriting consider the simple system of two jobs as defined in Table I. In the HI-crit mode both jobs are guaranteed 4 units of execution (although the LO-crit job would prefer 6). As τ_1 only executes for 2 units there is a gain time of 2 at time 2. But τ_2 is not active at time 2 and so the gain time cannot be utilised with Extended Priority Exchange or Capacity Sharing. With History Rewriting, however, the gain time of 2 becomes available at the deadline of τ_1 at time 8 after τ_2 has executed for 4 units. Two of these units can now be considered to be gain time and hence τ_2 can execute at time 8 for 2 more units thereby satisfying its full requirement.

IV. ROBUST PRIORITY ASSIGNMENT

For a dual-crit system $C(LO)$ values must, of course, be known. Once schedulability has been established however, it is possible to derive [19], using sensitivity analysis, a scaling factor f ($f > 1$) such that the system remains schedulable with all $C(LO)$ values replaced by $f \cdot C(LO)$. Using these scaled values at run-time will increase the robustness of the system, as LO-crit tasks will be able to execute for longer before they are suspended. Further, HI-crit tasks will be able to execute for longer without inducing a mode change.

Although the work of Santy et al. [19] allows the computation time of tasks to be increased it does this without modification to the priority ordering of the tasks. Here, we note that as the use of Audsley's Optimal Priority Assignment algorithm takes into account task computation times, a scheme that looks to increase robustness by extending the allowed execution times, will perform better if it also considers priority assignment.

In their work on Robust Priority Assignment algorithms, Davis and Burns [11] showed that for a general class of additional interference functions, Deadline Monotonic (DM) is both an optimal and a robust partial ordering for any subset of tasks that would on their own have DM as their optimal priority ordering. In this way, HI-crit tasks may be viewed as the subset of tasks which have DM as their optimal partial order with the LO-crit tasks constituting additional interference, and also vice-versa (assuming that all of the tasks have constrained deadlines). Hence an overall optimal and robust priority ordering can be achieved via a merge of the DM partial order of HI-crit tasks with the DM partial order of LO-crit tasks as described in [4]. This requires at most $2n-1$ task schedulability tests for a system of n tasks.

We note that changes to execution times will not affect the separate DM partial orderings of the two groups of tasks, but will potentially change the merge and hence the overall priority ordering. For example, if all $C(LO)$ values are close to zero but all $C(HI)$ values for HI-crit tasks are at their maximum level for schedulability then a total priority ordering in which all HI-crit tasks have higher priorities than all LO-crit tasks is optimal; however, if we increase the budgets for LO-crit execution and make $C(LO)$ equal to $C(HI)$ for all tasks then the optimal and robust priority ordering has the complete set of tasks in DM order.

Given the benefits that priority reassignment provides, we recommend this straightforward extension to Santy et al.'s

approach [19].

V. Conclusions

This paper has addressed some of the issues that have been raised with what has become the standard analysis model for mixed criticality systems. As the tightest available analysis often assumes any released LO-crit job completes, there is no benefit for the actual run-time behaviour to require that LO-crit jobs are immediately abandoned. There is, however, a need for a real implementation to incorporate a means of returning the mode of the system to the initial mode if conditions are acceptable for this to occur. These needs are satisfied by the model presented in this paper. Other topics covered in this paper include allowing reduced but still guaranteed behaviour for low criticality tasks after a criticality mode change, and the use of robust priority assignment to reduce the likelihood of such mode changes.

This paper, like many on mixed criticality, is limited to only addressing dual criticality systems. This restriction helps to clarify descriptions. However, it is important that protocols do generalise to a realistic number of criticality levels. Perhaps up to five levels may be needed (see, for example, the IEC 61508, DO-178B, DO-254 and ISO 26262 standards). In the models presented in this paper, HI-crit tasks have the particular property that they are not themselves abandoned if they execute for more than their budgets. As they have the highest criticality level the best run-time behaviour is always to allow them to continue to execute. When more than two levels are present then only the highest criticality level retains this 'privilege'. All the others must be monitored and criticality mode changes affected where necessary. The models developed in this paper are easily extended to a small number of levels. With, say, five levels of criticality it is unlikely that a task will have five levels of service defined. Nevertheless the models defined do have the capability to be used in this way.

Taken together, the contributions of this paper aim to define a model for mixed criticality systems that has practical utility. It aims to ease the movement of the wealth of theoretical results that have appeared since 2007 into industrial practice.

Acknowledgements

This research has been supported in part by NSF grants CNS 0834270, CNS 0834132, and CNS 1016954; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; AFRL grant FA8750-11-1-0033 and EPSRC(UK) grant MCC (EP/K01 1626/1). We thank Rob Davis for this assistance.

References

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

[2] S.K. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 61(8):1140–1152, 2012.

[3] S.K. 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 *Proc. of ECRTS, Pisa*, pages 145–154, 2012.

[4] S.K. 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.

[5] S.K. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *ECRTS*, pages 147–155, 2008.

[6] G. Bernat, I. Broster, and A. Burns. Rewriting history to exploit gain time. In *Proc. Real-time Systems Symposium*, pages 328–335, Lisbon, Portugal, 2004. Computer Society, IEEE.

[7] G. Bernat and A. Burns. Multiple servers and capacity sharing for implementing flexible scheduling. *Real-Time Systems Journal*, 22:49–75, 2002.

[8] E. Bini, M. Di Natale, and G.C. Buttazzo. Sensitivity analysis for fixed-priority real-time systems. In *Proc. ECRTS*, pages 13–22, 2006.

[9] A. Burns. The application of the original priority ceiling protocol to mixed criticality systems. In L. George and G. Lipari, editors, *Proc. ReTiMiCS, RTCSA*, pages 7–11, 2013.

[10] G. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *IEEE Real-Time Systems Symposium*, pages 286–295, 1998.

[11] R.I. Davis and A. Burns. Robust priority assignment for fixed priority real-time systems. In *Proc. of IEEE Real-Time Systems Symposium (RTSS)*, 2007.

[12] F. Dorin, P. Richard, M. Richard, and J. Goossens. Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-Time Systems Journal*, 46(3):305–331, 2010.

[13] P. Ekberg and W. Yi. Bounding and shaping the demand of mixed-criticality sporadic task systems. In *ECRTS*, pages 135–144, 2012.

[14] N. Guan, P. Ekberg, M. Stigge, and W. Yi. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *IEEE RTSS*, pages 13–23, 2011.

[15] K. Lakshmanan, D. de Niz, and R. Rajkumar. Mixed-criticality task synchronization in zero-slack scheduling. In *IEEE RTAS*, pages 47–56, 2011.

[16] M. Neukirchner, S. Quinton, and K. Lampka. Multi-mode monitoring for mixed-criticality real-time systems. In *Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013.

[17] P. Pedro and A. Burns. Schedulability analysis for mode changes in flexible real-time systems. In *10th Euromicro Workshop on Real-Time Systems*, pages 172–179. IEEE Computer Society, 1998.

[18] S. Punnekkat, R. Davis, and A. Burns. Sensitivity analysis of real-time task sets. In *Proc. of the Conference of Advances in Computing Science - ASIAN '97*, pages 72–82. Springer, 1997.

[19] F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with FP. In *Proc. of the Euromicro Conference on Real-Time Systems*, pages 155–165, 2012.

[20] F. Santy, G. Raravi, G. Nelissen, V. Nelis, P. Kumar, J. Goossens, and E. Tovar. Two protocols to reduce the criticality level of multiprocessor mixed-criticality systems. In *Proc. RTNS*, pages 183–192. ACM, 2013.

[21] B. Sprunt, J. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Proc. 9th IEEE Real-Time Systems Symposium*, pages 251–258, 1988.

[22] 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*, DATE, pages 147–152, 2013.

[23] K. Tindell and A Alonso. A very simple protocol for mode changes in priority preemptive systems. Technical report, Universidad Politecnica de Madrid, 1996.

[24] 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.

[25] Q. Zhao, Z. Gu, and H. Zeng. HLC-PCP: A resource synchronization protocol for certifiable mixed criticality scheduling. *Embedded Systems Letters, IEEE*, PP(99), 2013.

Extending Mixed Criticality Scheduling

Tom Fleming
Department of Computer Science,
University of York, UK.
Email: tdf506@york.ac.uk

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

***Abstract*—As Mixed Criticality work has progressed it has become increasingly clear that considering only 2 levels of criticality will not suffice. Many industrial standards such as IEC 61508 and DO-178B define 4 or 5 levels, whereas the majority of current analytical approaches consider just 2. In this work we evaluate the performance of several fixed priority approaches and how they might be extended to facilitate more than 2 criticality levels. A well-established scheme, Period Transformation is also considered and extended. The effectiveness of the extensions is assessed by way of an evaluation. We show that the schemes maintain their performance relative to each other as the number of criticality levels increases.**

I. Introduction

The field of Mixed Criticality systems is advancing rapidly. Driven by industrial pressure to support new and complex hardware, attention is turning away from the more simplistic uni-processor case. However, in order to further the case for implementation, such systems must be certified and standards must be met. When considering the number of criticality levels that a system might be required to manage it becomes clear that just 2 levels will not suffice. Standards such as IEC 61508 and DO-178B define 4 or 5 Safety Integrity Levels (SIL). It is reasonable to assume that mixed criticality implementations will be required to support at least 5 levels.

Before consideration can be given to an increased number of criticality levels on complex hardware, it seems logical to begin with a single processor approach. Much of the work since Vestal's [4] seminal paper in 2007 has revolved around 2 levels of criticality. Approaches such as AMC (Adaptive Mixed Criticality), presented by Baruah et al. [2] have focused on 2 criticality levels while claiming extendability to many. We focus on single processor analysis and extend both approaches suggested in [2], AMCrtb (Response Time Bound) & AMCmax, to facilitate n potential levels. Additionally we consider Period Transformation [3] as proposed by Vestal [4]. We provide some improvements on the initial analysis and compare its performance with other approaches.

The remainder of the document is organised as follows; Section II considers the original Dual Criticality approaches, Section III considers how such approaches can be extended to include a greater number of criticality levels, Section IV provides an evaluation and Section V closes the work with some concluding remarks.

II. Dual Criticality Analysis

Baruah et al. [2] present a dual-criticality scheme known as Adaptive Mixed Criticality (AMC). AMC monitors, at runtime, the execution of each task and ensures that it remains within its budget for the current criticality level. If a job exceeds its allocated budget, a criticality change is triggered, AMC permanently suspends all tasks at the current criticality level when a change occurs. Two methods are presented in [2]:

A. AMCrtb

The original analysis presented by Baruah et al. [2] is shown in Equations (1), (2) and (3), for a sporadic task model using standard notation (C, T, D, R with $D \leq T$), LO-crit & HI-crit. There are two stages to the approach, the first is to consider the LO and HI criticality levels individually and ensure they are schedulable. The second stage is to consider the schedulability of the criticality change from LO to HI.

Stage 1A: *Check the schedulability of the LO mode for all tasks.*

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

Stage 1B: *Check the schedulability of the HI mode for HI tasks.*

$$R_i(HI) = C_i(HI) + \sum_{j \in hpH(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(HI) \quad (2)$$

Where hpH is the set of all higher priority HI criticality tasks.

The next step is to assess the schedulability of any HI criticality task executing during a criticality level change.

Stage 2A: *Calculate the schedulability of the criticality change for HI tasks.*

$$R_i^*(HI) = C_i(HI) + \sum_{j \in hpH(i)} \left\lceil \frac{R_i^*(HI)}{T_j} \right\rceil C_j(HI) + \sum_{k \in hpL(i)} \left\lceil \frac{R_i(LO)}{T_k} \right\rceil C_k(LO) \quad (3)$$

Where hpL is the set of all higher priority LO criticality tasks. $R_i(LO)$ represents a static value for higher priority but lower criticality tasks, this allows AMC to place an upper bound upon any potential low-criticality interference during a criticality change. $R_i^*(HI)$ is the value replaced into the equation with each iteration. This is appropriate due to the way in which AMC handles a criticality level change. Under

AMC, all LO criticality tasks are suspended when a criticality change occurs, as such during this time their ability to interfere with the high-criticality tasks is limited. This limit is the LO response time of the HI-crit task as after that time the system will be running in the HI criticality mode, or the task will have completed and no criticality change need occur.

B. AMCmax

There are a finite number of points in time at which a criticality change might take place. It is possible to bound these points as the criticality change must occur sometime between the start of execution, time 0 and the LO response time ($R_i(LO)$). AMCmax uses these points and seeks to determine the point at which the worst-case phasing for a HI criticality task might occur.

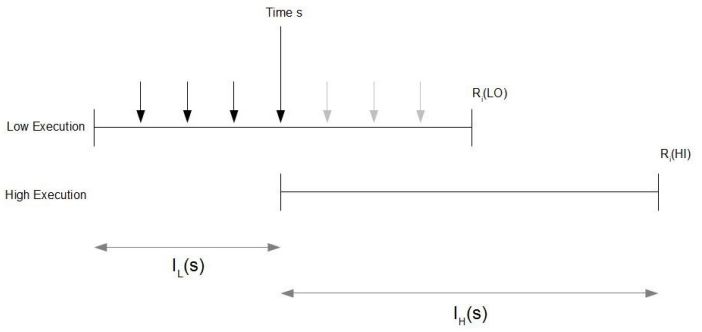


Fig. 1. Example AMCmax criticality change.

Figure 1 shows a criticality change occurring and the system moving into the HI mode. The diagram also shows the Interference suffered in both the LO and HI modes. Baruah et al. [2] illustrate this change with Equation (4), showing the calculations required to determine the response time of a HI criticality task if the change occurs at time s.

$$R_i^s(HI) = C_i(HI) + I_L(s) + I_H(s) \tag{4}$$

From Equation (4) it is easy to see the two segments of interference we must assess, I_L and I_H. These sections are also shown in Figure 1.

The technique used to assess the response time of low-criticality tasks is straightforward, it can be seen in Equation (5):

$$I_L(s) = \sum_{j \in hpL(i)} \left(\left\lfloor \frac{s}{T_j} \right\rfloor + 1 \right) C_j(LO) \tag{5}$$

The floor function is used to ensure that all tasks are accounted for immediately upon release. This algorithm is also used when calculating the LO response times of all tasks, in this case $R_i^s(LO)$ is used rather than s.

Baruah et al. [2] consider high-criticality response times $I_H(s)$. They consider the high mode as an interval of $t - s$ where $t > s$. t is the response time of the task and is the value that is replaced in each iteration. The number of releases in this interval, $t - s$, for a HI criticality task τ_k, can be calculated:

$$\left\lceil \frac{t-s}{T_k} \right\rceil + 1$$

This can be extended for cases where $D_k < T_k$:

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

The full calculation is shown in Equation (6) presented in the form of a function M. With input parameters k, s and t, where k is the task, s is time s and t is time t (or the response time replaced into the equation).

$$M(k,s,t) = min\left\{ \left\lceil \frac{t - s - (T_k - D_k)}{T_k} \right\rceil + 1, \left\lceil \frac{t}{T_k} \right\rceil \right\} \tag{6}$$

Equation (6), accounts for all completions of task τ_k, within the interval $s \ldots R_i(HI)$. Rare cases are possible where the calculation is overly pessimistic, the function ensures that the value returned is no greater than the total number of releases.

The number of releases in the LO criticality mode is easily calculable by removing the results of Equation (6) from the total number of releases.

$$\left(\left\lceil \frac{t}{T_k} \right\rceil - M(k,s,t) \right) C_k(LO)$$

Therefore $I_H(s)$ is:

$$I_H(s) = \sum_{k \in hpH(i)} \left\{ (M(k,s,t) C_k(HI)) + \left(\left(\left\lceil \frac{t}{T_k} \right\rceil - M(k,s,t) \right) C_k(LO) \right) \right\} \tag{7}$$

And thus the full equation:

$$R_i^s = \sum_{j \in hpL(i)} \left(\left\lfloor \frac{s}{T_j} \right\rfloor + 1 \right) C_j(LO) + \sum_{k \in hpH(i)} \left\{ (M(k,s,R_i^s) C_k(HI)) + \left(\left(\left\lceil \frac{R_i^s}{T_k} \right\rceil - M(k,s,R_i^s) \right) C_k(LO) \right) \right\} \tag{8}$$

And:

$$R_i = max(R_i^s) \forall s$$

Finally they look at which points of s, within $0 \ldots R_i(LO)$ require consideration. Baruah et al. [2] note that the amount of low-criticality interference increases (as a step function), as the value of time s increases. Similarly the high-criticality interference decreases as the low increases. Therefore the response time changes only at the release of a low-criticality job, thus we can limit our search to points of s where a LO criticality job is released.

C. Period Transformation

Vestal [4] proposed a Mixed Criticality Period Transformation (PT) approach. He used PT, not to create a harmonic task set, but to allow for a Criticality and Rate Monotonic based priority ordering. The approach proposes that only those HI criticality tasks with periods greater than that of the shortest

LO criticality period be transformed. This will allow all HI criticality tasks to attain a higher priority than the LO and thus avoid the problem of criticality inversion and allow for criticality monotonic assignment.

This gives us 3 groups of tasks. Those of a LO criticality, these do not need transformation. Those with a HI criticality but a period shorter than the shortest LO criticality task, these do not need transformation. Finally those with a HI criticality with a period greater than that of the shortest LO criticality task, these are the tasks that must be transformed.

The analysis of HI criticality tasks, in the HI mode is calculated via standard response time techniques [1]. The analysis for the LO criticality mode is detailed as follows:

Tasks are transformed by a factor, m.

$$m = \left\lceil \frac{T_j}{T_l} \right\rceil$$

Where τ_l is the LO criticality task with the shortest period and τ_j is a HI criticality task that must be transformed.

At runtime, transformed tasks are expected to execute up to their $C_j(HI)/m$ until they reach their untransformed, $C_j(LO)$, only then can we determine if a task will overrun its LO execution bounds and a mode change would need to occur. Transformed tasks, running in the LO mode execute in $C_j(HI)/m$ time slices until $C_j(LO)$.

The number of transformed dispatches that might interfere with τ_i could be calculated as follows:

$$\left\lceil \frac{R_i}{T_j/m} \right\rceil$$

The number calculated above will contain several complete executions of $C_j(LO)$ and a remainder, this remainder can execute for no longer than $C_j(LO)$, Vestal assumes this value. He calculates the number of transformed executions which complete to $C_j(LO)$:

$$\left\lfloor \frac{R_i}{T_j} \right\rfloor C_j(LO)$$

So including the added pessimism of those transformed executions that do not complete, the total interference from τ_j can be summed as follows.

$$\left\lfloor \frac{R_i}{T_j} \right\rfloor C_j(LO) + C_j(LO)$$

Clearly there are several disadvantages to Vestal's technique. The use of $C_j(LO)$ to account for transformed releases which do not constitute an entire $C_j(LO)$ is overly pessimistic. Vestal's approach also loses one of the key properties of Period Transformation, the ability to create harmonic task sets and, by proxy, the ≤ 1 utilisation bound for RM schedulability. Although not all tasks are transformed it is likely that by transforming the HI tasks the number of context switches will increase significantly. This coupled with the need for additional run-time monitoring causes PT overheads to remain high.

III. Extend to n Criticality Levels

A. AMCrtb

When considering n possible criticality levels for AMCrtb we re-examine the two stages of the analysis. In stage one we examine each level, up to n, and determine schedulability. In stage two we consider the feasability of $n-1$ criticality level changes.

1) Stage One: Consider a system containing 5 distinct criticality levels, $L1 \dots L5$ where $L1 > L5$. The analysis for $L5$ must consider the potential interference of all higher priority tasks, regardless of criticality level (as $L5$ is the lowest level). To calculate the interference suffered from higher priority $L4$ tasks we use the following term:

$$\sum_{j \in hp(i) | L_j = L4} \left\lceil \frac{R_i(L5)}{T_j} \right\rceil C_j(L5)$$

The algorithm looks for those higher priority tasks, τ_j, where the criticality level (L_j) is equal to $L4$. This considers any interference suffered from a task at $L4$, but uses their $L5$ values. The calculation can be completed to account for levels $L3 \dots L1$ as shown in Equation (9).

$$\begin{aligned} R_i(L5) = C_i(L5) + & \sum_{j \in hp(i) | L_j = L4} \left\lceil \frac{R_i(L5)}{T_j} \right\rceil C_j(L5) \quad + \\ & \sum_{k \in hp(i) | L_k = L3} \left\lceil \frac{R_i(L5)}{T_k} \right\rceil C_k(L5) \quad + \\ & \sum_{l \in hp(i) | L_l = L2} \left\lceil \frac{R_i(L5)}{T_l} \right\rceil C_l(L5) \quad + \\ & \sum_{m \in hp(i) | L_m = L1} \left\lceil \frac{R_i(L5)}{T_m} \right\rceil C_m(L5) \end{aligned} \tag{9}$$

This process is repeated for each of the remaining criticality levels to check their schedulability. It is possible to generalise these equations to one that can deal with $2 \longrightarrow n$ criticality levels. We must consider the schedulability of n criticality levels individually.

For each criticality level.

$$\forall L \in 1 \dots n$$

For all tasks where the criticality level is greater than or equal to L.

$$\forall \tau_i | L_i \geq L$$

Calculate the response times for that level.

$$R_i(L) = C_i(L) + \sum_{j \in hp(i) | L_j \geq L} \left\lceil \frac{R_i(L)}{T_j} \right\rceil C_j(L) \tag{10}$$

Equation (10) considers the response time of task τ_i at criticality level L by accounting for any interference from higher priority tasks with a criticality level greater than or equal to L. This test is repeated for each of the n criticality modes. Equations (1) and (2) are the dual criticality application of Equation (10).

2) Stage Two: When assessing the interference suffered during a criticality change we must consider two groups of tasks. The first group are those tasks of a higher priority and with a criticality level greater than or equal to the task in question. This has been considered in Stage One. The second group are those tasks with a higher priority but a lower criticality level. It is clear that under AMC, those tasks with a higher priority but lower criticality will have a bounded effect on a higher criticality task if a criticality change occurs.

The interference suffered by higher criticality task, τ_i from higher priority but lower criticality task, τ_k during a criticality change is bounded by τ_i's response time at τ_k's criticality level, $R_i(L_k)$.

$$\sum_{k \in hp(i) | L_k < L_i} \left\lceil \frac{R_i(L_k)}{T_k} \right\rceil C_k(L_k)$$

For all tasks with a higher priority than τ_i where the criticality level is lower. $R_i(L_k)$ is the response time of τ_i at the criticality level of τ_k. These values are static bounds and do not change upon each iteration.

If we combine the analysis for the higher priority tasks with a criticality level greater than or equal to L_i and the analysis for the higher priority tasks with a criticality level less than L_i we can produce an algorithm to assess the feasibility of the criticality level changes in a system. In a system with n criticality levels we must consider $n-1$ criticality level changes.

For each criticality level.

$$\forall L \in 1 \ldots n$$

For all tasks where the criticality level is greater than or equal to L

$$\forall \tau_i | L_i \geq L$$

Beginning at the lowest criticality level, calculate the schedulability of each criticality change.

$$R_i^*(L) = C_i(L) + \sum_{j \in hp(i) | L_j \geq L} \left\lceil \frac{R_i^*(L)}{T_j} \right\rceil C_j(L) \quad + \sum_{k \in hp(i) | L_k < L_i} \left\lceil \frac{R_i(L_k)}{T_k} \right\rceil C_k(L_k) \tag{11}$$

Equation (11) assesses the schedulability of criticality changes for $2 \longrightarrow n$ criticality levels. This combined with the algorithm in Equation (10) provides an AMCrtb schedulability test generalised to n levels of criticality.

B. AMCmax

Consider the case of two criticality levels A and B, where A is the lowest criticality level in the system, (B > A). We would use the dual criticality analysis shown in Equation 8.

In order to determine the response time of a level B task, AMCmax considers points of s where the criticality change might occur. These points are bounded by the $R_i(A)$ response time of the task in question. If this system were also to include a criticality level C, such that $C > B > A$ then a criticality change might occur at any point (s_2) between the original change from A to B, point s_1, and the task's response time in criticality mode B, $R_i(B)$.

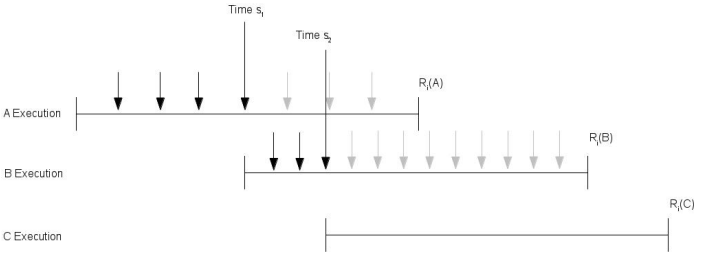


Fig. 2. The system with modes A and B with an additional level, C added.

To calculate the interference suffered between two points of s we define a new function, N:

$$N(k, s_1, s_2) = \left\lfloor \frac{s_2 - s_1 - (T_k - D_k)}{T_k} \right\rfloor + 1 \tag{12}$$

Function N provides the interference suffered between times s_1 and s_2. In this case, this function is used to assess the response time of tasks at criticality level B.

The calculation for the A criticality tasks is similar to the LO calculation, we use s_1 rather than s in order to differentiate between criticality changes.

$$\sum_{j \in hpA(i)} \left(\left\lfloor \frac{s_1}{T_j} \right\rfloor + 1 \right) C_j(A)$$

The calculation for the B criticality tasks changes to make use of the function N (see Equation (12)) as it is now used to determine the interference suffered between two points of s. The criticality A interference is calculated by removing the number of releases, as calculated by function N from the total number of releases.

$$\sum_{k \in hpB(i)} \left\{ N(k, s_1, s_2) C_k(B) + \left(\left\lceil \frac{s_2}{T_k} \right\rceil - N(k, s_1, s_2) \right) C_k(A) \right\}$$

Finally we may consider the calculation for criticality level C. Functions M and N are used in order to calculate the interference a criticality C task might suffer in modes C and B respectively. Both functions M and N are removed from the total number of releases to calculate the criticality A response time.

$$\sum_{l \in hpC} \left\{ M(l, s_2, R_i(C)) C_l(C) + N(l, s_1, s_2) C_l(B) + \left(\left\lceil \frac{R_i(C)}{T_l} \right\rceil - N(l, s_1, s_2) - M(l, s_2, R_i(C)) \right) C_i(A) \right\}$$

Where hpC refers to all higher priority tasks of criticality level C. The complete calculation for $\tau_i(C)$ is shown in Equation

(13).

$$R_i(C) = C_i(C) + \sum_{j \in hpA(i)} \left(\left\lfloor \frac{s_1}{T_j} \right\rfloor + 1 \right) C_j(A) + \sum_{k \in hpB(i)} \left\{ N(k, s_1, s_2) C_k(B) + \left(\left\lceil \frac{s_2}{T_k} \right\rceil - N(k, s_1, s_2) \right) C_k(A) \right\} + \sum_{l \in hpC} \left\{ M(l, s_2, R_i(C)) C_l(C) + N(l, s_1, s_2) C_l(B) + \left(\left\lceil \frac{R_i(C)}{T_l} \right\rceil - N(l, s_1, s_2) - M(l, s_2, R_i(C)) \right) C_i(A) \right\} \quad (13)$$

Where:

$$R_i(C) = max(R_i(C)) \forall s_n$$

Equation (13) shows how $R_i(C)$ can be calculated by considering points of s_1 where the change from A to B might occur and points of s_2 where the change from B to C might occur.

If we were to extend this system to introduce a 4th criticality level, D, we would follow the same steps as we did for criticality level C. Consider points for the criticality change from C to D at time s_3 bounded by the criticality C response time, $R_i(C)$. It is important to note that the function N is always used to calculate the number of releases between two points of s and the function M is always used to calculate the response time at the highest criticality level currently being considered.

The process of adding another set of points to check for each criticality level may be repeated to account for as many criticality levels as desired. However, the computational load increases almost exponentially as the number of criticality levels increases.

C. Period Transformation

In his analysis, Vestal [4] assumes a value of $C_j(LO)$ for the remaining transformed executions, $C_j(HI)/m$ that do not constitute a complete execution of $C_j(LO)$. This value, although an effective upper bound, is undesirably pessimistic. The work below considers a more accurate approach to finding the interference from transformed executions that do not constitute a complete untransformed execution.

We calculate the number of complete executions of $C_j(LO)$ that might interfere with τ_i:

$$\left\lfloor \frac{R_i}{T_j} \right\rfloor C_j(LO)$$

Rather than assuming the value of $C_j(LO)$ for all remaining transformed executions, we seek to determine the size of the incomplete interval. To find the size of the remaining interval, P, we do the following:

$$P = R_i - \left\lfloor \frac{R_i}{T_j} \right\rfloor T_j$$

And thus we use the value P, to calculate the number of transformed executions within the remaining interval, x:

$$x = \left\lceil \frac{P}{T_j/m} \right\rceil \frac{C_j(HI)}{m}$$

Therefore the complete calculation will include the transformed tasks within the incomplete interval and the complete executions of $C_j(LO)$. This is shown in Equation [14].

$$min\{x, C_j(LO)\} + \left\lfloor \frac{R_i}{T_j} \right\rfloor C_j(LO) \quad (14)$$

The interference suffered from the transformed tasks within the incomplete interval will be the minimum of x or $C_j(LO)$.

In keeping with the nature of this work we then considered how the more accurate analysis presented above might be adapted to work in a system with more than two criticality levels. The analysis itself is applicable with little alteration.

The number of complete executions of $C_j(L_i)$ within the interval can be calculated.

$$\left\lfloor \frac{R_i}{T_j} \right\rfloor C_j(L_i)$$

And therefore we calculate the interference in the remaining time period from the transformed executions.

$$P = R_i - \left\lfloor \frac{R_i}{T_j} \right\rfloor T_j$$

$$x = \left\lceil \frac{P}{T_j/m} \right\rceil \frac{C_j(L_j)}{m}$$

The complete calculation for n criticality levels is shown in Equation [15].

$$min\{x, C_j(L_i)\} + \left\lfloor \frac{R_i}{T_j} \right\rfloor C_j(L_i) \quad (15)$$

As can be seen, the analysis is almost directly applicable. The key challenge is the transformation of the tasks in such a way that a criticality monotonic order is created.

Rather than transforming all higher criticality tasks, with a period greater than the shortest period of any LO task, the process for n criticality levels must be iterative. Consider Table I where $HI > ME > LO$.

	T	L
τ_1	80	HI
τ_2	110	ME
τ_3	100	LO

TABLE I
3 CRITICALITY LEVEL PT EXAMPLE, UNTRANSFORMED

Of the three tasks shown in Table I, τ_2 is the only one requiring transformation as it has a period greater than that of τ_3. We can calculate the transformation factor, n, as follows:

$$m = \left\lceil \frac{110}{100} \right\rceil$$

Thus m = 2, this will give τ_2 a transformed period of 55, less than the period of τ_1. The set is not criticality monotonic and, as such, it is clear that this calculation will not suffice. Rather τ_1 must also be transformed to give it a period of 40.

IV. Evaluation

We tested the algorithms above on randomly generated task sets. The task sets were generated the same way as in [2]. We generated 5000 sets of 10 tasks per 2% total utilisation. As MC tasks might have multiple WCETs, we consider the WCET value generated for each task to be at the highest criticality level in the system. The lowest level in the system is half the highest, each additional level is evenly distributed between the highest and lowest.

Our experimental data uses Criticality Dependant Utilisation to assess the percentage of tasks schedulable at any particular utilisation.

$$U_i(L_i) = \frac{C_i(L_i)}{T_i} \tag{16}$$

In order to provide a thorough comparison we considered the performance of AMCrtb, AMCmax and PT (ignoring the inherent overheads). Alongside SMC [2], SMC-NO(Vestal's) [4] and Criticality Monotonic ordering (CrMPO).

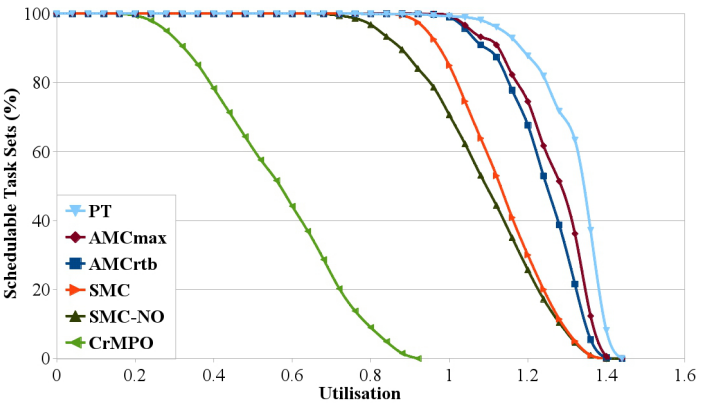


Fig. 3. Two Criticality Levels

Figure 3 shows the performance, at two criticality levels, of each technique at varying utilisations (Criticality Dependant). It is clear that AMCrtb out-performs SMC, CrMPO and SMC-NO. AMCmax performs slightly better than AMCrtb and PT performs well if overheads are ignored.

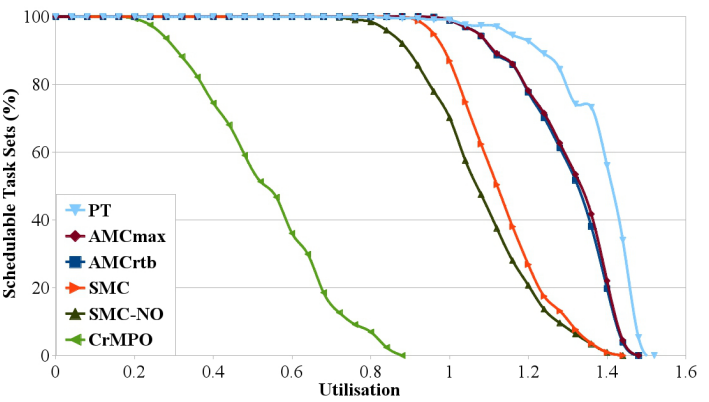


Fig. 4. Three Criticality Levels

As the number of criticality levels is increased each of the algorithms, apart from PT, maintain their performance relative to each other. Period Transformation's performance degrades due to the increasing complexity of the transformations required as the number of criticality levels is increased. Figures 3, 4 and 5 are typical of the relative results obtained for task sets with different characteristics.

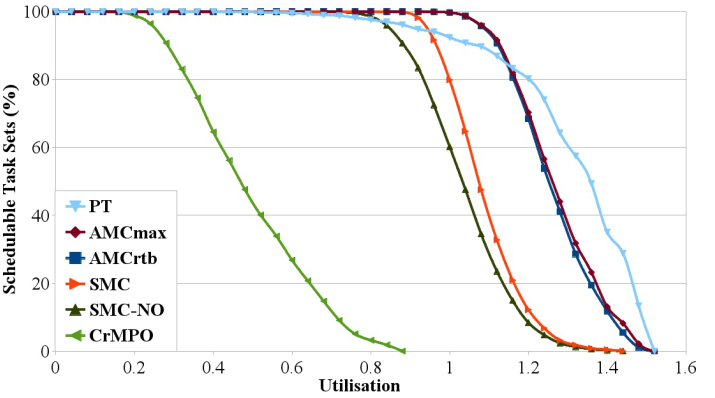


Fig. 5. Five Criticality Levels

V. Conclusion

It is clear that both AMC methods perform well compared to other techniques as the number of criticality levels is increased. Importantly both AMCrtb and AMCmax maintain their strong dominance over SMC. Although AMCmax does out-perform AMCrtb, AMCrtb remains an excellent approximation of AMCmax while keeping computational costs relatively low. As Mixed Criticality systems move forward it may become important to support even greater than 5 levels of criticality. In this case AMCrtb's relatively low computational cost would allow its application.

Period Transformation performs as expected. It provides a schedulability boost and avoids criticality inversion at the cost of vastly increased overheads and the requirement for additional runtime monitoring.

In short, AMCrtb remains dominant over SMC and provides a good approximation of AMCmax at any number of criticality levels. In practice AMCrtb will out perform PT.

References

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

[2] S. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 34 –43, 29 2011-dec. 2 2011.

[3] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *RTSS*, pages 181–191, 1986.

[4] 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*, pages 239 –243, dec. 2007.

On the Expressiveness of Fixed-Priority Scheduling Contexts for Mixed-Criticality Scheduling

Marcus Völp*†
*School of Computer Science, Logical Systems Lab
Carnegie Mellon University
Pittsburgh, PA, USA
mvoelp@cs.cmu.edu

Adam Lackorzynski†, Hermann Härtig†
†Institute for Systems Architecture, Operating Systems Group
Technische Universität Dresden
Dresden, Germany
{voelp, adam, haertig}@os.inf.tu-dresden.de

***Abstract*—Scheduling contexts allow flattening hierarchical schedules in virtualized mixed-criticality setups. However, their expressiveness in terms of supported higher-level scheduling algorithms is not yet well understood. This paper makes a first step in this direction by investigating how recently proposed mixed-criticality algorithms can be mapped to fixed-priority scheduling contexts and how scheduling contexts can be extended to support these algorithms. We found that although the initial implementation of scheduling contexts was rather limited, a few practically feasible extensions broadened their applicability to all investigated algorithms.**

I. Introduction

In 2005, Steinberg, Wolter and Härtig [1] introduced scheduling contexts as an elegant and simple way of implementing priority inheritance in microkernel-based systems. In these systems the majority of resources are threads executing in application-level servers. The basic idea is to schedule time quanta (described through scheduling contexts) instead of the threads or jobs that correspond to them. This way, inheritance is simply a matter of activating the recipient thread whenever the quanta is selected.

The primary purpose of scheduling contexts was to improve imprecise [2] and quality-assuring scheduling [3]. However, we have seen examples that make use of scheduling contexts in a much more general way, even when not donating them to other threads. For example, Lackorzynski et al. [4] demonstrated the flattening of hierarchical mixed-criticality schedules in virtualized guest operating systems by exporting part of the internal task structure to the hypervisor and by assigning guests multiple scheduling contexts to choose from. But how general are scheduling contexts and how can they be extended to become more general?

As a first step in this direction, this paper investigates how mixed-criticality scheduling can be mapped to a scheduling-context-based fixed-priority scheduler in the hypervisor.

Mixed-criticality scheduling [5] seeks to consolidate tasks of different importance (or criticality) into the same system. Naturally, because tasks of higher criticality may cause more severe damage when late, their analysis is taken more seriously and results in more pessimistic worst-case execution time (WCET) estimates. Mixed-criticality scheduling is about granting all tasks (lower and higher criticality) their optimistic estimates while guaranteeing the completion of higher criticality tasks in the exceptional case where one of the more optimistic WCET estimates ceases to hold.

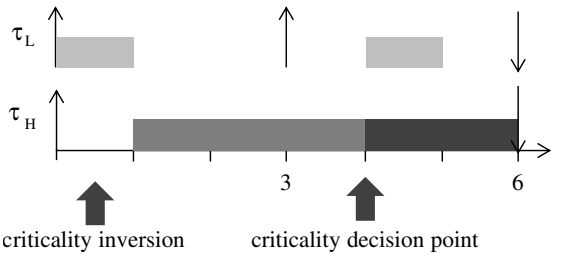


Fig. 1. Schedule of $\tau_L = (LO, 3, 3, (1, -)^T)$ and $\tau_H = (HI, 6, 6, (3, 2)^T)$. Shown is the criticality inversion of $\tau_{LO,1}$ and the criticality decision point after $\tau_{HI,1}$ received 3 time units.

As a side-effect of guaranteeing up to the optimistic WCET estimates for all tasks in case all higher criticality jobs complete within low bounds, a particularly puzzling situation called *criticality inversion* may occur. Figure 1 gives an example of such a situation for the two tasks τ_L and τ_H. Here, and in the following, we denote the release of a job with an upward arrow and its absolute deadline with a downward arrow. Darker colors are used to mark the *excess budget* of a task, that is the difference between the WCET estimate for the higher criticality level and for the respective next lower. If we had given τ_H priority over the first job of τ_L, the low criticality job could miss its deadline if τ_H executes longer than two time units. Latest at time 4, after τ_H received 3 time units, we know whether τ_H exceeds its low WCET estimate. If so, the scheduler drops the second job of τ_L and relocates its resources to τ_H in order to guarantee the completion of high criticality tasks in all situations. Otherwise, if τ_H stops within the low bounds, sufficient time remains to complete $\tau_{LO,2}$. We call the point in time after which τ_H received its low budget a *criticality decision point* of τ_H.

The contribution of this paper is an analysis of situations like the one in Figure 1 to identify whether and how mixed-criticality schedulers can be mapped to a configuration of scheduling contexts. More precisely, we shall look at such mappings for the mixed-criticality schedulers: criticality monotonic (CrMPO) [5], [6], own-criticality based priority (OCBP) [5], [6] and several static mixed criticality variants (SMC-*) [7], adaptive mixed criticality (AMC-*) [7], and earliest deadline first with virtual deadlines (EDF-VD) [8].

After formally introducing mixed-criticality tasksets, their feasibility criterion and scheduling contexts, we exemplify how such a mapping works by reciting some of the results from flattening. In Section III, we return to the question how to map the above schedulers to a fixed-priority scheduler with multiple scheduling contexts per task. Section IV reviews related attempts. Section V summarizes what we achieved and shows directions where to go from here.

II. Mixed Criticality, Scheduling Contexts and Flattening

A. Mixed Criticality

Although mixed-criticality scheduling is not limited to sporadic tasks, let us focus in this paper on this specific type of tasksets.

Let $l_i \in L$ be the criticality level of the sporadic task τ_i drawn from the totally ordered set of criticality levels L. We characterize τ_i by the tuple $\tau_i = (l_i, \delta_i, P_i, C_i)$ where δ_i is the deadline relative to the release $r_{i,j}$ of τ_i's current job $\tau_{i,j}$ and P_i is the minimal interrelease time. We assume constrained task sets, that is, $\delta_i \leq P_i$. The vector C_i denotes for each criticality level l the WCET estimate $C_i(l)$ at this level. We assume WCET estimates are monotonically increasing with increasing criticality level and constrain C_i by $C_i(l) \leq C_i(h)$ for every $l \leq h$. We generally assume that schedulers enforce the execution budgets they grant. That is, once a job exceeds $C_i(l_i)$, the scheduler will reclaim all resources assigned to it (in our case CPU-time). It is therefore safe to set $C_i(h) = C_i(l_i)$ for all criticality levels $h \geq l_i$. The feasibility criterion for mixed-criticality schedulers is:

Definition 1 (MC-Schedulability): A task set T is mixed-criticality schedulable if all jobs $\tau_{i,j}$ receive $C_i(l_i)$ time units in between $r_{i,j}$ and $r_{i,j} + \delta_i$ provided all jobs of higher criticality tasks τ_h complete before $C_h(l_i)$.

From Baruah and Vestal [9] we know that sporadic task sets are MC-schedulable if they are MC-schedulable at their synchronous arrival sequence. In this sequence, the first job of all tasks arrives at time 0 and subsequent jobs follow P_i apart. Also we know from Baruah et al. [10] that OCBP is optimal among the class of fixed job-priority algorithms.

In this paper, we shall use the terms *job* and *task* (i.e., sequence of jobs) to refer to the entities considered by mixed-criticality schedulers and scheduling contexts (*SC*) and *thread* when we talk about mapping tasks and jobs to a SC-based scheduler. We will introduce SCs in the next paragraph. We shall also write $\hat{x}$ to distinguish SC parameters from task parameters, which we denote by simple variables x. We assume that threads signal completion after they finish executing a job and that they then wait for a signal indicating the release of the next job.

B. Scheduling Contexts

In the following we introduce scheduling contexts (SC) and exemplify their use in previous work [4]. SCs are basic operating system primitives that are used for driving scheduling decisions. Traditionally, operating systems keep scheduling information (such as a thread's priority or budget) and all other thread-specific information (such as the thread's user-level register content) in the same data structure called thread control block (TCB). SC-based systems refactor TCBs into two data structures: the scheduling context (SC), which keeps all scheduling information plus the pointers to be linked into the ready queue, and the execution context (EC), which keeps all remaining state that is required to execute the thread.

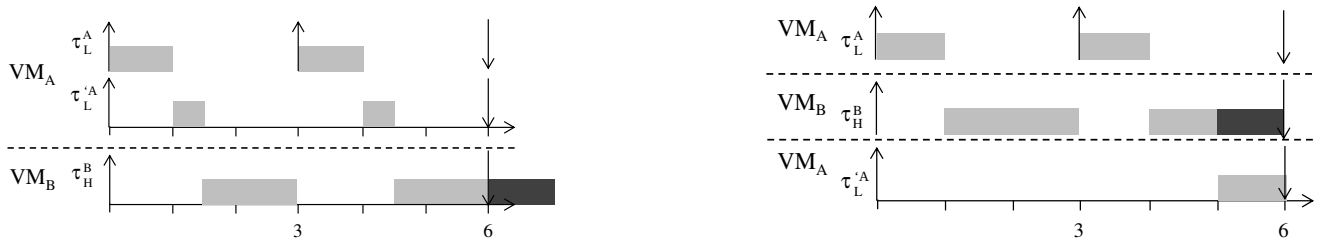
Now by separating SCs and ECs, the first limitation of TCBs that can easily be dropped is that threads can have no more than one time quantum. For now we regard a time quantum as a guarantee of the scheduler to provide CPU time up to a given budget $\hat{C}_i$ (typically set to the task's WCET C_i) every $\hat{P}_k$ time units whenever there is no thread that is currently consuming higher prioritized time. Imprecise computations [2] and quality assurance scheduling [3] made use of this option to prioritize the mandatory work of all threads over optional parts such as filters and other video post-processing steps, which improve the result. The mechanism that is required to implement these scenarios is the ability to switch between SCs. That is, a thread executing on an SC must be able to select another SC, possibly discarding the remaining budget of the former.

A second application becomes possible by allowing also incoming signals to choose SCs. When scheduling the virtual CPUs (vCPUs) of multiple virtual machines (VM) one has to frequently activate each to preserve the impression of reactiveness although the VM may be off focus and running non-interactive background load. By assigning one SC with a small high priority time quantum and a second SC with a larger quantum but lower priority, VMs can react quickly to incoming signals activating the high priority SC and dropping down to the low priority SC for non-interactive tasks or in other situations determined by the VM-internal scheduler. Running applications of different importance in a VM turns this setup into a mixed-criticality system.

C. Flattening: Exemplifying SCs for Mixed Criticality

To obtain a deeper understanding how SCs can express mixed-criticality systems and in particular to clarify some of the properties we like to preserve when extending SC-based scheduling, let us repeat some of the results from flattening hierarchical mixed-criticality scheduling.

When considering VMs, time-slicing multiple vCPUs on one physical CPU is a typical approach to progress tasks scheduled by the VMs' internal schedulers. However, when VMs are mixed-criticality, time-slicing ceases to work. In [4], we presented an example similar to the one depicted in Figure 2(a). Assume two VMs (VM_A and VM_B) run the two low criticality tasks $\tau_L^A = (LO, 3, 3, (1, 1)^T)$ and $\tau_L'^A = (LO, 6, 6, (1, 1)^T)$ in VM_A and the high criticality task $\tau_H^B = (HI, 6, 6, (3, 4)^T)$ in VM_B on the same physical CPU. It is easy to see that there is no single time quanta such that VM_A's and VM_B's internal scheduler can guarantee that all tasks meet their deadlines. Because $C_H(LO) = 3$, we have to prioritize τ_L^A over τ_H^B and invert criticality. But then, if we assign a single budget of $\hat{C}_A = 1.5$ time units every $\hat{P}_A = 3$ (or even a budget of 1 for the first 3 time units and of 2 for the second 3), VM_B can no longer guarantee the completion of τ_H^B. However, if like in Figure 2(b) we assign two distinct time quanta to VM_A by



(a) without SCs and flattening (b) with flattening

Fig. 2. Mixed-criticality schedule of the tasks $\tau_L^A = (LO, 3, 3, (1, 1)^T)$ and $\tau_L^{'A} = (LO, 6, 6, (1, 1)^T)$ in VM_A and $\tau_H^B = (HI, 6, 6, (3, 4)^T)$ in VM_B.

linking to it one scheduling context SC_A^1 with $\hat{C}_A^1 = 1$ every $\hat{P}_A^1 = 3$ and a second one SC_A^2 with $\hat{C}_A^2 = 1$ every $\hat{P}_A^2 = 6$, the taskset becomes MC schedulable. The priority ordering for this to work is that SC_A^1 is higher prioritized than SC_B and SC_A^2 lower than SC_B.

In Figure 2(b), the scheduler inside VM_A used the budget of SC_A^1 to run τ_L^A and SC_A^2 to run $\tau_L^{'A}$. However, it is also possible for VM_A to use SC_A^1 at time 3 to complete $\tau_L^{'A}$ and leave SC_A^2 for τ_L^A's second job. There are two points why this choice is important: First, although we will not further follow this direction in this paper, we would like to allow VMs to hide as much of their internal structure as possible. That is, VM internal schedulers should be able to give guarantees to the VM internal threads without having to expose all these threads to the hypervisor scheduler, which in fact may be different from the internal scheduler. However, unlike typical hierarchical schedulers (see e.g., Regehr and Stankovic [11] or Zhang and Burns [12]), SCs allow nested schedulers to work with more than just a single time quantum.

The second important point is that we seek to build our systems such that guarantees are robust against failures in tasks, nested schedulers or even the entire VM. As a consequence, any operation that a VM or thread may perform on its assigned SCs must not violate the timing guarantees offered to other threads. Notice, we do not extend this control to the thread as a whole because SCs provide us external control over the timing behavior of a thread without having to worry about the remainder of its state. An execution context without an SC simply will never get the CPU.

III. Mixed-Criticality Scheduling with Scheduling Contexts

We now turn our attention to the mapping of mixed criticality schedulers to SCs.

A. Criticality-Monotonic and Static Fixed Task-Priority Algorithms

The initial implementation described in Steinberg et al. [1] equipped SCs with just a budget $\hat{C}_i$, a period $\hat{P}_i$ and a priority π_i. However, they already allowed restricting their release to the point in time when both an inter-process communication message was received and $\hat{P}_i$ time units have passed since the last release. SCs were scheduled by a fixed-priority scheduler and only the single highest prioritized active SC was kept in the scheduler's ready queue.

It is easy to see that such a setting can support implicit deadline constrained sporadic tasksets (i.e., tasks with $\delta_i = P_i$) and static-priority mixed-criticality schedulers. A scheduler is static-priority if it assigns a fixed priority to each task and then relies on this priority (and the enforcement of deadlines and budgets) to ensure MC schedulability. Each task τ_i is assigned exactly[1] one SC with $\hat{C}_i = C_i(l_i)$ and $\hat{P}_i = P_i$.

Prominent examples of such a mixed-criticality scheduler are the fixed task-priority instances of the Criticality Monotonic Priority Ordering (CrMPO) [6] family. CrMPO assigns priorities such that higher criticality tasks are strictly higher prioritized than lower criticality tasks. Within the priority strips of equally critical tasks, priorities are assigned following a standard scheduling algorithm such as rate or deadline monotonic.

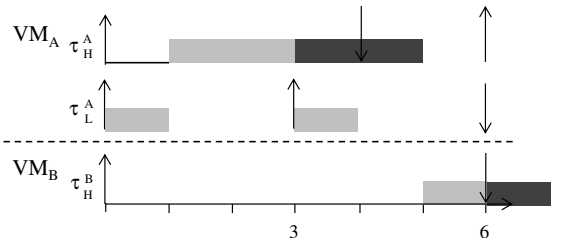
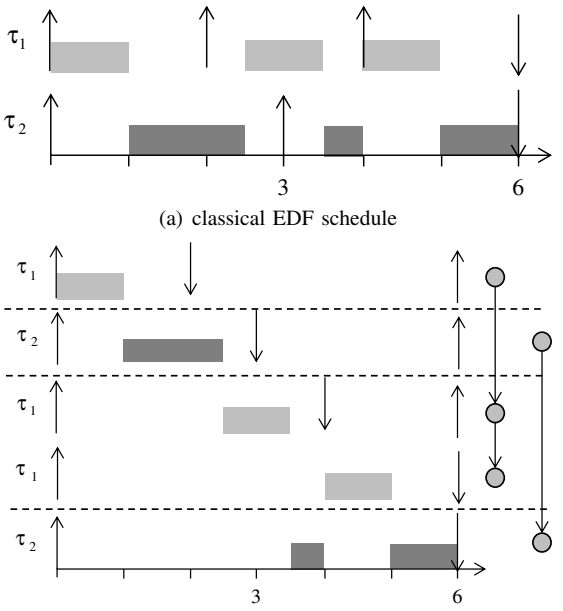


Fig. 3. Deadline overrun with side effect to task in other VM.

Standard SCs are not equipped for enforcing deadlines of constrained tasksets. For example, the taskset comprised of $\tau_H^A = (HI, 4, 6, (2, 4)^T)$, $\tau_L^A = (LO, 3, 3, (1, 1)^T)$ and $\tau_H^B = (HI, 6, 6, (1, 2))$ is MC-feasible (e.g., with $\pi_H^A > \pi_L^A > \pi_H^B$). However, a bug in τ_H^A causing a late start after its release may result in τ_H^A overrunning its deadline and in turn τ_H^B missing its deadline. Figure 3 illustrates this point.

To support constrained deadline sporadic tasks, our first extension to SCs is a deadline $\hat{\delta}_i$ up to which budgets must have been consumed to not interfere with other tasks in the way we have just seen. Because our scheduler enforces budgets by setting a timeout to the remaining budget $\hat{C}_{i,remaining}$, enforcing deadlines comes almost for free. Instead of setting this timeout to $t + \hat{C}_{i,remaining}$ when switching at time t to $\tau_{i,j}$, we set it to $\min(t + \hat{C}_{i,remaining}, r_{i,j} + \hat{\delta}_i)$ where $r_{i,j}$ is the release of $\tau_{i,j}$. The only situation where constrained

[1] In case of hierarchical scheduling, multiple tasks of the same VM with adjacent priorities and the same periods could be consolidated to the same SC. However, as already mentioned, this line of argumentation is out of the scope of this paper.



(a) classical EDF schedule

(b) expansion with per-task SC queues containing one SC per job

Fig. 4. Expansion of EDF schedule using SC queues and deadline-enforcing SCs

deadline enforcement causes a scheduling overhead not present in budget enforcing schedulers is when the activation signal of the next sporadic job $\tau_{i,j+1}$ arrives before $r_{i,j}+P_i$. In this case we have to set a second timer to $r_{i,j}+P_i$, which will always fire.

B. Own-Criticality Based Priority Ordering and Static Fixed Job-Priority Algorithms

The extension to deadline-enforcing SCs and a feature that was already present and required for imprecise computing [2] and quality-assuring scheduling (QAS) [3] brings us a big step closer to fixed job-priority algorithms such as own-criticality based priority ordering [6].

In QAS, the idea is to drop to a lower priority once the important part of work has completed. This gives other threads the chance of completing their important work before continuing to increase the quality of the result. The SC mapping therefore involves one SC for the mandatory part of the work, which must be completed, plus one additional SC for each optional, quality improving step. Once a thread completes its mandatory work, it steps ahead to the next SC, which in the course discards the budget of the previous SC and enables the new budget.

Although more efficient solutions exist for greedy fixed job-priority algorithms such as EDF, it is "almost" possible to use SCs and a fixed SC-priority scheduling algorithm to implement arbitrary static fixed job-priority algorithms. Let us first explain the idea and then fix the "almost" in the above statement.

Figure 4 gives an example for EDF with a classic (i.e., single-criticality) taskset. Rather than using SCs to describe a single recurring release of a task, we use it to describe the release of a single job in the hyperperiod HP of the entire taskset. A task τ_i executes $n = \frac{HP}{P_i}$ jobs in a hyperperiod. For each task, we instantiate n SCs and combine them into a list in the order of the release of these jobs. The parameters of these SCs are $\hat{P}_i = HP$ and $\hat{C}_i = C_i$ for all SCs and $\delta_{i,j} = P_i \cdot j$ for the SC representing the j^{th} job $\tau_{i,j}$ ($1 \leq j \leq n$). We set the priority of the SCs according to the fixed job-priority algorithm. For EDF, these are $\pi_{o,p} > \pi_{q,r}$ whenever $r_{o,p} + \delta_o \leq r_{q,r} + \delta_q$ (breaking ties if necessary).

Because SCs are ordered in lists, a thread executing on the list of SCs will discard the remaining budget prior to executing the next job. Also, no thread can extend the budget received for executing a job $\tau_{i,j}$ beyond this job's deadline. However, and here comes the "almost" into play, there is so far no means of preventing a thread from immediately switching to the next SC and hence from consuming the budget of a not-yet released job. Although this is not a problem for EDF where SC priority are monotonically decreasing, it becomes an issue when allowing priorities to increase.

To enforce the use of budgets only after the release of the corresponding job, we propose to apply the same mechanism for switching to the next job that Jean Wolter introduced to cope with sporadic tasks in the first place. In addition to $\hat{P}_i$, we therefore introduce a second inter-release time: the refill time $\hat{R}_i$ plus configuration options to determine whether SCs are *queued* or *loose* (i.e., simultaneously available) and whether switching to the next SC corresponds to the release of the next sporadic job. To not confuse terminology with the task parameters, we will use $\hat{P}_i$ as before for the job-to-job minimal inter-release time and set $\hat{R}_i = HP$ to ensure the job's availability in the next hyperperiod.

Several variants of static mixed-criticality (SMC) algorithms have been proposed that can all be mapped in the above described way. For example, Vestal [5] suggested an algorithm based on Audsley's algorithm [13] to determine a MC-feasible per job priority assignment called own-criticality based priority assignment (OCBP). The basic idea is that if a job τ_i still receives sufficient time in between its release and deadline when it runs at the lowest priority and if we don't care about the order of or deadline misses of higher prioritized jobs, then we can fix this job at this priority and search for a next suitable job in the remaining set. Thereby, higher prioritized jobs τ_h are assumed to require at most $C_h(l_i)$. Lacking budget enforcement, Vestal first assumed a complete analysis of lower criticality tasks also at higher criticality levels leading to $C_i(h) > C_i(l_i)$ for $h > l_i$. Adding this additional constraint, Baruah and Burns [14] could improve on the schedulability of this algorithm. In [7], Baruah, Burns and Davis could finally show that presenting OCBP with a limited choice (deadline monotonic sorting of jobs within a criticality level and then for each level the job with the largest absolute deadline) suffices to obtain a feasible schedule.

Notice, although SC representations of all jobs in the hyperperiod are possible, other representations may be more appropriate for systems where it is feasible to support one specific class of MC-scheduling algorithms. In these systems, where the majority of all jobs follow a standard priority assignment such as EDF, a third extension of SCs will be helpful: to regard queued SCs as exceptions of one dedicated SC yielding the regular behavior. For example, if all jobs but

the third and fifth follow the EDF priority assignment, an EDF SC-scheduler could be used to schedule the dedicated SC with the exception of those jobs that have alternatives queued. That is, after finishing the first job, the scheduler awaits the second release of the dedicated SC at which time it inserts it to the ready queue based on the stored relative deadline. However, for the third release it makes this decision based on the adjusted deadline of the alternative SC.

C. Adaptive Mixed-Criticality Algorithms

So far we have only considered scheduling algorithms that rely on the relative priority ordering of SCs to guarantee MC-schedulability. However, with an appropriate monitor of thread execution times, which is trivially given in the form of the next scheduling-context signal, MC schedulers could also react to tasks exceeding their low criticality WCET estimates. This class of scheduling algorithms is called adaptive mixed-criticality algorithms [7]. Whenever a job $\tau_{i,j}$ exceeds the WCET estimate $C_i(l)$ of a criticality level l, the scheduler follows this transition from l to $l+1$ and discontinues the execution of all l-criticality tasks. To do so, with scheduling contexts, a fourth extension is required: to disable a group of scheduling contexts. Adding this extension requires an indirection to an *enabled token*, which the scheduler can toggle to disable at once all SCs that refer to this token.

Notice, to meet the demands of the MC-schedulability criterion, lower criticality jobs need never be continued once a higher criticality job has exceeded its low WCET estimate. However, it is of course desirable to return to a fully operational system as quickly as possible. The challenge is therefore in quickly re-enabling SCs once they have been disabled and it is safe to re-enable low SCs. As low WCET overruns are extremely rare events, we propose to simply deactivate but not dequeue inactive SCs from the ready queue. This way, re-enabling boils down to setting a bit in the enabled token. The additional scheduling overhead when skipping over disabled SCs is deterministic and can be considered for the high criticality WCET estimates.

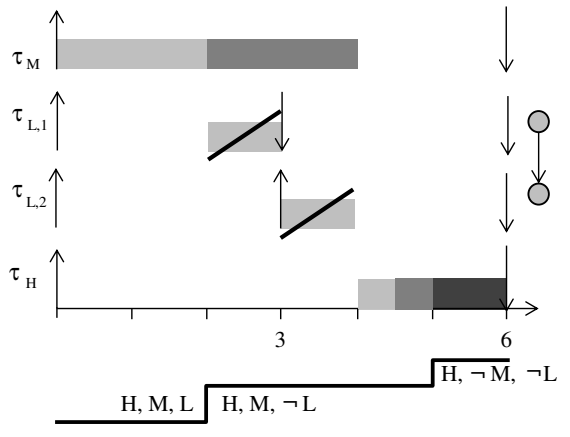


Fig. 5. Use of enabled token to disable groups of SCs.

Figure 5 shows the structure of enabled tokens in an example that is not schedulable with classical OCBP because of the consideration of $\tau_{L,1}$ for τ_H, although $\tau_{L,1}$'s execution time is hidden behind τ_M's medium-criticality excess budget. Because τ_M and τ_H require together all 6 time units, none of the low jobs $\tau_{L,1}$ and $\tau_{L,2}$ may be higher prioritized than both of the higher criticality tasks. On the other hand, because τ_M's low WCET estimate is 2 time units and τ_H's is 0.5, we cannot run both of them at a higher priority than the first low job $\tau_{L,1}$ because otherwise, if both the medium and the high criticality task stay within their low budgets, $\tau_{L,1}$'s completion could not be guaranteed. The only priority assignments that remain are therefore $\tau_{L,1}$ at an intermediate priority between τ_M and τ_H. However, OCBP considers $C_L(H)+C_M(H)$ if $\tau_{L,2}$ executes at a lower priority than τ_H or $2C_L(H)+C_M(H)$ otherwise, which both is not enough to complete τ_H at one of the two lowest priorities.

Shown at the bottom of Figure 5 is the gradual increase of the system's criticality level, which has lead to the dropping of τ_L's jobs. Also shown is the enabled token.

We suggest implementing the enabled token as a small bitfield complemented with a mask inside each SC. The mask is used to determine which bits are significant for this SC. This way, the first n criticality levels could be disabled by clearing the bit of the n^{th} criticality level and making all tasks of criticality l significant on all bits of higher or equal criticality.

D. Earliest Deadline First — Virtual Deadlines

Realizing that fixed-job priority algorithms (such as OCBP) are limited in their schedulable utilization, Baruah, Bonifaci and D'Angelo [8] transitioned from single priority schemes to a dual priority scheme (or more generally an n priority scheme where $n=|L|$ is the number of criticality levels). For as long as no task exceeds its low WCET estimate, the schedule follows the EDF algorithm with virtual deadlines to make room for a potential criticality change. Once such a change happens, low tasks are disabled and the high tasks transition to a second priority scheme based on classical EDF.

Group enable and disable (as we have seen it in the previous section) allow us to keep both priority settings simultaneously (either explicit or implicit through deadlines interpreted by a host EDF scheduler). Once a criticality change happens, all low-criticality SCs are disabled by clearing their significant low flag and high criticality SCs are enabled by setting the formerly disabled high criticality flag in the enabled token.

IV. Related Work

Most closely related to our work, although not mixed-criticality, is part of the work by Regehr and Stankovic in the context of hierarchical scheduling of soft real-time tasks (see Table 1 in [11]). As part of their hierarchical scheduling framework they derived a map how guarantees provided by one kind of scheduler translate into the guarantees that a nested scheduler may give for its tasks. For example, a scheduler receiving $X=50\%$ CPU share may translate this guarantee into multiple proportional share guarantees up to the point where the sum of shares Y_i is at most X. A particularly interesting case and direction of future work for this work is Surplus Fair Scheduling (SFS) [15]. Given multiple shares, SFS is able to produce a new set of shares carrying a combination of the received guarantees.

Zhang and Burns [12] investigate the schedulability of multiple earliest deadline first layers on top of a fixed-priority scheduler. Although not per se mixed criticality, Zhang's analysis can easily be extended to criticality monotonic settings where each nested EDF scheduler is responsible for tasks of one criticality level. To decide MC-schedulability, all that has to be done is to repeat the analysis for each criticality level l assuming the tasks in all higher prioritized EDF schedulers do not exceed $C_i(l)$. SCs and the consideration of slack in Zhang's analysis would even allow for occasional criticality inversions by raising selected tasks above higher criticality EDF levels provided there is enough slack in these levels.

Queued SCs generalize dual priority scheduling [16] when the transition to the next SC is not limited to thread-triggered events but for example to the release of the thread in the first place. In this case, $\hat{P}_i$ serves as trigger to switch to the higher band priority.

V. Conclusions and Future Work

As a first step in the direction of evaluating fixed SC-priority scheduling algorithms where tasks have multiple time quanta to choose from, we have investigated the mapping of five mixed-criticality algorithms. We found that although initially not all algorithms could be supported, small changes to the use and interface of SCs allowed us to map all investigated algorithms to SC-based scheduling. These changes are deadline enforcement, a separate refill period, support for sporadic jobs in the form of queued SCs and group enable/disable functionality through enabled tokens. Most of these extensions appeared recurringly as demands in discussions about Fiasco.OC's SC-based scheduling interface and are feasible to be implemented both in microkernel-based systems and elsewhere. Proper integration in a microkernel-based system regarding isolation and security characteristics are subject to evaluation.

Directions for future work include a more elaborate and formal handling of the question what scheduling guarantees can be given from a combination of fixed-priority time quanta and an investigation of further algorithms including and beyond mixed-criticality.

Acknowledgments

This work was in part funded by the DFG through the cluster of excellence "Center for Advancing Electronics Dresden", by the EU and the state Saxony through the ESF young researcher group "IMData" and by the national science foundation through grant NSF CNS-0931985.

References

[1] U. Steinberg, J. Wolter, and H. Härtig, "Fast Component Interaction for Real-Time Systems," in *17th Euromicro Conference on Real-Time Systems*, Palma de Mallorca, Spain, July 2005.

[2] J.-Y. Chung, J. W. S. Liu, and K.-J. Lin, "Scheduling periodic jobs that allow imprecise results," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1156–1173, Sep. 1990.

[3] C.-J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig, "Quality Assuring Scheduling - Deploying Stochastic Behavior to Improve Resource Utilization," in *22nd IEEE Real-Time Systems Symposium (RTSS)*, London, UK, Dec. 2001.

[4] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig, "Flattening hierarchical scheduling," in *Proceedings of the tenth ACM international conference on Embedded software*, ser. EMSOFT '12. New York, NY, USA: ACM, 2012, pp. 93–102.

[5] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Real-Time Systems Symposium*. Tucson, AZ, USA: IEEE, December 2007, pp. 239–243.

[6] H. Li and S. K. Baruah, "An Algorithm for Scheduling Certifiable Mixed-Criticality Sporadic Task Systems," in *RTSS*. IEEE Computer Society, 2010, pp. 183–192.

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

[8] S. K. Baruah, V. Bonifaci, G. D'Angelo, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie, "Mixed-criticality scheduling of sporadic task systems," in *Proceedings of the 19th European conference on Algorithms*, ser. ESA'11. Springer-Verlag, 2011, pp. 555–566.

[9] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications," in *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ser. ECRTS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 147–155.

[10] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *Computers, IEEE Transactions on*, vol. 61, no. 8, pp. 1140–1152, 2012.

[11] J. Regehr and J. A. Stankovic, "HLS: A framework for composing soft real-time schedulers," in *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*. Washington, DC, USA: IEEE Computer Society, 2001.

[12] F. Zhang and A. Burns, "Analysis of hierarchical edf pre-emptive scheduling," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, 2007, pp. 423–434.

[13] N. Audsley, "On priority assignment in fixed priority scheduling," *Information Processing Letters*, vol. 79, no. 1, pp. 39–44, 2001.

[14] S. Baruah and A. Burns, "Implementing mixed-criticality systems in ada," in *Reliable Software Technologies — Ada-Europe'11*. Springer, 2011, pp. 174–188.

[15] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, "Surplus fair scheduling: A proportional-share cpu scheduling algorithm for symmetric multiprocessors," in *4th Symposium on Operating System Design and Implementation (OSDI)*, San Diego, CA, USA, Oct. 2000, pp. 45–58.

[16] R. Davis and A. Wellings, "Dual priority scheduling," in *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, 1995, pp. 100–109.

Safety Assurance Driven Problem Formulation for Mixed-Criticality Scheduling

Patrick Graydon[2] and Iain Bate[1,2]
[1]Department of Computer Science, University of York, York, UK
[2]Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden
email: patrick.graydon@mdh.se and iain.bate@cs.york.ac.uk

***Abstract*—In 2007, Vestal proposed Mixed-Criticality Scheduling (MCS) to increase utilisation despite imperfect timing evidence. Others have since refined the MCS problem formulation, proposed alternative scheduling approaches, and evaluated their performance. We assess existing MCS problem formulations from a safety assurance perspective and report problems found. Among these is the use of the word 'criticality' to mean several related but distinctly different things such as Safety Integrity Levels (SIL), importance, and confidence. We conclude with suggestions for addressing the problems found.**

I. Introduction

In 2007, Vestal proposed Mixed Criticality Scheduling (MCS) as a way to achieve high utilisation despite imperfect timing evidence [1]. He observes that (a) increased confidence in WCET limits comes at the expense of increased pessimism and (b) tasks vary in criticality. MCS schedules a system so that all tasks can run if they adhere to a low-confidence WCET limit. Failing that, the most critical tasks can run if they adhere to a larger, higher-confidence WCET limit. Later work expanded the problem definition, proposed alternative scheduling algorithms, and demonstrated superior utilisation [2]–[10].

But existing definitions of the MCS problem are presented primarily from a scheduling perspective. We report an analysis of the MCS problem from a safety assurance perspective. Amongst other findings, we show that current formulations use the word 'criticality' to represent several related but distinct concepts, including Safety Integrity Level (SIL), importance, and confidence. We discuss how this affects safety assurance and present suggestions for addressing the problems found.

In section II, we survey formulations of the MCS problem. We discuss confidence in execution time assessments in section III and the MCS-related demands of safety assurance in section IV. In section V, we critically examine existing problem formulations and suggest improvements. We discuss future work in section VI and conclude in section VII.

II. Existing Problem Formulations

Vestal was the first to publish a formulation of the MCS problem [1]. Others, including Baruah and Yi, have extended the problem formulation to account for more parameters changing with criticality and to generalise criticality modes.

A. The Vestal Formulation

Vestal's 2007 formulation of the MCS problem begins with the 'conjecture that the higher the degree of assurance required that actual task execution times will never exceed the WCET parameters used for analysis, the larger and more conservative the latter values will become in practice' [1]. In his model, a system has tasks $\tau_1 \ldots \tau_n$, each with period T_i and deadline D_i. System development adheres to 'an ordered set of design assurance levels' $\mathcal{L} = \{A, B, C, D\}$ with A being the highest. $C_{i,l}$ gives the compute time for task τ_i at assurance level l, with $C_{i,A} \geq C_{i,B} \geq C_{i,C} \geq C_{i,D}$ for all i. The goal is 'to assure to level L_i' that each task τ_i 'never misses a deadline'.

B. The Baruah and Burns Formulation

Baruah and Burns extend Vestal's formulation [2], [5]. A system is 'defined as a finite set of components $\mathcal{K}$', each with level of criticality L [5]. As in Vestal's formulation, each task τ_i is defined by $(T_i, D_i, \vec{C_i}, L_i)$. $\vec{C_i}$ 'will be derived by a process dictated by the criticality level' and 'the higher the criticality level, the more conservative the verification process and hence the greater will be the value of C_i'. However, noting that 'the higher the criticality level, the greater the need for the task to complete well before any safety-critical timing constraint', the model permits different deadlines at different criticality levels so long as $L_i^1 > L_i^2 \Rightarrow D_i^1 \leq D_i^2$. Moreover, noting that 'the higher the criticality level, the tighter the level of control that may be needed', Baruah and Burns allow different periods at different criticality levels so long as $L_i^1 > L_i^2 \Rightarrow T_i^1 \leq T_i^2$. If a task at criticality level ℓ overruns $C_{i,\ell}$, tasks at criticality levels ℓ and lower are prevented from running again until the processor is idle [2].

Baruah and Burns identify two 'issues' that MCS must address: *static verification* and *run-time robustness* [5]. By the former, they mean that 'for each criticality level ℓ, all jobs of all tasks with criticality $\geq \ell$ will complete by their deadlines in any criticality-ℓ behavior'. By the latter, they mean that after a transient overload, 'a robust scheduling algorithm would, informally speaking, be able to "recover" … and go back to meeting the deadlines of lower-criticality jobs as well'.

C. The Ekberg and Yi Formulation

Ekberg and Yi extend Vestal's formulation further [7]. Previous formulations defined criticality modes purely as a mechanism for preserving timing guarantees. Noting that 'it should be up to the system designer to decide what it means for the system to be in any one particular criticality mode', they propose using criticality modes to reconfigure systems in

response to events such as hardware failures. (General models for such reconfiguration have been proposed elsewhere [11]; the contribution of Ekberg and Yi lies in combining MCS with more general notions of reconfiguration.) In their model, a system is defined by a set of tasks τ and a DAG G. The vertex set $V(G)$ defines the criticality modes and the edge set $E(G)$ specifies the permissible changes between them. Each task is defined by the set of criticality modes in which it is active, $\mathfrak{L}$, and for each $m \in \mathfrak{L}_i$ a tuple $(C_i(m), D_i(m), T_i(m))$. During a criticality level switch from m to m':

- Where $m \in \mathfrak{L}_i$ and $m' \notin \mathfrak{L}_i$, the system suspends τ_i.
- Where $m \notin \mathfrak{L}_i$ and $m' \in \mathfrak{L}_i$, the system activates τ_i.
- Where $m \in \mathfrak{L}_i$ and $m' \in \mathfrak{L}_i$, the system changes τ_i's parameters to those specified for m'.

These changes take immediate effect. For example, if the system suspends τ_i, it discards any active jobs.

Note that G is a DAG: if the system switches to a higher-criticality mode, it will never switch back. Ekberg and Yi observe in a footnote that 'one could easily find a time point where it is safe to switch back', but leave this to future work.

III. Confidence in Execution Time Assessments

A key assumption behind all three MCS problem formulations is *WCET confidence monotonicity*: that the degree to which a WCET limit overestimates true WCET increases monotonically with confidence. WCET assessment approaches can be classified as (a) *dynamic* (measurement-based), (b) *static* (analysis-based), or (c) *hybrid* [12]. We consider the WCET confidence monotonicity assumption with respect to each WCET assessment approach in turn.

A. Dynamic Approaches to WCET Assessment

The simplest dynamic WCET assessment approach is *High Water Mark testing* (HWM). In HWM, analysts use the longest execution time observed in testing as a WCET estimate [12]. The primary source of uncertainty in WCET limits derived from HWM testing is imperfect test coverage. Secondary sources include the correctness of any tools, the integrity of data gathering, any differences between the test system and the deployed system. These sources are epistemic, i.e. related to what we do not know rather than arising from chance.

Developers sometimes analyse execution paths and choose test cases in an attempt to stimulate the worst case [1]. If a and b are sets of test cases and $a \subset b$, then HWM testing using b inspires more confidence than the same testing using a. Unfortunately, it is not generally possible to quantify either the likelihood or degree of underestimate using this technique [12]. Two examples of why are as follows. Firstly, the tasks are analysed for their WCET in isolation with the cache flushed, however the cache is not flushed before the task starts executing. Cache-related anomalies exist which mean that this situation can lead to a higher execution than the measured WCET [13]. Secondly, the WCET is measured during system integration testing however this cannot for instance cover all initial cache states and preemptions scenarios.

In probabilistic approaches, analysts use Extremal Value Theory (EVT) to fit observed execution times to a distribution [12], [14]. By selecting a WCET estimate from further to the right of the distribution, a developer can reduce the uncertainty *from test coverage* at the cost of increased pessimism. Because the distributions do not model epistemic uncertainty, probability figures taken from them are not complete descriptions of the confidence testing should inspire. However, we know of no reason to think that total confidence does not rise monotonically with distribution-derived probabilities.

B. Static Approaches to WCET Assessment

Static WCET analysis considers all possible paths through analysed code. Thus, it is not affected by the main source of uncertainty in dynamic approaches. However, static analysis does not produce *perfect* confidence [12]. User inputs such as loop bound limits might be wrong, tools might be buggy, and processor models may be wrong.

Because primary sources of uncertainty in static approaches are epistemic, there is no clear relationship between confidence and overestimate amongst static approaches. Consider two static analysis tools A_1 and A_2 and a program p for which A_1 produces a greater WCET figure than A_2. The extra overestimate might mask some tool defects (if any), but if the tool qualification evidence for A_2 is more comprehensive than that for A_1, we might justifiably have more confidence that A_2 does not underestimate true WCET.

We can't quantify test coverage uncertainty in HWM-based approaches. But all other issues (e.g. developer competence, tool qualification evidence) being equal, static approaches produce greater confidence. Since a static analysis approach (if it is sound, given correct inputs, etc.) never underestimates WCET and HWM testing never overestimates, static approaches will overestimate more. The degree of overestimate will depend on the target CPU's complexity and the analysis tool's sophistication. Analysis results within 20% of the highest observed execution time have been achieved [15], but complex features such as multi-level caches can result in overestimates of 100% or more [16]. Analysis of software on multicore platforms is particularly challenging [12].

It is not clear how confidence and overestimation compare between static approaches and probabilistic approaches, at least at the high end of the distribution.

C. Hybrid Approaches to Determining WCET

Hybrid approaches combine static analysis and measurement [12], [17]. These approaches divide the software into blocks, dynamically measure the runtime of each, statically analyse the software structure, then combines block timing figures into a complete WCET estimate. The simplest form of hybrid approach uses a HWM time for each block, but more complex analyses take inter-block dependencies into account [18]. For a given test suite, hybrid approaches should be less likely to underestimate WCET than simple HWM testing: many test cases might provoke a given block's worst performance, but only one case yields the worst performance for the

task being tested. However, the basic sources of uncertainty are the same as with dynamic methods. Underestimation and overestimation are possible and their likelihood unknown [12]. It is less clear how either the confidence inspired by hybrid approaches or the degree of overestimate compares to those of static approaches. In addition, no proof of safeness exists for the hybrid approaches.

D. The WCET Confidence Monotonicity Assumption

Between the HWM testing approach and any sound, well-constructed static analysis approach, the WCET confidence monotonicity assumption almost certainly holds. With other approaches, it is less clear that this assumption holds. We return to this issue and its implications in subsection V-A.

IV. The Demands of Safety Assurance

Successful instantiations of MCS in safety critical systems must provide the properties and evidence needed for safety and safety assurance. In this section, we outline the timing-related demands of typical safety assurance approaches.

Standards for software for use in safety critical systems vary. For example, software for use on commercial air transports is constructed in conformance to RTCA DO-178B [19] or its successor DO-178C [20]. Software for general safety-related systems is often developed in conformance to IEC 61508 [21]. Software for road vehicles is developed in conformance with ISO 26262 [22]. There are standards for rail applications [23] and other specific domains. In yet other domains, standards dictate a systems safety approach but no specific approach to software [24]. In this section, we consider *typical* safety assurance approaches, referring to common standards for concrete examples where appropriate.

A. Derivation of Timing-Related Safety Requirements

Engineers typically derive timing-related safety requirements from hazard and risk analyses. For example, engineers following IEC 61508 (1) define the system concept and scope, (2) perform hazard and risk analysis, (3) define overall safety requirements, and (4) allocate safety requirements to particular functions and subsystems[1] [21]. The aim of hazard and risk analysis is to determine (a) the system hazards, (b) the event sequences that could lead to those, and (c) the risks related to the identified hazards. The overall safety requirements detail risk reduction measures meant to achieve functional safety targets. Engineers allocating safety requirements specify an appropriate SIL[2] for each function and then for the subsystems implementing those functions. Timing-related safety requirements capture when these functions must be activated, performed, completed, etc. if the system is to meet its safety targets. Most standards for software in safety critical systems use some variant of this process. (RTCA DO-178B [19] and DO-178C [20] do not: they exclude system safety engineering from their scope. These standards are used with SAE ARP 4754A [25], which does follow a broadly similar pattern.)

[1]IEC 61508 uses the term 'system' here, but this process applies to multiple interacting 'systems' and considers over-arching safety goals.

[2]SILs go by different names (e.g. ASIL, DAL, SWDAL) in other standards.

B. The Meaning and Role of Safety Integrity Levels

SILs play a complex and frequently misunderstood role in safety standards that use them [26]. SIL is *not* a measure of target reliability generally: if failure of a subsystem would have little effect on safety but a disastrous effect on business objectives, that subsystem might have a low SIL. SIL is *not* a measure of the importance of any particular property: a high-SIL function might cause harm if not performed correctly but not if completed late. SIL is also *not* a measure of *achieved* reliability. In IEC 61508 [21] (and also in ISO 26262 [22]), SIL drives which techniques (e.g. use of formal methods or dynamic reconfiguration) the standard *recommends*. However, developers are not *obligated* to use even highly recommended techniques: they may instead explain their reasoning to an assessor and agree an alternative process [21]. Even if software development process was a strong predictor of reliability – there is little or no direct evidence showing that it is – standards don't strictly dictate development practice[3].

The role of SILs is best viewed as part of a process of deriving an appropriate development process from the hazard and risk analysis [26]. For example, consider IEC 61508, which applies to control systems meant to mitigate risks posed by equipment under control (e.g. an industrial metal fabrication tool) [21]. During hazard and risk assessment, engineers work out both the *consequences* (i.e. severity) and *likelihood* of harm (assuming no mitigation from the control system). These factors determine *risk*, which is compared with tolerable risk thresholds to determine the need for mitigation. From this, engineers derive a tolerable rate of failure for each mitigation function. This failure rate is converted into a SIL and used to drive development of appropriate development process. While other standards that use SIL do so in a broadly similar way, there are substantial differences and these can be a source of confusion [26].

C. Partitioning and Integrity

Software running on one microprocessor frequently implements multiple functions. These functions might have different safety integrity needs and developers might want to save money by using less-rigourous processes to implement lower-SIL functions. However, software implementing one function could interfere with software implementing another, for example by writing to a random memory address. Standards typically address this by assigning to software the highest SIL of any function it implements *unless* developers demonstrate *partitioning integrity*. For example, IEC 61508 allows allocating different SILs to different parts of the software only when developers show that 'there is sufficient independence of implementation between these particular safety functions' [21].

D. Adequate Confidence in Timing Claims

None of the common standards for software in safety critical applications clearly defines what constitutes adequate confi-

[3]Not even DO-178B [19] or DO-178C [20]. Developers and assessors agree a *Plan for Software Aspects of Certification* that details the specific activities undertaken to achieve the standard's SWDAL-specific objectives.

dence in a claim about task execution time [12]. For example, RTCA DO-178B requires 'review and analysis' to 'determine the correctness and consistency of the source code, including ... worst-case execution timing' [19]. This objective applies at all but the least-critical SILs. A companion document clarifies that dynamic approaches to determining WCET are sometimes appropriate but does not discuss which approaches are appropriate and not appropriate at each SIL [27].

E. Survivability and Graceful Degradation

Common standards for software in safety critical applications focus mainly on preventing or detecting defects. But it is not generally possible to ensure that software-based systems will *never* fail. Recognising this, some standards, researchers, and authorities advocate building software so as to achieve *survivability* [28]. Survivability can be broadly defined as the ability of a system to provide *essential* services in the face of attacks and failures. To achieve this resilience, engineers build systems to reconfigure themselves in response to defined failure and attack conditions [11]. Such reconfiguration is *one* way to achieve the 'graceful degradation' that IEC 61508 recommends at low SILs and highly recommends at high SILs [21]. Researchers have formalised the definition of a survivability specification as $\{S, E, D, V, T, P\}$ where:

- S specifies acceptable forms of service from the system (e.g. full service, limp-home mode)
- E gives permitted values for each aspect of service value (e.g. $\{\text{mass air flow sensor} \mapsto \{\text{working}, \text{failed}\}, \ldots\}$)
- D defines legal combinations of the values in E
- $V : S \times D \to \mathbb{N}_1$ gives the relative value that each configuration supplies under each environmental condition
- $T \subseteq S \times S \times D$ defines the legal mode transitions
- $P : S \to \{p : \mathbb{R} \mid 0 < p < 1\}$ specifies the probability with which the implementation of each service mode must meet its dependability requirements (e.g. $\{\text{full\_service} \mapsto 0.999, \text{limp\_home} \mapsto 0.99999\}$) [28]

Achieving graceful degradation always requires understanding how a system reacts to adverse events but does not always require explicit mode transitions. For example, graceful degradation might mean avoiding a design that causes a system that receives one too many requests to service none of them.

F. Modes of Operation

Standards use the word 'mode' to mean different things. Each configuration in a survivability architecture can be called a mode. But when IEC 61508 directs engineers conducting hazard and risk analysis to 'give particular attention to abnormal or infrequent modes of operation of the [equipment under control]' [21], it means something different. The modes in this case are the ways in which the system is used (e.g. continuous production, prototyping/piecework, maintenance, etc.).

V. Critique of Existing Problem Formulations

Most published work on MCS is from a scheduling perspective. In this section, we criticise the MCS problem formulations outlined in section II and suggest improvements.

A. Impact of the WCET Confidence Monotonicity Assumption

As we showed in section III, it is not clear that the WCET confidence monotonicity assumption holds over all WCET assessment approaches. Static analysis approaches generally produce both greater confidence and greater overestimate than HWM testing approaches, but the issue is less clear where statistical approaches or hybrid approaches are concerned.

However, it might not matter that the WCET confidence monotonicity assumption does not hold universally. Suppose that, for a given task in a given system, a high-confidence WCET assessment approach yields a lower WCET limit than a low-confidence approach. Developers could simply use the high-confidence WCET limit for both C_{HI} and C_{LO}.

When judging the risk associated with a design that includes MCS, developers need to reason about the confidence inspired by the WCET assessment approach used. This is complicated by our general inability to precisely quantify the total confidence / uncertainty associated with WCET assessment techniques. But this difficulty is not unique to MCS; it applies even when other scheduling approaches are used [12].

1) Suggestion: Confidence in WCET limit figures is incompletely understood and MCS might not provide benefit in cases where the WCET confidence monotonicity assumption does not hold. But there are important cases where it does, chief among them between HWM testing and static analysis. For the most critical functions, engineers and regulators might agree that only a technique that significantly overestimates true WCET provides sufficient confidence. Standard safety processes dictate that developers must either demonstrate partitioning integrity or use the same technique for all parts of the software. The MCS problem formulation should position MCS as a way to demonstrate partitioning integrity despite using less-conservative techniques such as HWM testing for functions that can tolerate greater uncertainty.

B. Multiple Meanings of 'Criticality'

Vestal's formulation [1] used 'criticality' synonymously with SIL (or 'Design Assurance Level' in RTCA's DO-178B nomenclature [19]). That formulation uses criticality to indicate both the consequence of a task missing its deadline and confidence in the WCET limit figures used in timing analysis. As we noted in subsection IV-B, SIL is at best a crude indicator of these things. A high-SIL task overrunning its deadline might cause little or no safety impact and SIL is only the starting point for a negotiation between developers and assessors (if any) over which WCET assessment approach is appropriate. Moreover, engineers using a standard that defines five SILs are unlikely to use five different WCET assessment techniques.

The Baruah and Burns formulation [2], [5] extends the Vestal formulation [1] but uses the word criticality in much the same way. The Ekberg and Yi formulation [7] goes further, extending the Vestal formulation to combine MCS with reconfiguration for survivability. In doing so, it uses the word criticality as a label for what the survivability literature calls an 'acceptable form of service' [28]. This meaning is not interchangeable with either of the first two. Shifting to a new

form of service might increase the consequences of some tasks missing their deadlines and decrease the consequences for others. For example, suppose that a combat UAV is engaged in aerial combat and reconfigures into a dogfighting mode. In that mode, timely vision and control actuation might become more important than in normal flight while automatic de-icing becomes less important. But while the importance of a task might change when reconfiguring for survivability, confidence in WCET limits will not.

1) Suggestion: The MCS problem formulation should use different terms for each of the different concepts now being referred to as criticality. We suggest the following definitions:

- '*Importance*' should be used to describe the consequence of a task overrunning its deadline. Importance varies with service mode and might be lower than the SIL(s) associated with the tasks' service would indicate.
- '*Confidence*' should be used to describe confidence in a WCET limit or WCRT figure. *Uncertainty* is the lack of confidence (e.g. for the reasons discussed in section III).
- '*Mode*' should only be used as part of a complete term that describes a specific mode. *Service mode* should describe a mode used in reconfiguration for survivability. *Mode of operation* should describe the way in which a system is being used by its operators. *Scheduler mode* should be used to describe whether a scheduler is excluding low-importance tasks in order to help high-importance tasks meet their deadlines.

Because SILs are defined and used differently in different standards [26], the MCS problem should not be described in terms of SIL. Where helpful, documents giving advice for building systems using MCS can explain how importance and confidence map to a specific standard's definition of SIL.

C. There Are Modes, and Then There Are Modes

The Ekberg and Yi MCS problem formulation [7] includes a single reconfiguration mechanism meant to facilitate both (a) surviving equipment failure and (b) recovering from an overrun of a low-confidence WCET figure. This is meant to promote generality (for its own sake) and reuse of validated approaches. They write, 'we would like the task model to be as general as possible. ... It is not unlikely that some existing solutions regarding the scheduling of regular mode switching systems can be adapted for mixed-criticality scheduling' [7]. However, it is not clear that (i) the Ekberg and Yi reconfiguration model is general enough for general survivability purposes, (ii) that a sufficiently general model would be suited to tolerating overruns, or (iii) that a combined model would be practical from a safety assurance standpoint.

In subsection IV-E we described a formal model of reconfiguration for survivability [28]. Such reconfigurations might take much longer than simply setting a mode variable to a different value. For example, a system migrating a task to a different computing node might need to load object code and initial state into that node's memory [29]. Clearly, the time budget for such reconfigurations cannot be included in the time budget for normal task execution. To accommodate such reconfigurations, reconfiguration architectures include special procedures that are executed to accomplish the reconfiguration [11].

As Baruah and Burns point out, MCS is meant to protect important services from a task failing to meet its low-confidence WCET limits due to a transient overload [5]. Achieving this requires reacting within a period. It is doubtful that one mechanism would be suitable for such reconfigurations *and* those that take many periods to complete.

As we described in subsection IV-C, many standards require developers to demonstrate temporal partitioning integrity. Where such integrity relies in part on MCS, demonstrating it requires demonstrating that reconfiguration protects important services from overruns of low-confidence WCET limits. Graceful degradation, on the other hand, is validated through architectural review, testing and analysis of the implementation, and analysis of how mode transitions could lead to hazards. Using a single mechanism to implement both mixes goals and concepts usually kept separate in the standards' process and might complicate the safety assurance effort. If tolerating overruns requires x distinct modes and tolerating hardware failures requires y distinct modes, a unified system might implement $x \times y$ modes and corresponding mode transitions. Because each of these must be tested, the combined implementation might significantly increase testing effort.

1) Suggestion: While reconfiguration in response to low-confidence WCET limit overruns and reconfiguration in response to hardware failure share some features, these reconfigurations are different and should be handled by separate reconfiguration mechanisms. However, as discussed in subsection V-B, some tasks might be more important in some service modes than in others. The MCS problem formulation should be given in terms of task importance rather than SIL to facilitate graceful degradation.

D. To Kill or Not to Kill?

The *Partitioned Criticality* (PC) scheduling scheme assigns priorities so as to preserve temporal isolation without the need for a special monitoring and intervention mechanism [5]. In contrast, *Static Mixed Criticality* (SMC) uses run-time monitoring to restrict tasks to their run-time limits. *Adaptive Mixed Criticality* (AMC) goes further, using run-time monitoring to halt all criticality ℓ tasks whenever a criticality ℓ or higher task overruns its WCET limit. Unfortunately, some definitions of this monitoring are inappropriate for many applications.

It strains credulity to think that some tasks are so unimportant that they need never be executed following a transient overload. Indeed, anecdotes told to us in confidence reveal that some developers create high-SIL functions that depend on input from low-SIL functions, justifying this arrangement by designing the high-SIL task to tolerate bad or missing input until health monitoring restarts the low-SIL task. A reasonable MCS approach must restart halted tasks when it is safe to do so. Baruah and Burns do this when the processor is next idle [2]. The Ekberg and Yi MCS problem formulation kills unimportant tasks permanently, but a footnote raises the possibility of restarting halted tasks [7].

1) Suggestion: MCS problem formulations should explicitly define when they will restart tasks following an overrun. It would also help to explain what will happen when an important task overruns a high-confidence WCET limit.

VI. Future Work: An MCS Assurance Argument

A *safety argument* for a given system explains how evidence in a *safety case* shows satisfaction of safety requirements and, ultimately, the operational definition of 'adequately safe to operate' in use in that system [30]. Other forms of *assurance argument*, such as *conformance argument*, can be used to show conformance with a safety standard [31].

Timing evidence, scheduling policy, and task allocation have a complex relationship with risk and safety standards. We have described some of that complexity in this paper. Such complexity makes it difficult to determine by ad hoc review whether a given formulation of the MCS problem captures the properties and evidence needed for safety assurance.

In prior work, we created an assurance argument to facilitate criticism of timing-related safety evidence and processes [12]. In this work, we suggested improvements to existing MCS problem formulations. We further suggest creating and criticising a safety argument for the revised formulation as a means of assessing its completeness and utility. Conformance arguments would likewise help to assess the problem formulation's fitness for use with a given standard and type of system.

VII. Conclusions

In some safety-critical software systems, adequate risk reduction requires meeting deadlines. Unfortunately, some WCET assessment approaches that yield high confidence might substantially overestimate WCET. MCS promises to always permit important services to run for up to some high-confident system while delivering the utilisation that less-conservative WCET limits would allow.

In this paper, we reviewed three MCS problem formulations from a safety assurance perspective. We found four issues: (1) reliance on a questionable assumption about confidence in WCET limits, (2) use of the word 'criticality' to mean several different things, (3) flawed support for survivability-related reconfiguration, and (4) a haphazard treatment of recovery from transient overloads. We suggested improvements in the model formulation to address these issues. We further suggest the development of an assurance argument for MCS as a means of exploring the complex assurance issues and tradeoffs surrounding scheduling policy, task allocation, confidence in timing evidence, partitioning, and graceful degradation.

Acknowledgement

We acknowledge the Swedish Foundation for Strategic Research (SSF) SYNOPSIS Project and EPSRC (UK) grant MCC (EP/K01 1626/1) for supporting this work.

References

[1] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Proc. Int'l Real-Time Systems Symp. (RTSS)*, 2007.

[2] S. Baruah and A. Burns, "Implementing mixed criticality systems in Ada," in *Proc. Reliable Software Tech. (Ada-Europe)*, 2011.

[3] S. K. Baruah, A. Burns, and R. I. Davis, "Response-time analysis for mixed criticality systems," in *Proc. Int'l Real-Time Systems Symp. (RTSS)*, 2011.

[4] S. Baruah, "Certification-cognizant scheduling of tasks with pessimistic frequency specification," in *Proc. Int'l Symp. Industrial Embedded Systems (SIES)*, 2012.

[5] A. Burns and S. Baruah, "Timing faults and mixed criticality systems," in *Dependable and Historic Computing*, ser. LNCS. Springer Berlin Heidelberg, 2011, vol. 6875, pp. 147–166.

[6] A. Burns and R. Davis, "Mixed criticality systems: A review," Department of Computer Science, University of York, York, UK, Tech. Rep. MCC-1(b), July 2013.

[7] P. Ekberg and W. Yi, "Bounding and shaping the demand of generalized mixed-criticality sporadic task systems," *Real-Time Syst.*, 2013, in press.

[8] N. Guan, P. Ekberg, M. Stigge, and W. Yi, "Improving the scheduling of certifiable mixed-criticality sporadic task systems," Uppsala University, Uppsala, Sweden, Tech. Rep., 2012.

[9] H.-M. Huang, C. Gill, and C. Lu, "Implementation and evaluation of mixed-criticality scheduling approaches for periodic tasks," in *Proc. Real-Time and Embedded Tech. and Applications Symp. (RTAS)*, 2012.

[10] M. Neukirchner, S. Stein, H. Schrom, J. Schlatow, and R. Ernst, "Contract-based dynamic task management for mixed-criticality systems," in *Proc. Int'l Symp. Industrial Embedded Systems (SIES)*, 2011.

[11] E. A. Strunk, "Reconfiguration assurance in embedded system software," Ph.D. dissertation, University of Virginia, Charlottesville, USA, 2005.

[12] P. Graydon and I. Bate, "Realistic safety cases for the timing of systems," *The Computer Journal*, 2013, in press.

[13] I. Bate, P. Conmy, T. Kelly, and J. McDermid, "Use of modern processors in safety-critical applications," *The Computer Journal*, vol. 44, no. 6, pp. 531–543, 2001.

[14] S. Edgar and A. Burns, "Statistical analysis of WCET for scheduling," in *Proc. Int'l Real-Time Systems Symp. (RTSS)*, 2001.

[15] R. Chapman, "Static timing analysis and program proof," DPhil Thesis, University of York, York, UK, 1995.

[16] S. Chattopadhyay and A. Roychoudhury, "Unified cache modeling for WCET analysis and layout optimizations," in *Proc. Int'l Real-Time Systems Symp. (RTSS)*, 2009.

[17] Rapita Systems, "RapiTime explained," http://www.rapitasystems.com/downloads/rapitime_explained_white_paper, July 2011.

[18] G. Bernat, A. Burns, and M. Newby, "Probabilistic timing analysis: An approach using copulas," *Journal of Embedded Computing*, vol. 1, no. 2, pp. 179–194, April 2005.

[19] RTCA DO-178B, *Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Inc., 1992.

[20] RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Inc., 2011.

[21] IEC 61508:2010, *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*, 2nd ed. International Electrotechnical Commission, 2010.

[22] ISO 26262:2011, *Road Vehicles — Functional Safety*. International Organization for Standardization, 2011.

[23] CENELEC 50128:2001, *Railway Applications — Communications, Signalling and Processing Systems — Software for Railway Control and Protection Systems*. European Committee for Electrotechnical Standardization (CENELEC), 2001.

[24] Defence Standard 00-56, *Safety Management Requirements for Defence Systems, Issue 4*. UK Ministry of Defence, 2007.

[25] SAE ARP4754A, *Guidelines for Development of Civil Aircraft and Systems*. SAE, 2010.

[26] F. Redmill, "Understanding the use, misuse and abuse of safety integrity levels," Redmill Consultancy, Tech. Rep., 2005.

[27] RTCA DO-248B, *Final Report For Clarification Of DO-178B*. RTCA, Inc., 2001.

[28] J. C. Knight, E. A. Strunk, and K. J. Sullivan, "Towards a rigorous definition of information system survivability," in *Proc. DARPA Information Survivability Conf. and Expo.*, 2003.

[29] N. C. Audsley and M. Burke, "Distributed fault-tolerant avionic systems — A real-time perspective," in *Proc. IEEE Aerospace Conference*, 1998.

[30] T. P. Kelly, "Arguing safety — A systematic approach to managing safety cases," DPhil Thesis, University of York, York, UK, 1998.

[31] P. Graydon, I. Habli, R. Hawkins, T. Kelly, and J. Knight, "Arguing conformance," *IEEE Software*, vol. 29, no. 3, pp. 50–57, May 2012.

A safety concept for a wind power mixed-criticality embedded system based on multicore partitioning

Jon Perez, David Gonzalez, Salvador Trujillo
Embedded Systems Group
Ik4-IKERLAN Technology Research Centre
Mondragon, Spain
jmperez,dgonzalez,strujillo@ikerlan.es

Ton Trapman, Jose Miguel Garate
Software and Performance
Alstom Renewables
Barcelona, Spain
anton-aart.trapman,jose-miguel.garate@power.alstom.com

***Abstract*—The development of mixed-criticality systems that integrate applications of different criticality levels (safety, security, real-time and non real-time) can provide multiple benefits such as product cost-size-weight reduction, reliability increase and scalability. However, the integration of applications of different levels of criticality leads to several challenges with respect to safety certification standards.**

This paper defines a safety certification strategy for IEC-61508 compliant industrial mixed-criticality systems based on multicore partitioning. The final objective is the certification of a wind-turbine mixed-criticality control system according to IEC-61508 and ISO-13849 industrial safety standards. This approach is illustrated with a simplification of the safety concept currently under detailed review by a certification body.

***Index Terms*—mixed-criticality ; safety; IEC-61508; certification; multicore; partition**

I. Introduction

Conventional embedded system architectures in multiple domains follow a federated architecture paradigm, in which the system is composed of interconnected embedded subsystems where each of them provides a well defined functionality. The ever increasing demand for additional functionalities leads to a considerable complexity growth [1] that in some cases limits the scalability of the federated approach. For example, a modern off-shore wind turbine dependable control system manages up to three thousand inputs / outputs, several hundreds of functions are distributed over several hundred nodes grouped into eight subsystems interconnected with a fieldbus and the distributed software contains several hundred thousand lines of code.

The integration of additional functionalities also leads to an increase in the number of subsystems, connectors and wires increasing the overall cost-size-weight and reducing the overall reliability of the system. For example, in the automotive domain, field data has shown that between 30-60% of electrical failures are attributed to connector problems [2].

The integration of applications of different criticality (safety, security, real-time and non-real time) in a single embedded system is referred as mixed-criticality system. This integrated approach can improve scalability, increase reliability reducing the amount of systems-wires-connectors and reduce the overall cost-size-weight factor. However, safety certification according to industrial standards becomes a challenge because sufficient evidence must be provided to demonstrate that the resulting system is safe for its purpose. Higher safety integrity functions must be interference free with respect to lower safety integrity functions.

This paper contributes with the definition of a safety certification strategy for IEC-61508 compliant industrial mixed-criticality systems based on multicore partitioning, and illustrates it with a safety concept for a wind-turbine mixed-criticality control system. Both the strategy and the example safety concept consider the usage of Commercial off-the-shelf (COTS) multicore processors.

The paper is organized as follows. Section II introduces basic concepts and Section III analyses related work. Section IV describes the proposed safety certification strategy and Section V briefly describes the safety concept. Finally, Section VI draws the overall conclusion and future work.

II. Background

A. Certification standards

IEC-61508 [3], [4], [5] is an international standard for electrical, electronic and programmable electronic safety related systems. IEC-61508 is a generic safety standard from which different domain specific standards have been derived for industrial and transportation domains, e.g. machinery, industry process, automotive, railway, etc.

Safety Integrity Level (SIL) is a discrete level corresponding to a range of safety integrity values where 4 is the highest level an 1 is the lowest. As a rule of thumb, the highest the SIL the highest the certification cost.

B. Fail-safe and fail-operational

Safety systems can be classified as either fail-safe or fail-operational. A system is fail-safe if there is a safe state in the environment that can be reached in case of a system failure either by the safety function or diagnostics, e.g., a process plant can be safely stopped, a train can be stopped, a lift can be stopped, etc. A system is fail operational if no safe state can be reached in case of a system failure, e.g., a flight control system aboard an airplane, drive by wire in a car, etc.

III. Related work

Multiple analyses [6], [7], [8], [9], [10], [11] and research publications [12], [13], [14], [15], [16] indicate that is likely to be a significant increase in the use of multicore devices over the next years replacing applications that have traditionally used single core processors. Multicore and virtualization technology can support the development of mixed-criticality systems by means of software partition, or partition for short. Partitions provide functional separation of the applications and fault containment, to prevent any partitioned application from causing a failure in another partitioned application.

However, the development of safety critical embedded systems based on multicore and virtualization technology is a challenge [17], [18], [19], [20], [21], [22]. Providing sufficient evidence of isolation, separation and independence among safety and non-safety related functions distributed in a multicore processor is not a trivial task [21], [22].

IEC-61508 safety standard does not directly support nor restrict the certification of mixed-criticality systems. Whenever a system integrates safety functions of different criticality, sufficient independence of implementation must be shown among these functions [3], [4]. If there is not sufficient evidence, all integrated functions will need to meet the highest integrity level. Sufficient independence of implementation is established showing that the probability of a dependent failure between the higher and lower integrity parts is sufficiently low in comparison with the highest safety integrity level [4].

Therefore, spatial and temporal isolation are key requirements in mixed-criticality systems because otherwise low criticality applications could interfere with those of high criticality. While spatial isolation can be commonly achieved using state of the art solutions (e.g., MMU), temporal isolation at application level depends on the time guarantees provided by the underlying multicore processor. The usage of time deterministic architectures and processors [19] could simplify the collection of evidences for a certification process because determinism is a sufficient precondition for logical reasoning required for time behaviour analysis [1]. However, most of the existing COTS multicore processors were not designed with a focus on hard-real time applications but towards the maximal average performance. This is the source for multiple temporal isolation challenges [21], [22].

The avionics industry has widely adopted the Integrated Modular Avionics (IMA) [23] architecture, which allows integrating several applications on a single processing element. Applications are encapsulated into partitions that are temporally and spatially isolated from one another, enforcing fault containment [24].

However, the migration of an existing set of pre-certified single-core avionics IMA systems into a multi-IMA multicore system is not a trivial task. The fundamental challenge is to ensure that the temporal and spatial isolation of the partitions will be maintained without incurring huge recertification costs [8], [9], [16], [25], [26], [27], [28], [29].

IV. Safety Certification Strategy

This section describes an IEC-61508 compliant safety certification strategy for mixed-criticality systems based on multicore partitioning, based on the following assumptions:

- The IEC-61508 standard mainly targets fail-safe systems.
- The fault hypothesis defines overall safety assumptions
- An hypervisor ported to a given platform is provided as a certified compliant item
- The hypervisor supports a static cyclic scheduling algorithm with guaranteed time slots defined at design time
- A system level diagnosis strategy is defined

As multicore partitioning based solutions are still not common practice in industry, the strategy shown in Figure 1 considers a three step safety concept transformation from a federated architecture to a multicore integrated architecture.

- Transform federated to multiprocessor: Transform the safety concept of a federated architecture to a multiprocessor safety concept using well known techniques that are common practice in industry
- Transform multiprocessor to multicore: Transform previous safety concept to multicore safety concept still abstracted from detailed analysis of shared-resources. Analyse and select the platform with regard to isolation
- Analyse multicore shared resources: Define, analyse and asses in detail shared resources and their effect

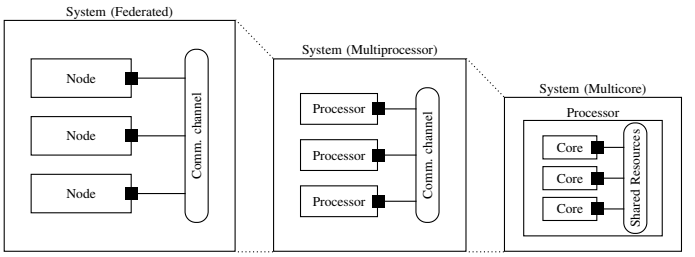


Fig. 1. Safety concept transformation strategy in consecutive steps.

A. IEC-61508 and fail-safe systems

IEC-61508 based safety-critical embedded systems must be developed with a safety life-cycle that aims to reduce the probability of systematic errors and ensure that sufficient fault avoidance and fault control techniques are implemented. Regarding temporal isolation, this means that isolation needs to be systematically guaranteed (or give safe worst case bounds) and diagnosis techniques must be used to detect temporal isolation violations (e.g., watchdog, logic execution, etc.). If this unexpected violation occurs, diagnosis should lead the system to safe-state (e.g., reset). Therefore, the lack of complete temporal isolation would reduce the availability of the system but should not jeopardize safety.

B. Fault hypothesis

The fault-hypothesis [30] of this strategy consists of the following assumptions:

- FSM: All safety relevant systems are developed with an IEC-61508 Functional Safety Management (FSM)
- Node: The node computer forms a single Fault-Containment Region (FCR) that can fail in an arbitrary failure mode. The permanent failure rate is assumed to be in the order of 10-100 FIT (i.e., about one thousand year) and the transient failure rate is assumed to be in the order of 100.000 FIT (i.e., about one year)
- Processor: The multicore processor might not provide complete temporal isolation (or not sufficient evidence for certification), but bounded temporal interference can be estimated and validated with measurements
- Hypervisor: The hypervisor provides interference freeness among partitions (bounded time and spatial isolation), it is certified and fails in an arbitrary failure mode when it is affected by a fault
- Partition: A partition can fail in an arbitrary failure mode, both in the temporal as well as the spatial domain

C. Compliant item: Hypervisor and platform

Hypervisor is a layer of software (or a combination of software / hardware) that allows running several independent execution environments in a single computer platform. Hypervisor solutions such as XtratuM [31] have to introduce a very low overhead compared with other kind of virtualizations (e.g., Java virtual machine); the throughput of the virtual machines has to be very close to that of the native hardware.

The strategy assumes that a hypervisor and platform are provided as a single certified compliant item according to IEC-61508. The safety manual should state that the compliant item provides the following techniques and properties:

- Startup, configuration and initialization: The hypervisor must start up, configure and initialize in a known, repeatable and correct state within a bounded time (e.g., internal data structures, virtualized resource initialization, etc.). Configuration data is static and defined at design stage.
- Virtualization of resources: Provide a virtual environment in a safe, transparent and efficient way (e.g., CPU, memory and Input / Output (I/O) devices)
- Isolation, diagnosis and integrity:
  - Spatial isolation: To prevent one partition from overwriting data in another partition, or a memory address not explicitly assigned to this partition
  - Temporal isolation: To ensure that a partition has sufficient processing time to complete its execution, ensuring that partition cyclic schedule and time slots are assigned as statically configured
  - Health monitoring: To control random and systematic failures at hypervisor or partitions level. Actions to handle these errors are statically defined.
  - Exclusive access to peripherals: Protect access to peripherals used by a safety partition
  - Hypervisor Execution Integrity: The hypervisor execution should be in privileged mode, isolated and protected against external software faults.
- Communication and synchronization:
  - Inter-partition communication: The hypervisor must support mechanisms that allow safe data exchange between two or more partitions
  - Time Synchronization: Fault-tolerant time synchronization that provides a global notion of time to the hypervisor partition scheduler

D. Scheduling

The scheduling of partitions should follow a static cyclic scheduling algorithm with pre-assigned guaranteed time slots defined at design time. The scheduling of partitions among cores should be synchronized based on the global notion of time provided by the hypervisor.

E. Diagnosis strategy

In order to manage the complexity management [1] arising from the safe integration of multiple mixed-criticality partitions, a diagnosis strategy is defined taking into consideration the following assumptions:

- Partitions are developed abstracted from the platform
- The hardware platform provides autonomous hardware diagnosis an diagnosis to be commanded by software
- The execution platform (hardware and hypervisor) is abstracted from the partitions to be executed. The hypervisor provides health monitoring that might be complemented with additional system diagnosis partition(s)
- The system architect is responsible for the architectural design, safety integration and must take care of:
  - Analysing safety manuals of integrated safety partitions and compliant items
  - Selection of partitions and diagnosis partitions
  - Defining the design time static configuration, e.g., scheduling and allocation of resources

Based on this assumptions, the recommended diagnosis strategy is described below:

- The partition should be self contained and should provide safety life-cycle related techniques (e.g., IEC-61508-3 Table A.4 defensive programming) and platform independent diagnosis (e.g., IEC-61508-2 Table A.7 input comparison voting) abstracted from the details of the underlying platform
- The hardware provides autonomous diagnosis (e.g., IEC-61508-2 Table A.9 Power Failure Monitor (PFM)) and diagnosis components to be commanded by software (e.g., IEC-61508-2 Table A.10 watchdog)
- The hypervisor and associated diagnosis partitions should support platform related diagnosis (e.g., IEC-61508-2 Table A.5 signature of a double word)
- The system architect specifies and integrates additional diagnosis partitions required to develop a safe product taking into consideration all safety manuals

V. Case Study

This section briefly describes a case-study where previously defined safety certification strategy (Section IV) is applied for the definition of a safety concept for a mixed-criticality wind power control based on multicore partitioning.

A wind park is composed of interconnected wind turbines and a centralized wind park control center as shown in Figure 2. As previously explained current wind turbine control unit follows a federated architectural approach and provides three major functionalities:

- 'Supervision': Wind turbine real-time control and supervision.
- 'SCADA': Non real-time Human Machine Interface (HMI) and communication with SCADA system
- 'Safety Protection': Safety functions that ensure that design limits of the wind turbine are not exceeded

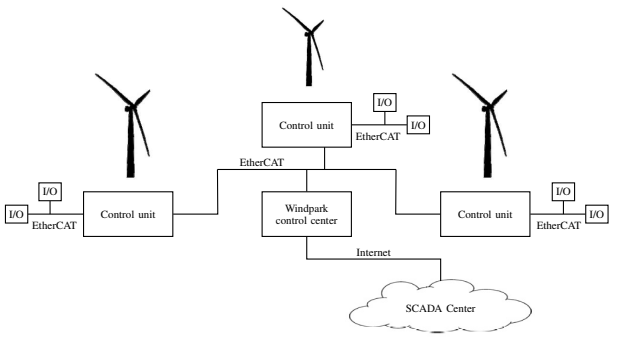


Fig. 2. Simplified wind park diagram.

The safety protection system must ensure that design limits of the wind turbine are not exceeded (e.g., over speed) and if exceeded output safety-relays connected to the safety-chain must be opened. As shown in Figure 3, there is a safety-chain composed of safety-relays in serial that activates the 'pitch control' safety function whenever the chain is opened. The 'pitch control' safety function leads the wind turbine to a safe-state within a Process Safety Time (PST). The safety protection system must meet 'PLd' level of ISO-13849 [32] and IEC-61508 SIL2/3.

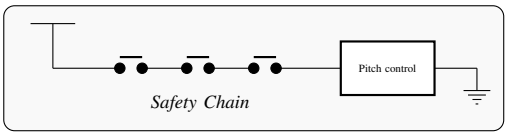


Fig. 3. Wind turbine safety chain.

A. Safety concept

This section describes the safety concept of a mixed-criticality wind power system based on multicore and virtualization partitioning.

1) Transformation (Federated to multiprocessor): The first step is to transform a subset of the current federated architecture into an integrated architecture based on two or more processors. The safety concept behind the architecture shown in Figure 4 is common practice in industry: $1oo2(D)$ dual-channel architecture based on two independent processors, two shared diverse input sources (rotation speed) and two output-relays connected in serial to the safety-chain.

The node has a Hardware Fault Tolerance (HFT) of one ($HFT = 1$)) based on two independent processors. Each processor controls one independent safety-relay that can be de-activated (safe-state) either directly commanded by 'safety protection' or indirectly by 'diagnosis'. If the 'diagnosis' detects a fatal error, it does not refresh the associated watchdog and this leads to a reset of the node. As a summary:

- '$P0$' and '$P1$' are independent single core processors
- '$P0$' processor executes safety related partitions only: 'safety protection' and 'diagnosis'
- '$P1$' processor executes all partitions
- Each processor controls one independent safety-relay
- EtherCAT 'communication stack' is managed in $P1$ and the safety-communication layer in 'safety protection'
- Local and cross-channel 'diagnosis' in each processor
- An independent 'watchdog' monitors each processor
- An IEC-61508 SIL2 system with $HFT = 1$ requires a Safe Failure Fraction (SFF) of $90\% > SFF >= 60\%$

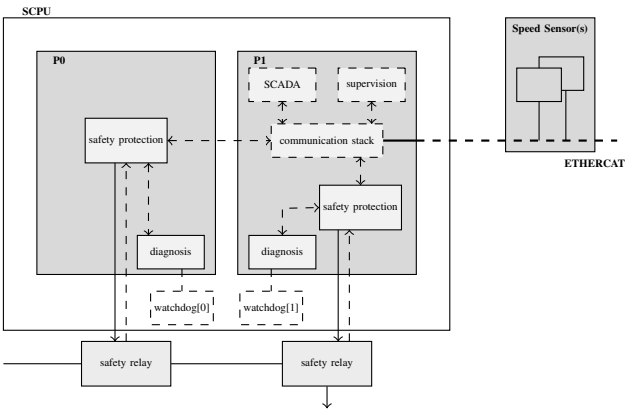


Fig. 4. Safety concept($1oo2$; 2 processors)

The future scalability of this approach is also limited. The number of integrated functionalities will continue to increase, but the usage of fans is not allowed in order to meet reliability and availability requirements. The computation power of the single core processor is limited and if processor '$P1$' does not provide sufficient computation power new processors will be need to be added. Adding new processors and their associated communication buses leads to additional reliability and availability issues (e.g., material reliability, EMC, etc.).

2) Transformation (multiprocessor to multicore): Previous multiprocessor based safety concept shown in Figure 4 is transformed into a multicore architecture shown in Figure 5.

At this abstraction level, different platforms are analysed taking into consideration features such as safety, computation, memory, communication, isolation, etc. The theoretical analysis based on available documentation must be validated with experimental evaluation. The mapping of partitions to cores can also be modified according to platforms specific constraints and properties. The selected platform shown in Figure 5 is an heterogeneous quadcore processor (two 'x86' cores and two 'LEON3 FT' softcores), that meets application requirements, application dependencies with 'x86' architecture and has been positively assessed [33].

In addition to this, the diagnosis strategy defined in the previous transformation needs to be reviewed taking into consideration the details of the new platform. For example, a single processor node requires a processor that meets IEC-61508-2 Annex E in order to claim a $HFT = 1$ and this is not common for COTS processors. If this claim does not hold, a higher SFF is required (a IEC-61508 SIL2 system with $HFT = 0$ requires $99\% > SFF >= 90\%$), which implies additional diagnosis techniques and updates in previously selected ones.

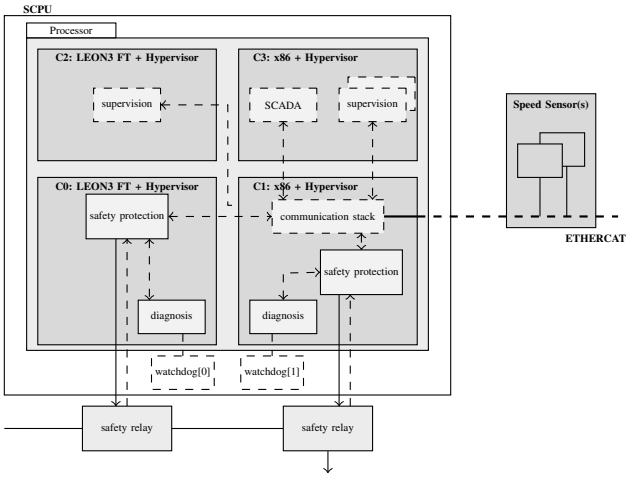


Fig. 5. Simplified safety concept (1oo2), multicore).

3) Shared resources: Figure 6 shows the detailed processor diagram taking into consideration major shared-resources. The real platform is composed of two commercial nodes, a dual-core Intel Atom processor connected via PCIe to an FPGA that integrates two 'LEON3 FT' softcores. For the purpose of this analysis, they are considered to be a single silicon rather than two independent silicon. 'LEON3 FT' softcores have associated a local memory for program and data ('LS memory') and use an external shared memory ('external shared memory') for inter-partition communication. 'x86' cores have $L1/L2$ cache and share and external memory ('external shared memory 2'). Communication among partitions allocated in 'x86' and 'LEON3 FT' cores is implemented using an external shared memory accessed by a shared bus (AHB bus - gateway - PCIe). A periodic interrupt common to all cores is used for hypervisor time synchronization purposes.

The extended safety concept includes FMEAs, error reaction

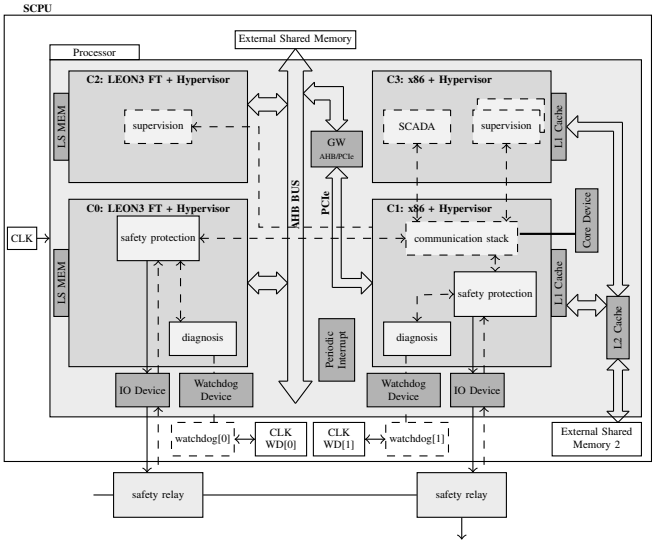


Fig. 6. Safety concept (1oo2), multicore with shared resources).

definitions and it is complemented with a detailed assessment of the platform [33]. Spatial isolation was positively assessed. However, it was concluded that temporal characteristics of partitions could be influenced by different loads scenarios in other partitions due to shared resources. For example:

1) Shared memory: x86' cores use shared-memory and 'LEON3 FT' cores use shared memory for inter-partition communication. Maximum temporal interference suffered by a partition is estimated and measured
2) Shared cache: Atom processor (dual core 'x86') does not support temporal freeness in shared cache, the maximum temporal interference suffered by a partition is measured
3) Interrupts: Some interrupts in the Atom processor can not be rerouted and this can influence the timing behaviour of the hypervisor, the maximum temporal interference suffered by a partition is measured
4) Communication channel: Complete decoupling of sender and receiver partitions connected with a communication channel require temporal isolation

Different solutions are defined in order to avoid and control failures due to previously described temporal interferences:

- Fault avoidance
  - Shared-resources: 'Safety protection' and 'diagnosis' partition Worst Case Execution Time (WCET) are measured for each core type ('x86' and 'LEON3 FT'). Both partitions are scheduled at the beginning of each periodic cycle with a pre-assigned time-slot bigger than the maximum estimated execution time, which considers both the WCET and maximum estimated time interference due to shared resources
  - Interrupts: All unused interrupts are routed to 'diagnosis' or health monitoring
  - Communication channel: The communication among 'safety protection' and 'diagnosis' partitions in different cores is delayed one execution cycle, which

it is considered sufficient to diminish temporal interferences due to shared resources.

- Fault control:
  - Shared-resources: Safety partitions are executed in two diverse cores ('x86' and 'LEON3 FT') with different hypervisor configuration. Each 'diagnosis' partition refresh an independent watchdog if monitored-time constraints are met.
  - Interrupts: 'Diagnosis' partition traps unused interrupts and decides whether to refresh an independent watchdog based on the severity of the error
  - Communication channel: Safety partitions monitor communication channel time-outs.

VI. Conclusion and Future Work

While mixed-criticality paradigm based on multicore and partitioning provides multiple potential benefits, it is clear that the safety certification of such systems based on COTS multiprocessors not designed for safety is a challenge.

This paper has contributed with a safety-certification strategy for IEC-61508 based safety systems based on COTS multiprocessors that have been illustrated with a safety concept currently under detailed review by a certification body. The assumptions and analysis considered at this stage will be reviewed in the following design stages and validated at the final stage of the case-study within FP7 MultiPARTES project.

Acknowledgement

This work has been supported in part by the European projects FP7 MultiPARTES and FP7 DREAMS under project No. 287702 and No. 610640 respectively and the national INNPACTO project VALMOD under grant number IPT-2011-1149-370000. Any opinions, findings and conclusions expressed in this article are those of the authors and do not necessarily reflect the views of funding agencies. The authors would like to thank Alfons Crespo from UPV, XtratuM team and TU Wien.

References

[1] H. Kopetz, "The complexity challenge in embedded system design," in *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, 2008, pp. 3–12.

[2] J. Swingler and J. W. McBride, "The degradation of road tested automotive connectors," in *Forty-Fifth IEEE Holm Conference on Electrical Contacts*, 1999, pp. 146–152.

[3] IEC, "IEC 61508-1: Functional safety of electrical/electronic/programmable electronic safety-related systems part 1: General requirements," 2010.

[4] ——, "IEC 61508-2: Functional safety of electrical/electronic/programmable electronic safety-related systems part 2: Requirements for electrical / electronic / programmable electronic safety-related systems," 2010.

[5] ——, "IEC 61508-3: Functional safety of electrical/electronic/programmable electronic safety-related systems part 3: Software requirements," 2010.

[6] "Mixed criticality systems," European Comission, Tech. Rep., February 3 2012.

[7] "MULCORS - use of multicore processors in airborne systems (research project EASA.2011/6)," EASA, Tech. Rep., 16th December 2012.

[8] EASA, "Certification memorandum - software aspects of certification - EASA CM SWCEH 002," Tech. Rep., 9th March 2013.

[9] ——, "Development assurance of airborne electronic hardware," 2011.

[10] S. Balacco and C. Rommel, "Next generation embedded hardware architectures: Driving onset of project delays, costs overruns and software development challenges," Klockwork, Inc., Tech. Rep., September 2010.

[11] "2013 - embedded market study," UBM Tech, Tech. Rep., 2013.

[12] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, "Mixed-criticality real-time scheduling for multicore systems," pp. 1864–1871, 2010.

[13] R. Ernst, "Certification of trusted MPSoC platforms," in *MPSoC Forum*, 2010.

[14] H. Kopetz, R. Obermaisser, C. El Salloum, and B. Huber, "Automotive software development for a multi-core system-on-a-chip," in *Fourth International Workshop on Software Engineering for Automotive Systems (ICSE Workshops SEAS)*, 2007, pp. 2–9.

[15] D. Gonzalez, J. M. Garate, A. Trapman, L. Monsalve, and S. Trujillo, "Mixed-criticality in wind power: The MultiPARTES approach," in *ESReDA Conference 2012*, 2012, p. 9.

[16] X. Jean, M. Gatti, G. VBerthon, and M. Fumey, "The use of multicore processors in airborne systems," Thales Avionics, Tech. Rep., 2011.

[17] J. Schneider, M. Bohn, and R. Rbger, "Migration of automotive real-time software to multicore systems: First steps towards an automated solution," in *22nd EUROMICRO Conference on Real-Time Systems*, 2010.

[18] R. Fuchsen, "How to address certification for multi-core based IMA platforms: Current status and potential solutions," in *IEEE/AIAA 29th Digital Avionics Systems Conference (DASC)*, 2010, pp. 5.E.3–1–5.E.3–11.

[19] C. E. Salloum, M. Elshuber, O. Hoftberger, H. Isakovic, and A. Wasicek, "The ACROSS MPSoC – a new generation of multi-core processors designed for safety-critical embedded systems," in *Digital System Design (DSD), 2012 15th Euromicro Conference on*, 2012, pp. 105–113.

[20] J. Abella, F. J. Cazorla, E. Quinones, A. Grasset, S. Yehia, P. Bonnot, D. Gizopoulos, R. Mariani, and G. Bernat, "Towards improved survivability in safety-critical systems," in *IEEE 17th International On-Line Testing Symposium (IOLTS)*, 2011, pp. 240–245.

[21] O. Kotaba, J. Nowotsch, M. Paulitsch, S. M. Petters, and H. Theilingx, "Multicore in real-time systems temporal isolation challenges due to shared resources," in *Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems (WICERT)*, 2013.

[22] R. Nevalainen, O. Slotosch, D. Truscan, U. Kremer, and V. Wong, "Impact of multicore platforms in hardware and software certification," in *Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems (WICERT)*, 2013.

[23] "RTCA DO-297 integrated modular avionics (IMA) development guidance and certification considerations," 2005.

[24] X. Jean, D. Faura, M. Gatti, L. Pautet, and T. Robert, "Ensuring robust partitioning in multicore platforms for ima systems," in *IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, 2012, pp. 7A41–7A49, export Date: 27 June 2013 Source: Scopus Art. No.: 6382408.

[25] J.-E. Kim, M.-K. Yoon, S. Im, R. Bradford, and L. Sha, "Optimized scheduling of multi-IMA partitions with exclusive region for synchronized real-time multi-core systems," pp. 970–975, 2013.

[26] L. M. Kinnan, "Use of multicore processors in avionics and its potential impact on implementation and certification," *SAE Technical Papers*, 2009.

[27] P. Huyck, "Arinc 653 and multi-core microprocessors - considerations and potential impacts," in *IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, 2012, pp. 6B41–6B47.

[28] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *Dependable Computing Conference (EDCC), 2012 Ninth European*, 2012, pp. 132–143.

[29] S. Fisher, "Certifying applications in a multi-core environment: a new approach gains success," SYSGO AG, Tech. Rep.

[30] H. Kopetz, *On the Fault Hypothesis for a Safety-Critical Real-Time System*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, vol. 4147, ch. 3, pp. 31–42.

[31] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The XtratuM approach," in *European Dependable Computing Conference (EDCC)*, 2010, pp. 67–72.

[32] IEC, "ISO 13849-1: Safety of machinery - safety-related parts of control systems ," p. 58, 2002.

[33] C. Helpa and H. Isakovic, "D3.5 - assesment of the MultiPARTES platform," TU Wien, Tech. Rep., 2013.

The Quest-V Separation Kernel for Mixed Criticality Systems

Ye Li, Richard West, and Eric Missimer

Computer Science Department
Boston University
Boston, MA 02215
Email: {liye,richwest,missimer}@cs.bu.edu

***Abstract*—Multi- and many-core processors are becoming increasingly popular in embedded systems. Many of these processors now feature hardware virtualization capabilities, such as the ARM Cortex A15, and x86 processors with Intel VT-x or AMD-V support. Hardware virtualization offers opportunities to partition physical resources, including processor cores, memory and I/O devices amongst guest virtual machines. Mixed criticality systems and services can then co-exist on the same platform in separate virtual machines. However, traditional virtual machine systems are too expensive because of the costs of trapping into hypervisors to multiplex and manage machine physical resources on behalf of separate guests. For example, hypervisors are needed to schedule separate VMs on physical processor cores. In this paper, we discuss the design of the Quest-V separation kernel, that partitions services of different criticalities in separate virtual machines, or *sandboxes*. Each sandbox encapsulates a subset of machine physical resources that it manages without requiring intervention of a hypervisor. Moreover, a hypervisor is not needed for normal operation, except to bootstrap the system and establish communication channels between sandboxes.**

I. Introduction

Embedded systems are increasingly featuring multi- and many-core processors, due in part to their power, performance and price benefits. These processors offer new opportunities for an increasingly significant class of mixed criticality systems. In mixed criticality systems, there is a combination of application and system components with different safety and timing requirements. For example, in an avionics system, the in-flight entertainment system is considered less critical than that of the flight control system. Similarly, in an automotive system, infotainment services (navigation, audio and so forth) would be considered less timing and safety critical than the vehicle management sub-systems for anti-lock brakes and traction control.

A major challenge to mixed criticality systems is the safe isolation of separate components with different levels of criticality. Isolation has traditionally been achieved by partitioning components across distributed modules, which communicate over a network such as a CAN bus. For example, Integrated Modular Avionics (IMA) [1] is used to describe a distributed real-time computer network capable of supporting applications of differing criticality levels aboard an aircraft. To implement such concepts on a multi-core platform, a software architecture that enforces the safe isolation of system components is required.

The notion of component isolation, or partitioning, is the basis of software system standards such as ARINC 653 [2] and the Multiple Independent Levels of Security (MILS) [3] architecture. Current implementations of these standards into operating system kernels [4][5][6] have focused on micro-kernel and virtual machine technologies for resource partitioning and component isolation. However, due to hardware limitations and software complexity, these systems either cannot completely eliminate covert channels of communication between isolated components or add prohibitive performance overhead.

Hardware-assisted virtualization provides an opportunity to efficiently separate system components with different levels of safety, security and criticality. Back in 2006, Intel and AMD introduced their VT-x and AMD-V processors, respectively, with support for hardware virtualization. More recently, the ARM Cortex A15 was introduced with hardware virtualization capabilities, for use in portable tablet devices. Similarly, some Intel Atom chips now have VT-x capabilities for use in automobile In-Vehicle Infotainment (IVI) systems [7], and other embedded systems.

While modern hypervisor solutions such as Xen [8] and Linux-KVM [9] leverage hardware virtualization to isolate their guest systems, they are still required for CPU and I/O resource multiplexing. Expensive traps into the hypervisor occur every time a guest system needs to be scheduled, or when an I/O device transfers data to or from a guest. This situation is both unnecessary and too costly for mixed criticality systems with real-time requirements.

In this paper we present a new operating system design leveraging hardware-assisted virtualization as an extra *ring of protection*, to achieve efficient resource partitioning and performance isolation for subsystems. Our sys-

This work is supported in part by NSF Grant #1117025.

tem, called Quest-V, is a separation kernel [10] design. The system avoids traps into a hypervisor (a.k.a. virtual machine monitor, or VMM) when making scheduling and I/O management decisions. Instead, all resources are partitioned at boot-time amongst system components that are capable of scheduling themselves on available processor cores. Similarly, system components are granted access to specific subsets of I/O devices and memory so that devices can be managed without involvement of a hypervisor.

In the rest of this paper, we describe how Quest-V can be configured to support a Linux front-end responsible for low criticality legacy services, while native Quest services can operate on separate hardware resources (memory, I/O and CPU cores) to ensure safe, predictable and efficient service guarantees. In this way, high criticality Quest services can co-exist with less critical Linux services on the same hardware platform.

II. Quest-V Separation Kernel Architecture

Quest-V partitions a system into a series of *sandbox kernels*, with each sandbox encompassing a subset of memory, I/O and CPU resources. The current implementation works on Intel VT-x platforms but plans are underway to port Quest-V to the AMD-V and ARM architectures.

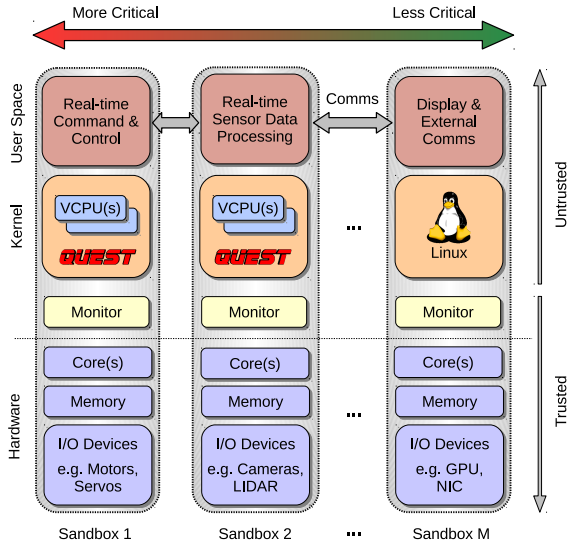


Fig. 1. Example Quest-V Architecture Overview

A high-level overview of the Quest-V architecture is shown in Figure 1. Trusted monitor code is used to launch *guest* services, which may include their own kernels and user space programs. A monitor is responsible for managing special *extended page tables* (EPTs) that translate guest physical addresses (GPAs) to host physical addresses (HPAs), as described later in Figure 2.

We chose to have a separate monitor for each sandbox, so that it manages only one set of EPT memory mappings for a single guest environment. The amount of added overhead of doing this is small, as each monitor's code fits within $4KB$ [1]. However, the benefits are that monitors are made much simpler, since they know which sandbox they are serving rather than having to determine at runtime the guest that needs their service. Typically, guests do not need intervention of monitors, except to establish shared memory communication channels with other sandboxes, which requires updating EPTs.

Mixed-Criticality Example – Figure 1 shows an example of three sandboxes, where two are configured with Quest native safety-critical services for command, control and sensor data processing. These services might be appropriate for a future automotive system that assists in vehicle control. Other less critical services could be assigned to vehicle infotainment services, which are partitioned in a sandbox that has access to a local display device. A non-real-time Linux system could be used in this case, perhaps managing a network interface (NIC) to communicate with other vehicles or the surrounding environment, via a vehicle-to-vehicle (V2V) or vehicle-to-infrastructure (V2I) communication link.

A. Resource Partitioning

Resource partitioning in Quest-V is mostly static, and takes place at boot-time. However, communication channels between sandboxes can be established at runtime, requiring some dynamic memory allocation.

CPU Partitioning – In Quest-V, scheduling is performed within each sandbox. Since processor cores are statically allocated to sandboxes, there is no need for monitors to perform sandbox scheduling as is typically required with traditional hypervisors. This approach eliminates most of the monitor traps otherwise necessary for sandbox context switches. It also means there is no notion of a global scheduler to manage the allocation of processor cores amongst guests. Each sandbox's local scheduler is free to implement its own policy, simplifying resource management. This approach also distributes contention amongst separate scheduling queues, without requiring synchronization on one global queue.

Memory Partitioning – Quest-V relies on hardware assisted virtualization support to perform memory partitioning. Figure 2 shows how address translation works for Quest-V sandboxes using Intel's extended page tables. Each sandbox kernel uses its own internal paging

[1]The EPTs take additional data space, but 12KB is enough for a 1GB sandbox.

structures to translate guest virtual addresses to guest physical addresses. EPT structures are then walked by the hardware to complete the translation to host physical addresses.

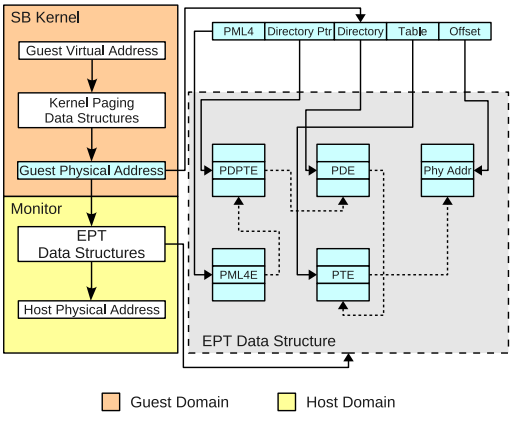


Fig. 2. Extended Page Table Mapping

On modern Intel x86 processors with EPT support, address mappings can be manipulated at 4KB page granularity. For each 4KB page we have the ability to set read, write and even execute permissions. Consequently, attempts by one sandbox to access illegitimate memory regions of another will incur an EPT violation, causing a trap to the local monitor. The EPT data structures are, themselves, restricted to access by the monitors, thereby preventing tampering by sandbox kernels.

EPT mappings are cached by hardware TLBs, expediting the cost of address translation. Only on returning to a guest after trapping into a monitor are these TLBs flushed. Consequently, by avoiding exits into monitor code, each sandbox operates with similar performance to that of systems with conventional page-based virtual address spaces [11].

Cache Partitioning – Microarchitectural resources such as caches and memory buses provide a source of contention on multicore platforms. Using hardware performance counters we are able to establish cache occupancies for different sandboxes [12]. Also, memory page coloring can be used to partition shared caches [13].

I/O Partitioning – In Quest-V, device management is performed within each sandbox directly. Device interrupts are delivered to a sandbox kernel without monitor intervention. This differs from the "split driver" model of systems such as Xen, which have a special domain to handle interrupts before they are directed into a guest. Allowing sandboxes to have direct access to I/O devices greatly reduces the overhead of monitor traps to handle interrupts.

To partition I/O devices, Quest-V first has to restrict access to device specific hardware registers. Device registers are usually either memory mapped or accessed through a special I/O address space (e.g. I/O ports). For the x86, both approaches are used. For memory mapped registers, EPTs are used to prevent their accesses from unauthorized sandboxes. For port-addressed registers, special hardware support is necessary. On Intel processors with VT-x, all variants of `in` and `out` instructions can be configured to cause a monitor trap if access to a certain port address is attempted. As a result, a hardware provided I/O bitmap can be used to partition the whole I/O address space amongst different sandboxes. Unauthorized access to a certain register can thus be ignored or trigger a fault recovery event.

Any sandbox attempting access to a PCI device must use memory-mapped or port-based registers identified in a special PCI *configuration space* [14]. Quest-V intercepts access to this configuration space, which is accessed via both an address and data I/O port. A trap to the local sandbox monitor occurs when there is a PCI data port access. The monitor then determines which device's configuration space is to be accessed by the trapped instruction. A device *blacklist* for each sandbox containing the *Device ID* and *Vendor ID* of restricted PCI devices is used by the monitor to control actual device access.

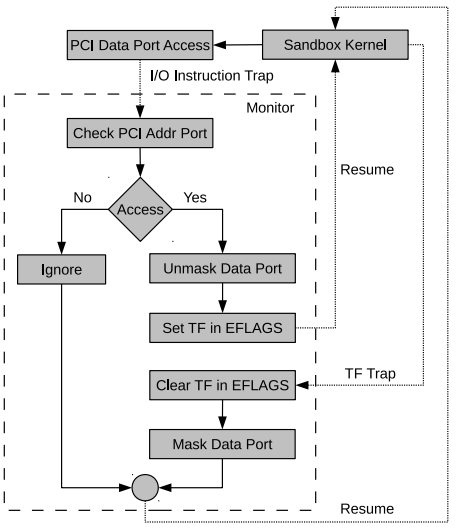


Fig. 3. PCI Configuration Space Protection

A simplified control flow of the handling of PCI configuration space protection in a Quest-V monitor is given in Figure 3. Notice that simply allowing access to a PCI data port is not sufficient because we only want to allow the single I/O instruction that caused the

monitor trap, and which passed the monitor check, to be correctly executed. Once this is done, the monitor should immediately restrict access to the PCI data port again. This behavior is achieved by setting the *trap flag* (TF) bit in the sandbox kernel system flags to cause a single step debug exception after it executes the next instruction. By configuring the processor to generate a monitor trap on debug exception, the system can immediately return to the monitor after executing the I/O instruction. After this, the monitor is able to mask the PCI data port again for the sandbox kernel, to mediate future device access.

In addition to direct access to device registers, interrupts from I/O devices also need to be partitioned amongst sandboxes. In modern multi-core platforms, an external interrupt controller is almost always present to allow configuration of interrupt delivery behaviors. On modern Intel x86 processors, this is done through an I/O Advanced Programmable Interrupt Controller (IOAPIC). Each IOAPIC has an I/O *redirection table* that can be programmed to deliver device interrupts to all, or a subset of, sandboxes. Each entry in the I/O redirection table corresponds to a certain interrupt request from an I/O device on the PCI bus.

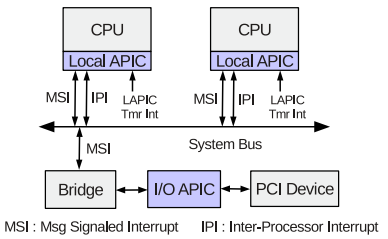


Fig. 4. APIC Configuration

Figure 4 shows the hardware APIC configuration. Quest-V uses EPT entries to restrict access to memory regions, used to access IOAPIC registers. This way, the IOAPIC interrupt redirection table is prevented from alteration by an unauthorized sandbox. Attempts by a sandbox to access the IOAPIC space cause a trap to the local monitor as a result of an EPT violation. The monitor then checks to see if the sandbox has authorization to update the table before allowing any changes to be made. Consequently, device interrupts are safely partitioned amongst sandboxes.

B. Linux Sandbox Support

In addition to native Quest kernels, Quest-V is also designed to support other third party sandbox kernels such as Linux and AUTOSAR OS. Currently, we have successfully ported a Puppy Linux distribution with Linux 3.8 kernel to serve as our system front-end, providing a window manager and graphical user interface. A Quest kernel is first started in the Linux sandbox to bootstrap our paravirtualized Linux kernel. Because Quest-V exposes maximum possible privilege of hardware access to sandbox kernels, the actual changes made to the original Linux 3.8 kernel are well under 100 lines. These changes are mainly focused on limiting Linux's view of available memory and handling I/O device DMA offsets caused by memory virtualization.

The VGA frame buffer and GPU hardware are always assigned to the bootstrapped Linux sandbox. All the other sandboxes will have their default terminal I/O tunneled through shared memory channels to virtual terminals in the Linux front-end. We developed libraries, user space applications and a kernel module to support this redirection in Linux. A screen shot of Quest-V after booting the Linux front-end sandbox is shown in Figure 5. Here, we show two virtual terminals connected to two different native Quest sandboxes similar to the configuration shown in Figure 1. In this particular example, we allocated 512MB to the Linux sandbox (including an in-memory root filesystem) and 256MB to each native Quest sandbox. The network interface card has been assigned to Quest sandbox 1, while the serial device has been assigned to Quest sandbox 2. The Linux sandbox is granted ownership of the USB host controller in addition to the GPU and VGA frame buffer. Observe that although the machine has four processor cores, the Linux kernel detects only one core.

C. VCPU Scheduling

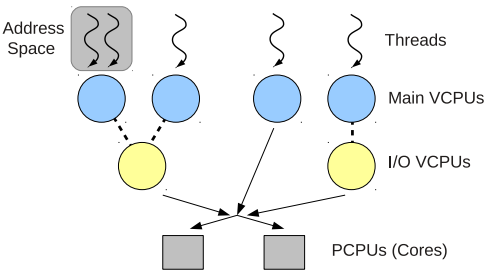


Fig. 6. VCPU Scheduling Hierarchy

Sandboxes running Quest kernels use a form of *virtual* CPU (VCPU) scheduling for conventional tasks and interrupt handlers [15]. Two classes of VCPUs exist, as shown in Figure 6: (1) *Main VCPUs* are used to schedule and track the PCPU usage of conventional software threads, while (2) *I/O VCPUs* are used to account for, and schedule the execution of, interrupt handlers for I/O devices. This distinction allows for interrupts from

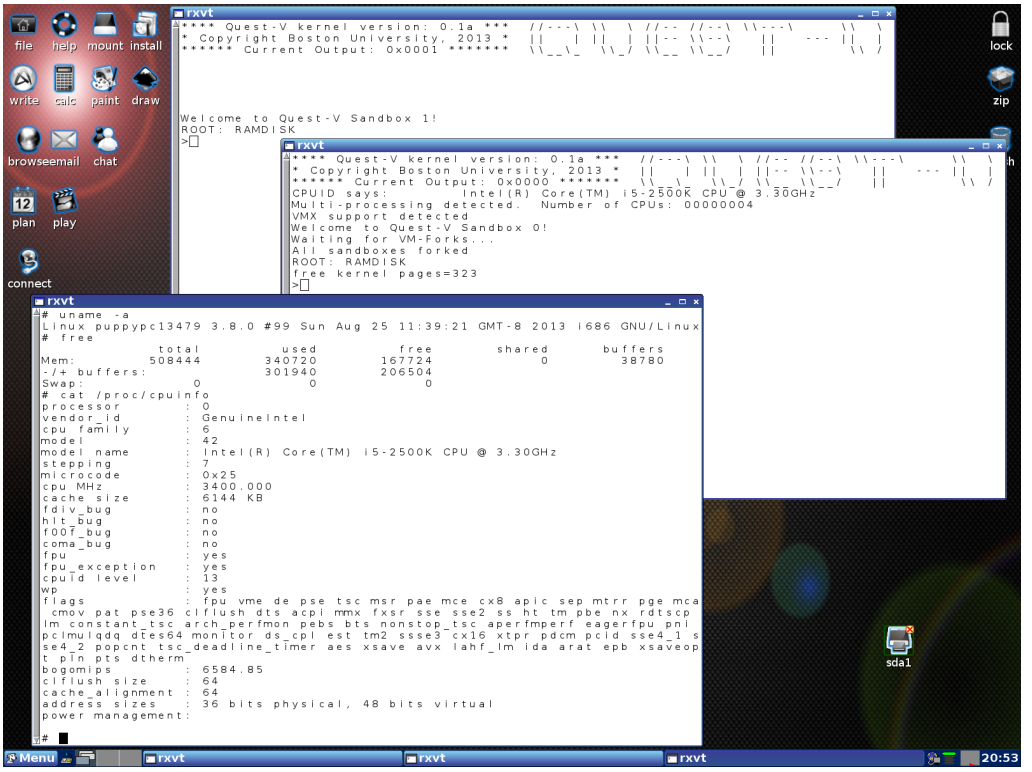


Fig. 5. Quest-V with Linux Front-End

I/O devices to be scheduled as threads, which may be deferred execution when threads associated with higher priority VCPUs having available budgets are runnable. I/O VCPUs can be specified for certain devices, or for certain tasks that issue I/O requests, thereby allowing interrupts to be handled at different priorities and with different CPU shares than conventional tasks associated with Main VCPUs.

By default, VCPUs act like Sporadic Servers [16], [17]. Local APIC timers are programmed to replenish VCPU budgets as they are consumed during thread execution. Sporadic Servers enable a system to be treated as a collection of equivalent periodic tasks scheduled by a rate-monotonic scheduler (RMS) [18]. This is significant, given I/O events can occur at arbitrary (aperiodic) times, potentially triggering the wakeup of blocked tasks (again, at arbitrary times) having higher priority than those currently running. RMS analysis can be applied, to ensure each VCPU is guaranteed its share of CPU time, V_U, in finite windows of real-time.

III. Experimental Evaluation

Figure 7 shows the performance of a video benchmark using a non-virtualized Linux system (labelled *Linux*), and a Linux front-end running in Quest-V (labelled

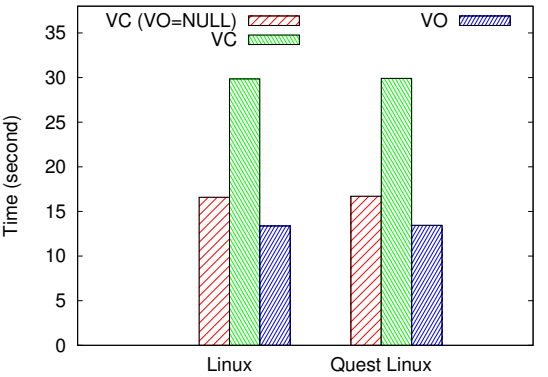


Fig. 7. mplayer HD Video Benchmark

Quest Linux). This experiment was run on a mini-ITX machine with an Intel Core i5-2500K 4-core processor, featuring 4GB RAM. The figure shows the time spent in the video codec (VC) and the video output (VO), to play back a 1080P MPEG2 video from the x264 HD Benchmark.

In Quest-V, the Linux sandbox has exclusive control over the integrated HD Graphics 3000 GPU. The video is about 2 minutes long and the file is 102MB in size. During normal playback, both Linux and Quest-

Linux produce the same frame rate. By switching to benchmark mode with no audio (invoking mplayer with `-benchmark` and `-nosound`), mplayer will try to decode and display each frame as fast as possible. With the `-vo=null` argument, mplayer will further skip the video output and try to decode as fast as possible. The results show that Quest-V virtualization inflicts almost no overhead for HD video decode and display in Linux.

IV. Related Work

Xen [8], Linux-KVM [9], XtratuM [6] and the Wind River Hypervisor [19] all use virtualization technologies to logically isolate and multiplex guest virtual machines on a shared set of physical resources. LynxSecure [5] is another similar approach targeted at safety-critical real-time systems.

PikeOS [4] is a separation micro-kernel [20] that supports multiple guest VMs, and targets safety-critical domains such as Integrated Modular Avionics. The micro-kernel supports a virtualization layer that is required to manage the spatial and temporal partitioning of resources amongst guests.

In contrast to the above systems, Quest-V statically partitions machines resources into separate sandboxes. Services of different criticalities can be mapped into separate sandboxes. Each sandbox manages its own resources independently of an underlying hypervisor. Quest-V also avoids the need for a split-driver model involving a special domain (e.g., `Dom0` in Xen) to handle device interrupts. Interrupts are delivered directly to the sandbox associated with the corresponding device, using I/O passthrough.

V. Conclusions and Future Work

This paper introduces Quest-V, which is an open-source separation kernel built from the ground up. It uses hardware virtualization in a first-class manner to separate system components of different criticalities. It avoids traditional costs associated with hypervisor systems, by statically allocating partitioned subsets of machine resources to guest sandboxes, which perform their own scheduling, memory and I/O management. Hardware virtualization simply provides an extra logical ring of protection. Sandboxes can communicate via shared memory channels that are mapped to immutable entries in extended page tables (EPTs). Only trusted monitors are capable of changing the entries in these EPTs, preventing access to arbitrary memory regions in remote sandboxes.

Quest-V requires system monitors to be trusted. Although these occupy a small memory footprint and are not involved in normal system operation, the system can be compromised if the monitors are corrupted. Future work will investigate the use of hardware features such as Intel's trusted execution technology (TXT) to enforce safety of the monitors. Additionally, online fault detection and recovery strategies will be considered.

Please see `www.questos.org` for more details.

References

[1] C. B. Watkins, "Integrated Modular Avionics: Managing the allocation of shared intersystem resources," in *Proceedings of the 25th Digital Avionics Systems Conference*, pp. 1–12, 2006.

[2] "ARINC 653 - an avionics standard for safe, partitioned systems." Wind River Systems / IEEE Seminar, August 2008.

[3] J. Alves-foss, W. S. Harrison, P. Oman, and C. Taylor, "The MILS architecture for high-assurance embedded systems," *International Journal of Embedded Systems*, vol. 2, pp. 239–247, 2006.

[4] "SYSGO PikeOS." http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept.

[5] "LynxSecure Embedded Hypervisor and Separation Kernel." http://www.lynuxworks.com/virtualization/hypervisor.php.

[6] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The xtratum approach.," in *EDCC*, pp. 67–72, 2010.

[7] "Intel In-Vehicle Infotainment (IVI) system." http://www.intel.com/content/www/us/en/intelligent-systems/in-vehicle-infotainment/in-vehicle-infotainment-in-car-entertainment-with-intel-inside.html.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp. 164–177, 2003.

[9] I. Habib, "Virtualization with kvm," *Linux J.*, vol. 2008, no. 166, p. 8, 2008.

[10] J. M. Rushby, "Design and verification of secure systems," in *Proceedings of the eighth ACM symposium on Operating systems principles*, pp. 12–21, 1981.

[11] Y. Li, R. West, E. Missimer, and M. Danish, "Quest-V: A virtualized multikernel for high-confidence embedded systems." http://www.cs.bu.edu/fac/richwest/papers/quest-v.pdf.

[12] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang, *Multicore Technology: Architecture, Reconfiguration and Modeling*, ch. 8. CRC Press, ISBN-10: 1439880638, 2013.

[13] J. Liedtke, H. Härtig, and M. Hohmuth, "OS-controlled cache predictability for real-time systems," in *the 3rd IEEE Real-time Technology and Applications Symposium*, 1997.

[14] PCI: http://wiki.osdev.org/PCI.

[15] M. Danish, Y. Li, and R. West, "Virtual-CPU Scheduling in the Quest Operating System," in *Proceedings of the 17th Real-Time and Embedded Technology and Applications Symposium*, pp. 169–179, 2011.

[16] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *Real-Time Systems Journal*, vol. 1, no. 1, pp. 27–60, 1989.

[17] M. Stanovich, T. P. Baker, A. I. Wang, and M. G. Harbour, "Defects of the POSIX sporadic server and how to correct them," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010.

[18] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[19] http://www.windriver.com/products/hypervisor/.

[20] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pp. 207–220, 2009.

Memory Architectures for NoC-Based Real-Time Mixed Criticality Systems

Neil C. Audsley
Department of Computer Science,
University of York, UK
email: neil.audsley@york.ac.uk

Abstract

Mixed criticality systems (MCS) allow software components of differing criticalities to use the same physical resources (ie. CPU, memory). MCS highlight the trade-off between partitioning components of different criticalities and efficient resource usage. Components are partitioned due to safety concerns, but physical partitioning requires more resources than if components are unpartitioned and share resources.

Influential recent work in scheduling of MCS shows the benefits of sharing resources between criticality levels. One issue with this work is that allowing resource sharing within MCS requires that sufficient partitioning and separation can be provided by the architecture to ensure mixed criticality components can share resources without compromising system safety at the highest criticality levels.

This paper examines this issue from the perspective of the memory hierarchy for Network-on-Chip architectures, considering memory system design for partitioning to support MCS scheduling approaches for multiprocessor systems – without the need for the resource expensive approaches of totally separated (ie. federated) approaches.

I. Introduction

Mixed Criticality Systems (MCS) are becoming increasingly important within the embedded real-time domain. Such systems contain components with different levels of criticality on a single platform. Domain motivation for MCS comes from mainly from aerospace and automotive. The former embodies the concept of components with different criticality levels within domain safety standards (eg. IEC 61508, DO-178B). Interestingly, the aerospace industry has also been developing mixed criticality infrastructure (ie. platforms, OSs), allowing components of different criticality levels to reside on the same physical platform, for several over 20 years via IMA [1], [2], [3], [4]. The automotive industry appears to be moving towards MCS as the number and complexity of in-vehicle software components increases.

In both aerospace and automotive domains, the main motivation for considering MCS designs is to meet non-functional requirements related to, for example, space, cost and weight – and for aerospace especially, fault-tolerance[1]. By allowing components of different criticalities to share physical resources, potentially fewer resources are required. In contrast, traditional (federated) systems architectures dictate that components of differing criticalities are allocated physically separate resources.

Mixed criticality systems highlight the fundamental trade-off between partitioning or separating components of different criticalities and efficient resource usage. From a safety perspective, partitioning is required: high criticality components must be protected from the interference and potential failure of lower criticality components. To achieve adequate partitioning, federated architectures can be used. These help when there is a need to show that a single failure cannot lead to a complete system failure – a typical requirement for safety-critical (fault-tolerant) systems. However, the approach is expensive in terms of physical resources needed.

In contrast, influential recent work in mixed criticality scheduling has shown the benefits of sharing resources between criticality levels [5]. The work is motivated primarily by resource efficiency, allowing software components of different criticalities to be scheduled on the same CPU (either in a single or multiprocessor system).

This paper examines MCS from the perspective of the memory hierarchy, showing how memory systems can be designed so that partitioning can be achieved to support MCS scheduling approaches – without the need for the resource expensive approaches of totally separated (ie. federated) approaches. The approach taken is to use a memory hierarchy designed for a predictable multiprocessor Network-on-Chip (NoC) system and enable support for multiple criticality levels within that architecture. We note that this is in contrast to conventional critical system design which seeks to use unpredictable (commodity) components and constrain them to be (almost) predictable.

The remainder of this paper is structured as follows. Section II reviews memory architectures. In section III the predictable memory architecture developed within the EU T-CREST project is introduced, which is extended into an appropriate architecture for MCS systems in section IV. Conclusions are offered in section V.

II. Memory Architecture Review

Memory provides storage for state (data) and code (instructions) which must be delivered to the CPU when required, within the time and resource constraints of the system. For real-time systems, constraints are focussed upon time – increasing memory latencies will increase worst-case execution times (WCET) and reduce the overall schedulability of the system. There are a number of issues for memory architectures

[1]In an IMA system in aerospace if a lane fails (e.g. one processor board from three) functionality from the failed board can be re-assigned to one of the other boards. This results in the possibility that software components of mixed criticality levels would eventually be resident on the same board, even if the original configuration placed functions of differing criticalities on physically separate boards.

for realistic mixed criticality systems, including[2]:

1) *Performance* – the increasing gap between CPU performance and memory system performance (ie. the "memory wall" [6]).
2) *Scalability* – increasing numbers of CPUs sharing the same memory hierarchy.
3) *Physical Separation* – a memory architecture must also provide sufficient physical separation between software components using memory to meet system safety requirements.

Performance and scalability requirements suggest moving towards higher performance architectures – which is in direct conflict towards the provision of physical separation needed within safety-related systems. At the extreme, in safety-critical systems a single failure cannot lead to a total system failure. When applied to the system architecture this implies that degrees of fault-tolerance are used (eg. redundancy). When applied to the memory architecture it implies protection between memory used by different components (particularly if they are of differing criticality levels). The simple method to achieve protection is physically separated system components; or at least to physically separate components of different criticality levels. Otherwise, the method by which memory protection is achieved becomes an important part of the safety argument for the system. Unfortunately, conventional (commodity) CPUs do not provide simple memory protection, but rather memory protection based upon complex memory management units (eg. virtual memory) which can be viewed as too complex to be free of errors (and hence usable within safety-critical systems).

The conventional memory hierarchy for safety-critical systems is illustrated in Figure 1. Storage increases in volume, but decreases in performance and cost (per byte), as the hierarchy is descended [7]. The degree of potential sharing between software components also increases as the hierarchy is descended.

The remainder of this section discusses the memory hierarchy, together with memory architectures for basic systems, multicore systems and many core systems. Throughout, issues in mixed criticality are highlighted.

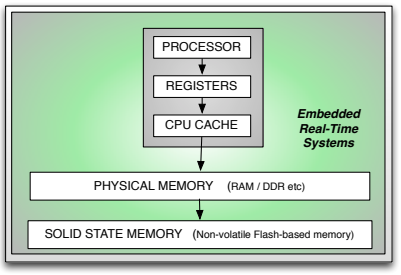


Fig. 1. Memory Hierarchy for Embedded Real-Time Systems

[2]*Application Complexity* is also important in terms of the memory hierarchy, as increasing complexity is being seen in terms of the amount of data being input, accessed, processed and stored by a system, e.g. due to the use of high bandwidth I/O devices within systems, and large scale persistent data storage (eg. for storing maps in navigation). These issues are not discussed further in this paper.

A. Basic Memory Architecture

Many systems have used simplistic memory architectures – a single CPU is connected directly to a memory structure consisting of some form of RAM, together with persistent program / data storage, the latter used mainly during bootstrap.

Within this architecture, memory latencies consist of the latency of the memory device itself; together with the bus between CPU and memory. In simplistic architectures the latter is minimal (a few cycles). Burst mode memory accesses together with DMA (potentially from I/O devices) can increase the latency of memory access as viewed by an application executing upon the CPU (eg. as their memory access may be delayed by the DMA burst instigated by an I/O device).

More complex CPUs may include caches to help performance, particularly as CPU speeds increase compared to available memory bandwidth. Worst-case execution time (WCET) calculation becomes more complex in the presence of cache [8], [9] as it is difficult to predict during analysis the state of cache at any time. Other memory variants include scratchpad memory (SPM), which make WCET calculation easier and less pessimistic [10], [11].

Clearly, basic memory architectures can be used with more complex high performance CPUs, with multi-level caches for performance, and that support memory protection via Memory Management Units (MMUs). We note the issue regarding the complexity of MMUs raised above.

B. Multicore Memory Architecture

Multicore CPUs (within the context of this paper) refer to the incorporation of several CPUs on the same chip, biased towards a shared memory architecture [7]. The development from the basic memory architecture to the multicore architecture is merely the presence of several (commonly upto 4) accessing memory via the same shared bus.

One of the largest effects of CPUs sharing the bus is upon memory latencies – a CPU may have to wait for the memory transactions of other CPUs to finish before it is able to access the bus. Similarly, a memory request reaching the physical memory may be delayed by the completion of requests for other CPUs (eg. when using DDR).

Within multicore architectures, there are many memory design choices, two important ones being whether one or more cache levels are shared between CPUs; whether to use symmetric or non-symmetric memory architecture. Shared caches are often used in high performance (eg. desktop) architectures, needing hardware support for cache coherence for obtain adequate performance [7]. In terms of real-time systems, it is not clear that any performance advantage is gained, given that execution times are extremely hard to model and bound accurately with shared coherent caches, resulting in pessimistic WCETs [8] (potentially removing any performance gained). From a safety-critical perspective, it is unclear whether the complexity of cache-coherence implementations (at the hardware level within the CPU) are appropriate.

Non-Uniform Memory Architectures (NUMA) allow different physical views of memory for each CPU. Thus, the addition of private memories to the CPUs within a symmetric architecture would make the architecture NUMA. We

can consider the non-uniform nature of the architecture at a number of levels. At a physical level, access latencies will be different depending upon whether the CPU addresses the shared memory, or its own private memory. Access to the other CPU's private memory may well be provided at the physical level (ie. within the address map of the CPU), or could require OS, and/or application support.

NUMA architectures where CPUs have private memories as well as access to shared (global) memory are an interesting option for safety-critical systems. Private memories support requirements for physical separation – assuming that components of different criticalities were placed upon different physical CPUs. If components changed criticality level however, there is a requirement to then move to a different CPU, which raises issues of migration overhead as well as schedulability of the changed system configuration.

C. Many-Core Memory Architecture

Many-core architectures (within the context of this paper) refer to approaches to scaling the number of CPUs within a chip in a manner far exceeding conventional multicore. A key difference in approach is to move away from a shared bus approach to connecting CPUs to shared memory as this approach is not scalable from the perspective of contention delays [12]. In contrast, many-core architectures adopt a form of communication mesh to connect CPUs and memories [13]. This results in many routes between a CPU and memory, so that contention on the connections between CPUs and memory can be reduced. We note that when requests actually arrive at the memory, contention will still occur at the physical level.

The manycore approach is exemplified by the Network-on-Chip (NoC) approach [14]. Typically a packet switched communications mesh of regular Manhattan topology is used, with arbiters at junctions routing traffic, and CPU tiles connected to arbiters via a local link. Essentially, the NoC appears as a distributed system, with separate CPUs connected by a communications network. In terms of memory architecture, external shared memory is attached to an arbiter at the edge of the mesh (as is any I/O); CPU tiles can also contain local memory. Alternatively, a second network can be used to connect CPUs to shared external memory to remove memory traffic from the mesh [15], [16].

This is essentially a non-symmetric approach. Each CPU has a different view of the memory depending upon how many hops are between it and the external memory, and other CPUs (if a CPU is allowed to indirectly access other CPUs local memory [17], [18]).

In terms of achieving adequate performance from the memory architecture, the mapping of application components to CPUs and memories within the many-core becomes crucial. Code needs to be close to required data; application threads that communicate should be placed on CPUs that are close. Mapping approaches are an active research area [19], noting that optimal solutions are NP-complete.

D. Summary: Memory Requirements for MCS

Mixed criticality systems highlight the fundamental trade-off between partitioning or separating components of different criticalities and efficient resource usage. Recent MCS research (based upon [5]) considers scheduling CPUs between processes of differing criticality levels, largely ignoring other resources. For memory, available separation is defined by the physical architecture. To facilitate recent MCS scheduling approaches, memory architectures require software processes of differing criticality levels to share physical memory, with two areas of sharing to consider:

- *Shared route / connection between CPU and memory*: Shared connections imply the potential for interference between competing memory transactions. This can lead to race conditions between memory transactions, and difficulties in bounding transaction latency.
- *Shared physical memory*: Where memory (and memory controller) are physically shared between many CPUs, the memory is effectively multiplexed between the CPUs – hence memory requests from one CPU may be delayed by those of another.

One of the characteristics of MCS scheduling approaches is that Worst-Case Execution Times (WCET) are dependent on criticality level – more conservative (higher) values are used if the process is assigned a higher criticality level. In terms of memory access latencies (a part of WCET), there is a fundamental requirement for predictability. However, different assumptions regarding competing memory transactions can be made to allow less pessimistic memory latency times to be derived (see sections III and IV).

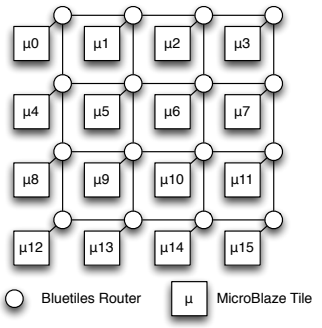


Fig. 2. Blueshell Network-on-Chip with 16 CPU Tiles

III. Predicitable Memory Architecture

The EU T-CREST project[3] is developing NoC architectures suitable for safety-critical systems requiring the highest levels of predictability. In this section we discuss a shared memory tree architecture (Bluetree) that has been developed within T-CREST. Subsequently we show how this architecture supports mixed criticality systems.

We note that the memory hierarchy that is discussed in this section was designed to support timing predictability. This paper contends that this is a more appropriate starting point for a MCS memory architecture than adopting commodity hardware approaches and attempting to restrict their behaviour to being (nearly) predictable.

[3]T-CREST – Time Predictable Multicore Architecture for Embedded Systems: http://www.t-crest.org/

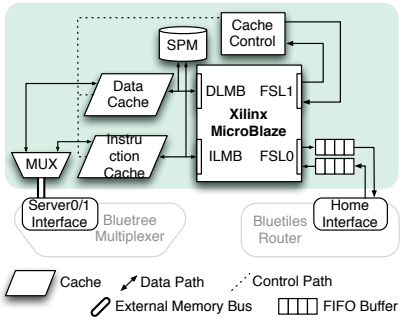


Fig. 3. Internal Structure of Microblaze CPU Tile

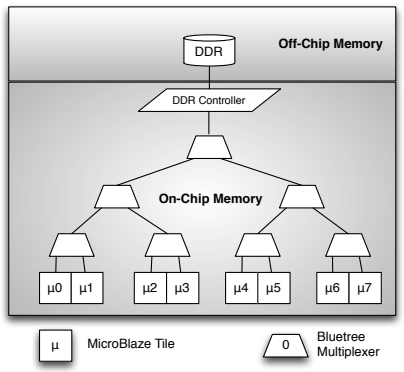


Fig. 4. Bluetree Shared Memory Tree for 8 CPU Tiles
Each CPU tile in the Bluetiles NoC (see figure 2) also connects directly (via cache) to the Bluetree shared memory (binary) tree to access external DDR memory, shown above. There is no interference between CPU to CPU messages across the NoC and CPU to memory transactions across the tree.

A. Bluetree Shared Memory Tree Architecture

Bluetree is a shared memory tree architecture developed for NoC architectures [15], [16], [20] CPUs are arranged in a mesh (Figure 2) to allow CPU to CPU communication. Each CPU tile within the architecture has two main interfaces (Figure 3), one to access the mesh (*Home Interface*) and one to access shared memory (*Server Interface*). The separation of inter-CPU communications from shared memory access (eg. for cache misses) ensures that memory accesses do not interfere with CPU to CPU communications – thus aiding predictability. The shared memory tree architecture is illustrated in Figure 4:[4]

- *Routers*: Routers are 32-bit bi-directional with X-Y routing used (destination is contained in the first word). We note that the choice of routing policy does not impact upon the research presented in this paper, since this research focuses on the communication over Bluetree; Bluetiles is only used for simple synchronisation.
- *Shared Memory Tree (Bluetree)*: 2-to-1 multiplexors form a tree connecting CPUs to memory – CPUs are the leaves of the tree, memory being at the root. High-bandwidth memory requests do not impact the performance of other CPUs – there is no interference between CPU to CPU messages across the NoC and CPU to memory transactions across the tree. Each multiplexor port allows 128 bits of data (corresponding to the cache line size).
- *CPU Tiles* [15], [16]: CPU tiles are built using the Microblaze CPU. CPU configuration is 8kB split data and instruction caches, and a 8kB shared scratchpad used for fast local storage. The CPU accesses the cache via Microblaze LMB interfaces; cache misses being issued to external memory via Bluetree. The CPU tile contains custom cache control is configured to allow selective invalidation of cache lines and to record prefetch related data on a per cache line basis (the cache control unit also serves as the Microblaze's interrupt controller and provides a clock-cycle counter facility). Cache control is accessed via Microblaze Fast Simplex Links (FSL), utilising single-cycle FIFOs. Further details of the cache design are given in [15], [16].

[4]Bluetree has also been implemented with a pre-fetch unit between off-chip and on-chip memory[15], [16], [20].

B. Off-chip Memory

The Bluetree shared memory tree is connected to off-chip memory as shown in Figure 4. The key component is the memory controller which interfaces between requests originating from the CPU and passing over the shared memory tree, and the external (off-chip) SDRAM (ie. DDR). The controller is based upon the architecture in [21], but has been developed with additional configuration infrastructure to allow, effectively, several distinct channels. The motivation for this is to allow, for example, separate memory access bandwidths to be specified for each channel, thus providing a degree of separation in the memory system – essentially each channel forms a separate queue of memory requests which are then multiplexed (according to remaining bandwidth) onto the single SDRAM.

IV. Predictable Memory Architecture for MCS

This section discusses the predictable memory architecture described in section III in the context of MCS. Initially an appropriate arbitration scheme is outlined for Bluetree. This is then taken, together with definition of the worst-case latency across the memory tree, to provide a worst-case timing of the memory architecture. Finally, this is assessed to see whether it supports MCS and MCS scheduling approaches.

A. Arbitration within Bluetree

The shared memory tree requires arbitration at each of the multiplexors. A simple approach would be to adopt a first-come-first-served approach. It could be argued that this could lead to starvation, as one input (CPU) to a multiplexor could monopolise – although this would require memory requests to be generated on *every* clock cycle, which is unlikely. Another approach would be to always favour one input over another, but this could lead to, eg. priority inversion (in a priority scheduled system). Another approach could be to ensure that turns were taken at each multiplexor to eliminate starvation, but would partially suffer from effects like priority inversion.

The approach taken within Bluetree is to allow run-time programming of the arbiters. Essentially, each memory request from a CPU is accompanied by some measure of importance (eg. priority) so that the most important request always wins

arbitration at a multiplexor. We note that this easily maps onto criticalities – at each multiplexor, if two requests arrive at the same time (ie. same clock cycle) the request with the highest criticality would be forwarded. In the event of a tie (i.e. equal criticality) a secondary mechanism can be used, eg. turns.

The Bluetree arbitration approach provides separation between streams of memory requests originating from software components of different memory criticalities. From a safety-critical perspective it would be relatively straightforward to show that there can be no interference between input channels to the multiplexor (as they have separate latches) – and therefore that separation is maintained (multiplexor logic is simple, cf. MMUs).

B. Worst-Case Latency

The worst-case time for a memory request issued Bluetree is the sum of three components:

1) *Latency to cross Bluetree from CPU to memory controller*: The basic latency to cross a shared memory tree multiplexor is two cycles (Bluetree uses a buffer for both input and output) [16], hence total latency is 2 + (2 * the memory tree depth) – noting that there is an extra latch stage used to link shared memory tree to the memory controller.
2) *Latency to cross the memory controller and access the SDRAM*: This varies according to the exact configuration and build of the controller, but typical figures are around 25 cycles for a read (and 1 cycle for subsequent reads in a burst), and 1 cycle for a write. These figures refer the time needed *before* return can be made to the CPU.
3) *Latency to cross Bluetree from memory controller to CPU*: This is 1 + the memory tree depth (as no latching occurs within the tree on the return path). For a burst, the first word returned would suffer the full return path latency, successive bytes are delivered in successive clock cycles.

Note that a burst read, which can be encapsulated within a single 128 bit request to the memory controller (and is therefore non-preemptable within the tree) is actually broken into a series of successive memory requests to the SDRAM – hence there is potential for pre-emption within the memory controller if required.

C. Worst-Case Timing of Bluetree for MCS

The worst-case timing across the shared memory tree requires the worst-case latency and MCS arbitration approaches (above) to be combined.

Consider a CPU executing with a software component of criticality level X. All memory requests from that CPU (whilst that software component is executing) have the criticality of the issuing software component. At each multiplexor a memory request can be delayed by at most one request of equal criticality; and by the maximum number of successive requests of higher criticality memory requests.

When a memory request exits the multiplexor connected to the memory controller and At the memory controller, there

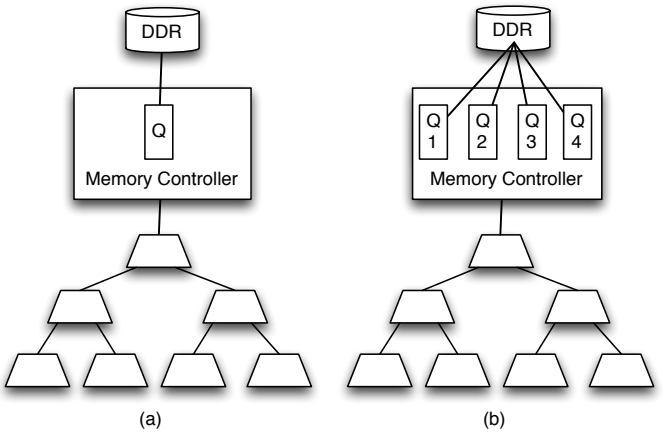


Fig. 5. Memory tree with (a) Single Memory Queue and (b) Memory Queue per Criticality Level

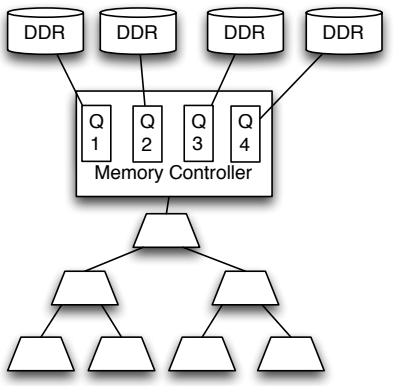


Fig. 6. Memory tree with Memory Queue and Memory per Criticality Level

are three possibilities when considering memory request m of criticality X:

1) *Single Memory Queue* (Figure 5(a)) – account must be taken for all other memory requests that arrive ahead of m that are therefore ahead within the memory controller queue.
2) *Multiple Memory Queue* (Figure 5(b)) – where there are separate queues for each criticality level. Here account must be taken for all other memory requests of the same criticality that arrive ahead of m that are therefore ahead within the memory controller queue; and of all other memory requests in higher criticality queues.
3) *Multiple Memory Queue and Memory* (Figure 6) – where there are separate queues and physical memories for each criticality level. Here account must be taken for all other memory requests of the same criticality that arrive ahead of m that are therefore ahead within the memory controller queue.

One issue remains, that of the effect of bursts. If they are not pre-emptable (see discussion above) then allowance has to be made for the total burst length when calculating maximum delay at the memory controller. If this becomes dominant, it may be appropriate to adopt more bandwidth oriented memory

controller approaches (eg. see [21]).

D. Mixed Criticality Timing Analysis Principles and Memory

The essential principle of MCS is that a software tasks' WCET is dependent upon its criticality level. At a high level of criticality code will have a higher WCET than if it is assigned a lower criticality level. This reflects the degree of pessimism that is required at different criticality levels – if exceeding the WCET is deemed a failure, then at the highest level of criticality (in aerospace systems) this failure has a probability of no more than 1 in 10^9 occurrences. At lower levels of criticality, the same code can be assigned lower (less pessimistic) WCET bounds. Although the chances of exceeding the WCET has increased, the effect on the system is not as great – eg. system integrity is not compromised by multiple failures of components at the lowest criticality level.

The essential principles outlined above applies equally well to the analysis for the memory architecture defined within this section. As the criticality level of a software component increases, a more pessimistic view of the potential interference of other tasks upon memory requests must be included in the analysis. If assigned the highest criticality level, each memory access of a task would assume worst-case interference upon its memory requests as given above – potentially large. At lower levels of criticality, la ess pessimistic view can be taken: that accesses will suffer less interference. This is achieved by considering the number of cycles assumed between successive memory transactions from other CPUs – for high levels of criticality the minimal number of cycles can be assumed; for low levels of criticality longer intervals can be assumed (more realistic in the average case).

V. Conclusions

This paper has considered the role of the memory hierarchy within many core architectures (specifically Network-on-Chip) proposing an appropriate memory hierarchy for MCS based upon a predictable shared memory tree memory hierarchy. The paper has shown that sufficient partitioning and separation can be provided by the architecture to ensure mixed criticality components can share resources without compromising system safety at the highest criticality levels. Thus the approach supports the MCS scheduling work within the real-time community which allows components of mixed criticality to share resources.

The architecture includes an arbitration approach within the memory tree that directly supports criticality levels. The additional benefit of this is that if stricter memory separation was needed to support safety-critical requirements, separate memories can be included, one per criticality level. This is less than normally required for federated architectures which would dictate one memory per component (not criticality level).

Acknowledgements

This work is supported in part by EU FP7 project T-CREST (288008).

References

[1] R. A. Edwards, "ASAAC Phase I Harmonized Concept Summary," in *Proceedings ERA Avionics Conference and Exhibition*, UK, 1994.

[2] *ARINC 651: Design Guidance for Integrated Modular Avionics*. Airlines Electronic Engineering Committee (AEEC), 9th November 1991.

[3] *ARINC 653: Avionics Application Software Standard Interface*. Airlines Electronic Engineering Committee (AEEC), June 17th, 1996.

[4] N. Audsley and A. Wellings, "Analysing apex applications," in *Real-Time Systems Symposium, 1996., 17th IEEE*. IEEE, 1996, pp. 39–44.

[5] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance." in *RTSS*. IEEE Computer Society, 2007, pp. 239–243.

[6] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[7] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier Science, 2011.

[8] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem – overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[9] J. Whitham, R. I. Davis, N. C. Audsley, S. Altmeyer, and C. Maiza, "Investigation of scratchpad memory for preemptive multitasking," in *Proceedings of the 2012 IEEE 33rd Real-Time Systems Symposium*, ser. RTSS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 3–13.

[10] J. Whitham and N. Audsley, "Investigating average versus worst-case timing behavior of data caches and data scratchpads," in *Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems*, ser. ECRTS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 165–174.

[11] P. Marwedel, *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*, ser. Embedded systems. Springer, 2010.

[12] H. G. Lee, N. Chang, U. Y. Ogras, and R. Marculescu, "On-chip communication architecture exploration," *ACM Trans on Design Automation of Electronic Systems*, vol. 12, no. 3, 2007.

[13] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pp. 684–689, 2001.

[14] G. De Micheli and L. Benini, *Networks on Chips: Technology and Tools*, ser. Systems on Silicon. Elsevier Science, 2006.

[15] G. Plumbridge, J. Whitham, and N. C. Audsley, "Blueshell : A Platform for Rapid Prototyping of Multiprocessor NoCs and Accelerators," in *Proceedings HEART Workshop*, 2013.

[16] J. Garside and N. C. Audsley, "Prefetching Across a Shared Memory Tree within a Network-on-Chip Architecture," in *Proceedings International Symposium on System-on-Chip*, 2013.

[17] I. Loi and L. Benini, "An efficient distributed memory interface for many-core platform with 3d stacked dram," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2010, pp. 99–104.

[18] B. C. Lam, A. D. George, and H. Lam, "Tshmem: Shared-memory parallel computing on tilera many-core processors," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 325–334.

[19] P. K. Sahu and S. Chattopadhyay, "A survey on application mapping strategies for network-on-chip design," *Journal of Systems Architecture*, vol. 59, no. 1, pp. 60 – 76, 2013.

[20] J. Garside and N. C. Audsley, "Investigating Shared Memory Tree Prefetching within NoC Architectures," in *Proceedings Memory Architecture and Organization Workshop MEAOW*, 2013.

[21] B. Akesson and K. Goossens, "Architectures and modeling of predictable memory controllers for improved system integration." in *DATE*. IEEE, 2011, pp. 851–856.

Maximizing the execution rate of low-criticality tasks in mixed criticality systems

Mathieu Jan and Lilia Zaourar
CEA, LIST
Embedded Real Time Systems Laboratory
F-91191 Gif-sur-Yvette, France
Email: Firstname.Lastname@cea.fr

Maurice Pitel
Schneider Electric Industries
37, quai Paul Louis Merlin
F-38050 Grenoble, France
Email: Maurice.Pitel@schneider-electric.com

***Abstract*—Industrial fields must build at the most competitive price real-time systems made of an increasing number of functionalities. This can be achieved by hosting high-criticality tasks as well as consumer real-time low-criticality tasks on a same chip. The design of such Mixed-Criticality (MC) systems requires the use of an appropriate task model and a specific scheduling strategy. In this work, inspired by the existing elastic task model, we introduce stretching factors as a way for the low-criticality tasks to reduce their utilization, as well as a level of importance in order to define an order for applying these stretching factors. At run-time, the slack time generated by both the over-provisioned high-criticality and the low-criticality tasks is used to maximize the execution rate of the low-criticality tasks. We also show how to integrate this approach in the Time-Triggered paradigm (TT), in particular its impact on the data visibility principle between the low-criticality tasks when they have been stretched.**

I. Introduction

Traditionally, industrial systems use a dedicated (possibly multiprocessor) chip for executing a set of real-time tasks with a same level of criticality. When such tasks are safety-critical, high margins are taken on their Worst-Case Execution Time (WCET). This leads to the specification of high allocated budgets of time for such high-criticality tasks. Besides, the probability that the WCET of a set of high-criticality tasks occur simultaneously is very low. However, the schedulability demonstration of safety-critical systems must be performed in the worst-case situation, due to certification constraints. This therefore leads to a huge over-sizing of the CPU resources that are needed compared to what is really used, in average, while the system is running. This practice becomes incompatible with the current trend of tighter economical constraints of various industrial domains, such as the automotive or energy distribution fields. Therefore, there is a need to use these generally unused processing capabilities for executing the low-criticality tasks.

This type of system where both the low and high-criticality tasks are allocated on a single chip are called Mixed Criticality (MC) systems. Note that in general, two criticality levels are generally considered in existing work on this topic, as well as in the remainder of this paper. To fulfil certification requirements, mainly from the avionic domain, and enable an efficient scheduling of the high and low-criticality tasks, task models and scheduling algorithms addressing MC systems have recently been proposed ([18], [10], [3]). The goal is to increase the schedulability of the low-criticality tasks, while still guaranteeing in the worst-case scenario the schedulability of the high-criticality tasks. Classical scheduling algorithms can indeed lead to well-known priority inversion problems, as they are unaware of the criticality parameter of the tasks. Most existing work defines two modes for a MC system and each task must specify its criticality level. The MC system starts in the low-criticality mode where all tasks are executed. However, when a deadline is missed the MC system switches to the high-criticality mode, where only the high-criticality tasks are executed. The low-criticality tasks are simply dropped. This degrades the level of service provided by MC systems. Besides, processing capabilities are wasted.

Maximizing the level of service provided by a set of real-time tasks for controlling a physical system has been the subject of numerous work in the domain of real-time control ([7], [1]). When a perturbation/event suddenly affects the controlled system, the higher the sampling period is, the better the reactivity of the system is. However, the higher the processing load is. This requirement has therefore led to the proposal of a more flexible task model, called the elastic task model [5], where the periodicities of tasks can take a range of values. This task model, combined with an appropriate scheduling algorithm, avoids the need of dropping tasks when a deadline is missed. The periodicity of some tasks must simply be increased appropriately. Recently, the elastic task model has been studied in the context of MC systems for the low-criticality tasks [16]. The goal is to deal with the service abrupt problem of dropping the low-criticality tasks when the MC system switches to the high-criticality mode.

This work also proposes a solution to this problem. In this work, we also adapt the elastic task model for solving the following problem: *how to maximize the utilization of the processing capabilities of an architecture, in which high- and low-criticality tasks are schedulable taken separately, while the sum is not?* One goal is therefore to avoid dropping the low-criticality tasks. Off-line, we use a linear programming approach to compute the different *stretching factors* that must be applied on the periodicity of the low-criticality tasks, so that the schedulability of the high-criticality tasks are guaranteed. On-line, we bet on the availability of slack time generated by high and low-criticality tasks. No stretching factors are therefore applied in order to execute the low-criticality tasks at their fastest rates. However, when a deadline is going to be missed by a low-criticality task, its deadline is stretched up to point that prevents the deadline miss.

Another contribution of this work is to show *how our proposed approach to deal with the low-criticality tasks can be used within the TT paradigm* [13], used to build safety-critical real-time systems. Using this paradigm, the triggering of activities, that correspond to task releases and deadlines, are specified by the application designer. At these dates, data exchanged between the tasks are made visible. Stretching a task introduces a modification in these statically defined triggering points as well as in the visibility date of the produced data. Therefore, when stretching a low-criticality task, the deadline of some other low-criticality tasks must also be postponed in order to keep the system temporally consistent.

The remainder of this paper is as follows. Section II describes the related work in the scheduling algorithms for MC systems as well as their practical implementation. Then Section III formulates the problem, while Section IV presents the proposed task model and the on-line decision algorithm used within the TT paradigm to stretch the low-criticality tasks. Finally, Section V evaluates the proposed solution and Section VI concludes.

II. Related work

In [18], Vestal introduces the most used task model for specifying MC real-time systems, therefore sometimes called the MC task model. The classical periodic task model is extended with two WCET values, named $C_i(LO)$ and $C_i(HI)$, and a criticality level, which can be either low or high. As stated previously, we consider that a MC system has only two modes of criticality: high and low. $C_i(LO)$ is the maximum allowed execution time for the task in the low-criticality mode, while $C_i(HI)$ is the maximum allowed execution time for the task in the high-criticality mode ($C_i(LO) < C_i(HI)$). The rationale for specifying two WCETs is that the higher the criticality level, the more conservative the verification process and hence the greater will be the value of C_i.

In the context of digital control systems, early work has focused on an off-line analysis to compute the tasks frequencies that minimize the cost of the control and tracking error. This has lead to the definition of the so-called elastic task model [5] to increase the flexibility of the periodic task model. In addition to its execution time, each task is characterized by: a nominal period T_{i_0}, a minimum period $T_{i_{min}}$, a maximum period $T_{i_{max}}$ and an elastic coefficient $e_i \geq 0$. The elastic coefficient specifies the flexibility of the task to vary its utilization within the range of possible periods. [17] also presents a task model which allows to jointly optimize the used computing resources and the control performance of a computer-based instrumentation and control system. Each control task is able to trigger itself: the timing constraints are dynamically adjusted based on the whether the controlled system is stable or subject to perturbations. Finally, [14] proposes to integrate in the task model a parameter to specify a minimal distance between two consecutive skips of instances of a task, that is between two deadline misses.

In this area, our closest related work is [16], where the elastic task model is applied in the context of MC systems for specifying the behavior of the low-criticality tasks. However, this task model does not allow to specify an order between the low-criticality tasks, as how a set of elastic tasks is compressed depends on their utilization.

Appropriate scheduling algorithms must then be defined to support task models used within MC systems, in particular the criticality-level parameter. The goal is to ensure the schedulability in the worst-case scenario of the high-criticality tasks, while improving the schedulability of the low-criticality tasks. This is possible thanks to the introduction of $C_i(LO)$ for each high-criticality task. Several approaches have been followed: using either a fixed priority algorithm [18], a zero-slack algorithm on top a fixed priority scheduler [10], the assignment of virtual and smaller deadlines for the high-criticality tasks (EDF-VD for EDF-Virtual Deadlines) [3] or the definition or early release points for accelerating the execution rate of the low-criticality tasks [16] (ER-EDF for Early Release EDF). This last decision algorithm for releasing earlier or not the low-criticality tasks is the closest related work to ours. However, it takes the opposite approach to execute the tasks at their fastest execution rate: it computes a new (early) release point when the task finishes, while we extend the deadline of the task when it is going to miss its deadline.

On the implementation side of MC systems, [11] presents a first implementation of a MC hierarchical scheduling framework on a multi-core system, that addresses the criticality levels of the avionic domain. It is based on LITMUS [6], an extension to Linux that was developed to study in practice real-time multiprocessor schedulers. The focus is put on optimizing the implementation of the proposed hierarchical schedulers for MC systems, by using fine-grained locking mechanisms to reduce scheduling overheads. In our work, we are also interested in the integration of MC scheduling into real-time operating systems, but more specifically in the TT paradigm. Finally, [2] presents an implementation in ADA of mode changes in MC systems, from low-criticality to high-criticality as well as the opposite. The authors consider the problem of when returning to the original ordering, as doing it prematurely can cause a high-criticality task to miss its deadline. However, none of these works have considered the impact on data exchanges between the tasks when switching to another mode of execution.

III. Problem formulation

A. Motivation

Within an embedded system, the tasks can be either critical, less (or even non-) critical. Let us for instance, take a protection relay used within medium voltage electrical networks. The safety-function of the software part of protection relays is to first detect any faults within the supervised power network, and then ask the tripping of the circuit breakers in order to isolate the faulty portion of the network. More details on the required set of high-criticality tasks needed to achieve this functionality can be found in [12]. As any safety-related system, protection relays have to comply with a Safety Integrated Level (SIL), as defined by the *IEC 61508* standard. This standard requires that the schedulability demonstration of the high-criticality tasks must be performed. To take into account worst-case situations, high margins are taken on the WCET of these high-criticality tasks. This leads to an over sizing of the required CPU power, compared to what is required in average while the system is running.

On the other hand, there is a need to embed additional less (or non) safety functionalities, such as displaying information,

optimizing the distribution of energy to the current need, etc., in order to distinguish the product from competitors. This leads to the requirement of executing applications with different levels of criticality on the same system. Besides, assuming the WCET for the high-criticality tasks and executing the low-criticality tasks in the remaining CPU power is no longer a viable approach: too much processing power is wasted. Such an approach is no longer compatible with current economical constraints that push for minimizing the CPU power. Such products must take advantage of the slack time generated by the tasks, when they are not using their WCET, in order to execute the low-criticality tasks. Therefore, the problem we address is to *enable the design of MC systems, where taken separately the high and the low criticality tasks are schedulable but the union is not.*

B. Approach

Our goal in this work is to allow the low-criticality tasks to use the slack time and, when a deadline is going to be missed by a low-criticality task, to relax its temporal constraints. To this end, we consider the deadline of the low-criticality task as a flexible parameter that can be extended. This flexibility is handled through a so-called *stretching period factor* (or simply stretching) that we introduce in the classical implicit-deadline periodic task model. A stretching factor is a value by which the periodicity of a low-criticality task can be multiplied. Stretching factors should be specified off-line by the application designer so that a bound is defined. They can be a set of values or a range of values and a same value of stretching factor can be given to several tasks. Besides, we assign to each low-criticality task an importance level. This importance level denotes an order for choosing which low-criticality tasks should be stretched first. Our task model is simpler than the elastic task model [5], as it does not contain a minimum period and an elastic coefficient, linked to the utilization of the task.

Off-line, the stretching factors are used in the schedulability analysis of a task set, made of both the high and the low-criticality tasks. This guarantees that the stretching factors used on-line cannot lead to a situation where a deadline is missed. Besides our task model introduces a way to specify an order between the low-criticality tasks for applying the stretching factors. This gives more control to the application designer to specify a set of possible temporal behavior for the low-criticality tasks. Furthermore, the formal demonstration of the fulfilment of end-to-end constraints for these tasks is therefore still possible. This was main requirement that lead us to the definition of our task model.

IV. Using stretching factors

A. Task model and notations

Let $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ be a set of n independent, synchronous, preemptible and periodic tasks. The set Γ is partitioned into two disjoint subsets: Γ_{ct} the set of the high-criticality tasks, made of n_{high} tasks, and Γ_{nct} the set of the low-criticality tasks, made of n_{low} tasks. We therefore have $n = n_{low} + n_{high}$.

Γ is handled using the classical implicit-deadline periodic task model. Each task $\tau_i \in \Gamma$ has the following temporal parameters $\tau_i = (P_i, C_i, D_i)$. P_i is the period of the task, C_i is its WCET, D_i its deadline and we have $P_i = D_i$. Furthermore, each non-critical task $\tau_i \in \Gamma_{nct}$ has two additional parameters V_i and $S_{i,max}$, where V_i is the importance level of the task and $S_{i,max}$ is the maximum stretching factor that can be applied to P_i. We denote by S_i an actual value for the stretching factor of task τ_i and we have: $1 \leq S_i \leq S_{i,max}$. We denote by $P_{i,max}$ the period when $S_{i,max}$ is applied and corresponds to the maximum period the task can have. The higher the value of V_i, the higher is the importance level of the low-criticality task. But the later its stretching factor will be increased first when a deadline is going to be missed.

The processor utilization of τ_i, a low-criticality task, is: $u_i = \frac{C_i}{S_i \times P_i}$. The lower bound of u_i (noted $u_{i_{min}}$) is reached when the maximum stretch factor ($S_{i,max}$) is applied to τ_i, while its upper bound (noted $u_{i_{max}}$) is reached when its nominal period is used. The processor utilization of a high-criticality task is $u_i = \frac{C_i}{P_i}$. We note respectively U_{low} and U_{high} the total utilization of the low and the high-criticality tasks: $U_{low} = \sum_{\tau_i \in \Gamma_{nct}} \frac{C_i}{S_i \times P_i}$ and $U_{high} = \sum_{\tau_i \in \Gamma_{ct}} \frac{C_i}{P_i}$. The total utilization of the system is noted U and is equals to $U_{low} + U_{high}$. Finally, let m be the number of processors.

B. Off-line CPU maximization

Off-line, we compute for each low-criticality task the minimum stretching factor ($S_{i,min}$) that must be used so that no deadline is missed, assuming that each task uses it WCET. Therefore, we have $S_{i,min} \leq S_{i,max}$. $S_{i,min}$ is a feedback to the application designer on the worst-case temporal behavior the low-criticality task may use on-line. As we focus on the low-criticality tasks only, we can therefore remove from U the utilization generated by the high-criticality tasks. We denote by U_r this remaining CPU capacity, which is equal to $m - U_{high}$. A first constraint therefore expresses the fact that U_r upper bounds the utilization that can generate the low-criticality tasks:

$$U_{low} \leq U_r \Leftrightarrow \sum_{i \in \Gamma_{nct}} \frac{C_i}{S_i \times P_i} \leq U_r \tag{1}$$

Then, a second constraint therefore expresses the fact that the utilization value of a low-criticality task is bounded, as seen in the previous section.

$$u_{i_{min}} \leq \frac{C_i}{S_i \times P_i} \leq u_{i_{max}} \tag{2}$$

Our objective is to maximize the utilization of the resources, while stretching the less important low-criticality tasks first, that is:

$$MaxZ = \sum_{i \in \Gamma_{nct}} V_i \times u_i = \sum_{i \in \Gamma_{nct}} V_i \times \frac{C_i}{S_i \times P_i} \tag{3}$$

By applying the following change of variable: $x_i = \frac{1}{S_i}, \forall \tau_i \in \Gamma_{nct}$, we obtain the following linear program:

$$\text{(LP-1)} \quad \begin{cases} \text{Max :} & \sum_{i \in \Gamma_{nct}} V_i \times x_i \times \frac{C_i}{P_i} \\ \text{s.t :} & \sum_{i \in \Gamma_{nct}} x_i \times \frac{C_i}{P_i} \leq U_r \\ & u_{i_{min}} \leq x_i \times \frac{C_i}{P_i} \leq u_{i_{max}}, \forall \tau_i \in \Gamma_{nct} \end{cases} \tag{4}$$

Algorithm 1 Decision algorithm for setting the stretching factors of low-criticality tasks.

Require: $\tau_i \in \Gamma_{nct_k}$ and the current time t
1: $S_i \leftarrow ComputeStretching(\tau_i, t, D_i, S_i)$;
2: **if** $S_i \geq S_{i,min}$ **then** Stop τ_i and log the error; **end if**
3: $D_i \leftarrow S_i * P_i$;
4: UpdateReady(τ_i);
5: Call the scheduler;

The total number of decision variables of the linear program LP-1 is equal to n_{low}, the number of low-criticality tasks. The total number of constraints is equal to $2n_{low} + 1$. Indeed, there are two constraints for each decision variable in order to express the upper and lower bounds, plus the constraint of remaining CPU capacity. LP-1 can thus be solved in polynomial time.

C. On-line decision algorithm

We now focus on the on-line decision algorithm that sets the stretching factors. We assume that the high-criticality tasks have a higher priority over the low-criticality tasks. Clearly, this increases the number of times the low-criticality tasks are preempted. As in [19], a wrapper-task mechanism for the slack time can be used to avoid such situations.

Our decision algorithm is called when the system detects that one of the low-criticality task is going to miss its deadline. This is the beginning of an overloaded situation for the low-criticality tasks, during which other low-criticality tasks may reach a point where they are also going to miss or have already missed their deadlines. This last case can occur when the system reschedules the low-criticality tasks after some high-criticality tasks have been executed, while in an overloaded situation. Therefore, our decision algorithm is also called before scheduling the low-criticality tasks.

When our decision algorithm is called, we assume that the most important low-criticality task is being executed. To achieve this, the low-criticality tasks could be scheduled using a hierarchical approach: first using the importance level and then using EDF within a given importance level. Another solution would be to use EDF-VD [3] to favour important low-criticality tasks that have far away deadlines over less important low-criticality tasks that have closer deadlines. More generally, our decision algorithm can be combined with any scheduling algorithm if the aforementioned hypotheses are fulfilled. Finally, when a low-criticality task finishes, if it has stretched, then its stretching factor is reset to 1. Note that other strategies could be defined, such as a fixed-timeout strategy before resetting the stretching factor in order to avoid additional calls to our decision algorithm and the associated system calls overhead, if the overloaded situation goes on. The definition and the evaluation of such strategies is currently left as future work.

Algorithm 1 presents the major steps of our decision algorithm when the low-criticality task τ_i is going to miss its deadline. The function *ComputeStretching*, used to compute the stretching factor of τ_i (line 1), can be implemented using different strategies. An optimistic strategy would be to choose a first reasonable value that leads to set a deadline in the future

Algorithm 2 Additionnal steps in the decision algorithm when integrated in the TT paradigm, compared to algorithm 1.

Require: Γ_{nct_k} with $\tau_i \in \Gamma_{nct_k}$
1: **for all** $\tau_j \in \Gamma_k \neq \tau_i$ **do**
2: **if** τ_j is ready **then** RemoveFromReady(τ_j);
3: **else** RemoveFromSleeping(τ_j); **end if**
4: **if** $S_i \geq S_{j,min}$ **then** Stop Γ_{nct_k}, log the error; **end if**
5: $D_j \leftarrow P_j + (D_j - P_i)$;
6: **if** τ_j is finished **then** SetFlag(Stretched); **end if**
7: InsertReady(τ_j);
8: **end for**

($S_i * P_i > t$). By reasonable, we mean that the task has a good chance, according to the distribution of its execution time, to finish its execution before its new deadline (set at line 5). Other strategies are possible that might reduce the number of times the stretched deadline is reached.

D. Using stretching factors within the TT paradigm

Applying stretching factors to the low-criticality tasks within the TT paradigm raises an issue. The hypothesis of independent tasks that can be made at a system level, does not hold any more at the application level. In the TT paradigm, to each produced data is indeed associated a timestamp: the deadline of the producer task. Then, a task may only use data whose timestamps are equals or inferior to its release date, leading to a deterministic execution with demonstrable end-to-end temporal constraints. Therefore, the visibility date of data produced by a low-criticality task changes when the task is stretched. However, the low-criticality tasks have defined triggering points (release date and deadlines) assuming the non-stretched temporal behavior. This leads to an inconsistency between the expected temporal behavior of the tasks, if the stretching factor of a single task is modified. In addition, this inconsistency has an impact on the various worst-case end-to-end temporal behaviors that can fulfil the low-criticality tasks.

To solve this issue, we assume that the application designer can gather in groups the low-criticality tasks that must be kept temporally consistent between them. Therefore, Γ_{nct} is made of a set of groups, noted Γ_{nct_k}. A low-criticality task τ_i can only be inside a single group and the multiplicity of a group can range between 1 to n_{low}. Off-line, our task model must be adapted so that the importance level and the stretching-factor parameter is applied to the group level. Therefore, the only modification to the linear program LP-1 is to consider the utilization of each group Γ_{nct_k} and not the utilization of each task. The utilization of a group Γ_{nct_k} is defined as $\frac{1}{S_k} \times \sum_{\tau_i \in \Gamma_{nct_k}} \frac{C_i}{P_i}$.

Algorithm 2 then presents the additional steps that must be done before calling the scheduler (line 5 in algorithm 1) to stretch the other tasks within a group Γ_{nct_k}, in which task τ_i is going to miss its deadline. Two cases must be considered when recomputing the deadline of a task τ_j: either it has been released but is not finished (line 2) or it is already finished (line 3). In this last case, the visibility date of already produced data must be changed and the part of the task that sets the visibility date must therefore be re-executed. This is signaled by setting the flag *Stretched* (line 6) and setting back the task in the set

of tasks waiting to be executed (line 7) using the *InsertReady* function. This function is also called in the other case to sort the tasks according to the scheduling policy, as their deadlines have changed. Computing the new deadlines simply consists of translating the offsets triggering points of Γ_{nct_k} with the initial deadline to the stretched deadline of τ_i (line 5). The *Stretched* flag is also used by the tasks, from Γ_{nct_l} with $l \neq k$, to avoid updating the values of data they use while the visibility dates of the stretched tasks are recomputed.

Note that the low-criticality tasks cannot use at a given timestamps different values of the same data (i.e., a data inconsistency). Our algorithm is indeed called at the visibility date of data, produced by the initial stretched task. Therefore by definition, these data are not yet visible by the other tasks. While on a uniprocessor system implementing this is straightforward, on a multiprocessor this can be achieved using a spin-lock for the management of scheduling structures. Also note that the stretching factors of the other groups Γ_{nct_l} (with $l \neq k$) do not need to be modified. These groups are indeed by definition less important, so their stretching factors will be computed when the hierarchical scheduler will schedule them.

V. Preliminary evaluations

The goal of our simulation experiments is to validate our proposed task model in its ability to specify an order for choosing which tasks should stretch. We therefore focus on the off-line part of our proposal.

A. Simulation environment

We generate random task sets for both the low-criticality and the high-criticality tasks. The utilization of each task is computed randomly between 0.01 and 0.99 with a uniform distribution using the UUniFast-Discard algorithm from Davis and Burns [9], an extended version of the UUniFast algorithm from Bini and Buttazzo [4] targeting multiprocessor systems. We bound the range of possible periods between $10ms$ and $100ms$ and use a uniform distribution when assigning P_i. The task sets with a hyper-period larger than $10s$ are rejected to remain in a realistic bound of typical industrial systems. Each task is assigned a boolean value that determines whether it is a high-criticality or a low-criticality task. This step is repeated until the number of high-criticality tasks and their total utilization U_{high} reaches a value of 50%. Then, for each low-criticality task a value between 10 and 100 is randomly generated. This value represents the importance of this task in the system (in practice less importance levels would be used). Finally, we assume for each low-criticality task that $S_{i,max} = 2$.

For the evaluation, three task sets are generated. In order to get as close as possible to expected MC systems, we assume that each task set is made of 20% of high-criticality tasks. The first task set (TS_1) is made of 50 tasks with 5 high-criticality tasks. The second task set (TS_2) is made of 60 tasks with 12 high-criticality tasks. Finally, the third task set (TS_3) is made of 70 tasks with 14 high-criticality tasks. For each set, we generated the tasks three times so that the initial total CPU utilization is 100%, 125% and 150% on a 2 processors system.

TABLE I. Obtained $S_{i,min}$ according to metrics *Aver*, *Aver25+* and *Aver75* for the task sets TS_1, TS_2 and TS_3 respectively.

U	*Aver*	*Aver25+*	*Aver75*	*Aver* w/o V_i
125	1.69/1.36/1.59	1/1/1	1.94/1.48/1.79	1.65/1.3/1.48
150	1.86/1.65/1.83	1.5/1/1.37	2/1.87/2	1.97/1.67/1.74

B. Stretching factors analysis

We use the following metrics to evaluate the behavior of our task model for the different aforementioned initial total CPU utilization: *Aver*, *Aver25+* and *Aver75*. *Aver* is the average stretching factor for all the low-criticality tasks. *Aver25+* is the average stretching factor for the 25% most important low-criticality tasks, while *Aver75* is the average stretching factor for the remaining tasks, i.e. the 75% less important tasks.

Table I shows the numerical results obtained for the stretching factors $S_{i,min}$ according to the previously introduced metrics and for each initial utilizations on a 2 processors system. The last column presents the results of $S_{i,min}$ when the importance level parameter is not used, i.e. all the low-criticality tasks have the same importance. Therefore, all the low-criticality tasks will have the same $S_{i,min}$ value, making the use of the *Aver25+* and the *Aver75* metrics unnecessary. In each cell, the three values are respectively the value of $S_{i,min}$ for the task set TS_1, TS_2 and TS_3. The result for the initial utilization of 100% is omitted as all the stretching factors are equal to 1 by construction.

As expected, these results show that the stretching factors are reduced for the most important tasks (*Aver25+*) and much higher for the less important tasks (*Aver75*). For instance, when the initial utilization is 150%, the most important low-criticality tasks have in average their stretching factors ranging from 1 (stretching is not required) to 1.5. On the other hand, the less important low-criticality tasks have in average their stretching factors ranging from 1.87 to 2, the maximum possible value ($S_{i,max}$). Table I also shows that without the importance level the stretching factors are slightly inferior (column 5) than when the importance parameter is used (column 2). That is, the low-criticality tasks should be stretched more when using the importance level parameter. However, when using this parameter in the model to compute stretching factors, the most important low-criticality tasks (column 3) have their stretching factors greatly reduced and even not used in most cases. These results demonstrate that our improved model of elastic tasks allows to execute both the low and the high-criticality tasks of a MC system, while giving first priority to the high-criticality tasks and then to the low-criticality tasks according to their importance level.

Figure 1 shows the distribution of the values of $S_{i,min}$ in two different configurations. Tasks are ordered by decreasing importance level. Configuration A corresponds to the task set TS_3 with an utilization of 150% where the values of V_i are randomly generated. In the configuration B, the application designer specified that the 25% most important low-criticality tasks should have their values of $S_{i,min}$ set to 1.25, while the other tasks can have higher stretching factors ($S_{i,max} = 2$). Such control over the values of the stretching factor for each task opens the opportunity for application designers to more easily dimension the required processing power when

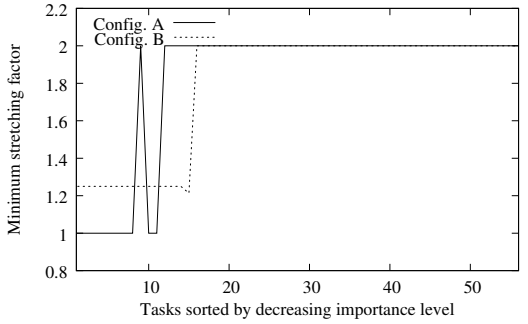


Fig. 1. Evolution of the distribution of stretching factors when setting the $S_{i,min}$ for the 25% most important low-criticality tasks.

designing a MC system.

VI. Conclusion

Within real-time embedded systems, there is a need to embed less (or even non-) safety tasks in addition to hard real-time tasks. This requires executing applications with different levels of criticality on a same chip. Such systems are called Mixed-Criticality (MC) systems. Certification constraints to prove the schedulability of MC systems lead to an over sizing of the CPU power, compared to what is required in average while the system is running. Such an approach is no longer compatible with current economical constraints that push for minimizing the CPU power.

In this work, we propose a task model and an associated on-line decision algorithm to maximize the execution rate of the low-criticality tasks within a safety-critical real-time system. Our task model, inspired by the elastic task model, allows to specify an order between the low-criticality tasks for applying so called stretching factors. Off-line, the minimum value for these stretching factors so that a MC system can be scheduled are computed. On-line, we then show how these stretching factors are used to relax the temporal constraints of the low-criticality tasks in order to avoid any deadline miss, while maximizing the execution rate of the low-criticality tasks. This approach can be used to size the required processing power for designing a MC system.

In future work, we plan to evaluate the actual values stretching factors can take depending on the distribution of the actual execution time of the low-criticality tasks. Besides, we plan to evaluate the overhead introduced by the different possible on-line decision algorithms for increasing/resetting the stretching factors. We also plan to investigate a different approach for supporting the execution part of our contribution, by relying on the use of a generalized form of the time-triggered paradigm, called eXternal-Triggered (xT) [8]. Using this paradigm, recomputing the visibility dates of low-criticality tasks being stretched would no longer be necessary. Finally, it would be interesting to apply the proposed task model in order to lessen the deadline miss ratio of the low-criticality tasks when setting a trade-off with energy consumption [15].

References

[1] P. Albertos, A. Crespo, I. Ripoll, M. Valles, and P. Balbastre. Rt control scheduling to reduce control performance degrading. In *Proc. of the 39th IEEE Conf. on Decision and Control*, volume 5, pages 4889–4894, Sydney, Australia, 2000.

[2] S. Baruah and A. Burns. Implementing mixed criticality systems in ada. In *Proc of the 16th Ada-Europe Intl. Conf. on Reliable software technologies*, pages 174–188, Edinburgh, UK, 2011.

[3] S. K. 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 *24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*, pages 145–154, Pisa, Italy, July 2012.

[4] E. Bini and G. C. Buttazzo. Biasing effects in schedulability measures. In *Proc. of the 16th Euromicro Conf. on Real-Time Systems (ECRTS 2004)*, pages 196–203, 2004.

[5] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Trans. Comput.*, 51(3):289–302, Mar. 2002.

[6] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. Litmus^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proc. of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*, pages 111–126, Washington, USA, 2006.

[7] A. Cervin and J. Eker. The control server: a computational model for real-time control tasks. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 113–120, 2003.

[8] D. Chabrol, D. Roux, V. David, M. Jan, M. A. Hmid, P. Oudin, and G. Zeppa. Time- and angle-triggered real-time kernel for powertrain applications. In *Proc. of the Design, Automation & Test in Europe Conference & Exhibition (DATE 13)*, pages 1060–1063, Grenoble, France, March 2013.

[9] R. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Syst.*, 47(1):1–40, Jan. 2011.

[10] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Proc. of the 30th IEEE Real-Time Systems Symposium (RTSS 2009)*, pages 291–300, Washington, DC, USA, December 2009.

[11] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson. Rtos support for multicore mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 197–208, 2012.

[12] M. Jan, V. David, J. Lalande, and M. Pitel. Usage of the safety-oriented real-time OASIS approach to build deterministic protection relays. In *Proc. of the 5^{th} Intl. Symp. on Industrial Embedded Systems (SIES 2010)*, pages 128–135, Trento, Italy, 2010.

[13] H. Kopetz. The time-triggered model of computation. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS'98)*, pages 168–177, Madrid, Spain, 1998.

[14] G. Koren and D. Shasha. Skip-over: algorithms and complexity for overloaded systems that allow skips. In *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, pages 110–117, 1995.

[15] V. Legout, M. Jan, and L. Pautet. Mixed-criticality multiprocessor real-time systems: Energy consumption vs deadline misses. In *Proc. of the 1st workshop on Real-Time Mixed Criticality Systems*, Tapei, Taiwan, August 2013.

[16] H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *Design, Automation and Test in Europe (DATE 13)*, pages 147–152, Grenoble, France, March 2013.

[17] M. Velasco, J. M. Fuertes, and P. Marti. The self triggered task model for real-time control systems. In *24th IEEE Real-Time Systems Symposium (work in progress, RTSS 2003)*, pages 67–70, Cancun, Mexico, 2003.

[18] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proc. of the 28th IEEE Intl. Real-Time Systems Symposium (RTSS 2007)*, pages 239–243, Tucson, USA, 2007.

[19] D. Zhu and H. Aydin. Reliability-aware energy management for periodic real-time tasks. *IEEE Transactions on Computers*, 58(10):1382–1397, 2009.

Multi-Criteria Evaluation of Partitioned EDF-VD for Mixed-Criticality Systems Upon Identical Processors

Paul Rodriguez
Université Libre de Bruxelles
ECE Paris
paurodri@ulb.ac.be

Laurent George
University of Paris-Est
LIGM / ECE Paris
lgeorge@ieee.org

Yasmina Abdeddaïm
University of Paris-Est
LIGM / ESIEE
yasmina.abdeddaim@esiee.fr

Joël Goossens
Université Libre de Bruxelles
joel.goossens@ulb.ac.be

***Abstract*—In this paper, we consider the partitioned EDF-VD scheduling problem of mixed critical systems with two criticality levels (LO and HI) on identical multiprocessors. Partitioned scheduling is an NP-hard problem that has been widely studied in the literature. The most common metaheuristic to solve partitioning problems consists in ordering tasks by a given criteria (such as task utilization) then assign tasks to processors in that order, choosing which processor using a heuristic rule such as First Fit or Best Fit. The current state of the art results show that First Fit Decreasing Density provides the best success ratio for single-criticality scheduling. In the context of mixed-criticality, we would like to investigate whether this is also true for assigning LO and HI critical tasks to processors. We consider two cases, one called "criticality aware" that first tries to assign HI tasks to processors and then LO tasks separately and the other one called "criticality unaware" that assigns tasks without taking their criticality into account. We test the performance of all combinations of sorting/partitioning heuristics in both cases, which leads to 1024 different heuristics in the aware case and 32 in the unaware case. We define two search algorithms to efficiently find which of these heuristics obtains the best success ratio. In addition, a new mixed-criticality multiprocessor random task set generation algorithm is proposed.**

I. Introduction

In this paper, we consider the problem of multiprocessor scheduling of mixed critical periodic tasks in the dual criticality case (LO and HI). In the multiprocessor case, two main paradigms have been considered for the scheduling of real-time tasks: the global scheduling and the partitioning approaches.

The global scheduling approach allows a job to migrate during its execution. At any time, a job is run only by one processor but the scheduler can decide to migrate it to another processor. The global scheduling approach provides better feasibility bounds but does not take into account job migrations costs.

In this paper, we consider the Partitioned scheduling problem. With the partitioning approach, the feasibility analysis on a multiprocessor requires to solve two problems:

- First, find a partitioning algorithm according to a partitioning heuristic.
- Second, use a uniprocessor feasibility condition on each processor to decide on the schedulability of the task set.

The partitioning approach therefore consists in statically assigning tasks to processors and then in solving the feasibility problem for a given partitioning on each processor. The problem of finding a feasible partitioning is a bin packing problem known to be NP-hard in the strong sense [1].

The partitioned approach received much more attention in the industry than the global one as it is a natural extension of uniprocessor scheduling. Partitioning has the advantage of utilizing all the feasibility results of uniprocessor scheduling [2] but also introduces some pessimism. In pathological cases, a partitioned system could be unfeasible with utilization just above 50% of the platform's capacity but simulation results [3] show that partitioned scheduling offers good success ratio (the ratio of successfully scheduled task sets over all task sets considered) even for high utilization. Furthermore, partitioned scheduling remains the industry standard in multiprocessor real-time systems and is for this reason still an important research subject, especially when all tasks do not have the same criticality. This means that the system may be subject to various certification processes, which are only concerned with the validation of a subset of the functionalities.

Furthermore, the certification processes are carried out using analysis methods whose rigor depends on the criticality of the tasks that need to be certified. Mixed-criticality systems are an attempt to model systems that need to be certified at various levels of assurance. In practice, a task that is subject to multiple certification processes will be characterized by multiple estimations of its Worst-Case Execution Time (WCET), some of which are more pessimistic than others. This reflects the differences in rigor adopted by the certification authorities.

The more a certification authority wants to ensure a task will never exceed its WCET, the more conservative its estimation will be. Nevertheless, when certifying that a task will meet its constraints, the assurance level that is used for other tasks is equal to the criticality of that particular task. This means that when running the system, if another task is run at a level of assurance that is higher (w.r.t. to its execution duration) than the criticality of the initial task, then this task will be suspended, since the conditions that guarantee its feasibility are no longer met.

A. This paper

In this paper we want to compare a set of criticality aware and unaware partitioned heuristics using experimental success ratio measurements based on task sets created through a new random system generator. As the large number of contesting heuristics introduces a considerable workload, special evaluation techniques based on a race approach are used to find the

best performing heuristic in the least amount of tests. These methods are also described in this paper through pseudocode.

B. Organisation

In Section II a brief review of other works in mixed-criticality uniprocessor and multiprocessor scheduling is given. In Section III the mixed-criticality task set model and notations are recalled. Section IV covers the description of the partitioning problem and a definition of the partitioning heuristics that are used in this paper and Section V covers our methods to evaluate the average success ratios of these heuristics. In Section VI our random task set generation algorithm is explained in detail as well as the results of our experiments.

II. Related Work

Mixed-Criticality scheduling is an emerging research domain which has gained a lot of interest in the past years. This approach was first introduced by Vestal [4]. In his work, he highlighted the difficulty in computing exact WCETs, and observed that in practice, the higher the degree of assurance required that a task will never exceed its WCET, the more conservative the approximation of the latter becomes. This degree of assurance is characterized by a level of criticality. He also suggested a fixed-task-priority strategy based on the Audsley priority assignment scheme [5]. Dorin et al. [6] proved that under the restricted case of independent task systems with constrained-deadlines, Vestal's modified Audsley's approach was optimal in the class of fixed-task priority algorithms. Nowadays, the Mixed-Criticality (MC)-Schedulability problem is commonly known to arise in two different contexts. The first one is concerned with applications that are subject to multiple certification requirements. In this context, different Certification Authorities (CA) need to validate the application functionalities. Nevertheless, the more critical a functionality is, the more pessimistic the CA will be in the estimation of the WCET. Baruah et al. [7] studied mixed-criticality systems in this context, but restricted their work to a set of mixed-criticality jobs. In particular, Baruah [8] pointed out the intractability of the MC-Schedulability problem, and quantified the fundamental limitations of MC-Scheduling for certification considerations. To tackle the intractability of MC-Scheduling, they suggest two sufficient schedulability conditions, referred to as the WCR-schedulability and OCBP-schedulability conditions. Later, Baruah and Li [9] extended their previous work and suggested a fixed-job-priority scheduling strategy based on their OCBP-schedulability condition. Baruah et al. also adapted the Earliest Deadline First algorithm to mixed-criticality systems, by modifying the deadlines of tasks. This approach is known as EDF-VD and is the one under study in this paper. More recently, Guan et al. [10] presented a new approach for scheduling mixed-criticality systems, which relies on an offline fixed-job-priority ordering computation, which is then used on-line by the scheduler. At the same time, Baruah et al. [11] formalized the response time analysis for mixed-criticality tasks. In [12] an overview of mixed-criticality scheduling on multiprocessors is proposed.

III. Model and notations

This paper is set in the context of constrained deadline periodic synchronous dual-criticality real-time task systems on a discrete timeline model. Each system is represented by a task set $\tau = \{\tau_1, \tau_2, \ \dots \ \tau_n\}$ where each task τ_i is a 4-uple $(C_i^{LO}, C_i^{HI}, D_i, T_i)$ where C_i^{LO} is the worst case execution time (WCET) in LO mode, C_i^{HI} is the WCET in HI mode, D_i is the relative deadline (the maximum allowed amount of time between an arrival and the corresponding end of this task) and T_i is the inter-arrival time (the period). All tasks satisfy the conditions $0 < C_i^{LO} \leqslant D_i \leqslant T_i$. If $C_i^{HI} > 0$, τ_i is said to be a HI criticality task and additionally $C_i^{LO} < C_i^{HI} \leqslant D_i$. A few additional notations are defined :

- The set of HI tasks : $\tau_{HI} = \{\tau_i \in \tau \mid C_i^{HI} > 0\}$
- The set of LO tasks : $\tau_{LO} = \{\tau_i \in \tau \mid C_i^{HI} = 0\}$
- Task τ_i utilization in LO and HI modes : $U_{LO}(\tau_i) = \frac{C_i^{LO}}{T_i}$ and $U_{HI}(\tau_i) = \frac{C_i^{HI}}{T_i}$
- Task τ_i density in LO and HI modes : $\lambda_{LO}(\tau_i) = \frac{C_i^{LO}}{\min(D_i, T_i)}$ and $\lambda_{HI}(\tau_i) = \frac{C_i^{HI}}{\min(D_i, T_i)}$
- LO mode system utilization : $U_{LO}(\tau) = \sum_{\tau_i \in \tau} U_{LO}(\tau_i)$
- HI mode system utilization : $U_{HI}(\tau) = \sum_{\tau_i \in \tau} U_{HI}(\tau_i)$
- n_{LO} and n_{HI} are (respectively) the size of τ_{LO} (respectively) τ_{HI}
- $n = n_{HI} + n_{LO}$

IV. Partitioned EDF-VD

A. EDF-VD Feasibility Condition

EDF-VD is a mixed-criticality uniprocessor scheduling algorithm [13]. It has later been extended to multiprocessor platforms [14] by using the same concept as fpEDF [15], a multiprocessor single-criticality scheduling algorithm. EDF-VD performs by applying EDF on a set of tasks where the HI criticality tasks have smaller relative deadlines when the system is in LO mode. In [13] all such virtual deadlines (VD) are the product of a unique factor (the value is the same for all tasks in the system) with each of their original deadlines. A improvement over EDF-VD is made in [16] by defining an heuristic algorithm (tuneSystem) that reduces virtual deadlines on a per-task basis. The schedulability test corresponding to tuneSystem is also defined in [16]. This finer grained uniprocessor test is the one used by the partitioning heuristics in this paper, as it displays among the best average success ratio in the current uniprocessor mixed-criticality scheduling state of the art.

Ekberg and Yi [16] extend the common definition of the demand bound function (DBF) to mixed-criticality systems. For a dual-criticality task set τ, two demand bound functions are defined : $dbf_{LO}(t)$ and $dbf_{HI}(t)$. These two functions have the properties of a single-criticality DBF. This means that we have the EDF feasiblity condition :

$$\forall t > 0 : dbf_{LO}(t) \leqslant t \text{ and } dbf_{HI}(t) \leqslant t$$
$$\iff \text{the system is schedulable}$$

To verify schedulability using these conditions, the tuneSystem algorithm (see Algorithm 1) is run [16]. tuneSystem like EDF-VD will shift the load of the HI mode DBF towards

the LO mode DBF by reducing the virtual LO mode deadlines of HI tasks. However, tuneSystem makes accurate modifications to single tasks instead of multiplying all of them by a factor, which makes it more complex but also more powerful than EDF-VD [16]. Each time some t that does not satisfy the condition is found, a deadline is changed to fix the problem if possible. When the virtual deadlines of multiple tasks have been reduced, the algorithm can also backtrack some of these changes in order to maintain the condition on the LO mode DBF. This continues until either the condition is satisfied or no deadline changes can be made anymore, in which case the system is not schedulable using tuneSystem. This algorithm achieves much higher success ratio than EDF-VD for utilization above 80% as EDF-VD starts to fail the scheduling of some systems at around 70% utilization [16]. Before running the algorithm, a bound ($tBound$) on the latest time instant for which the dbf condition must be verified is computed using the technique in [17]. Doing this in HI and LO mode results in two different bounds, out of which the highest must be chosen.

Algorithm 1: tuneSystem

Require: $tBound$ as defined in text
$candidates \leftarrow \tau_{HI}$
$changed \leftarrow$ **true**
$modTask \leftarrow \epsilon$
while $changed$ **do**
  $changed \leftarrow$ **false**
  for $t = 0$ **to** $tBound$ **do**
    if $dbf_{LO}(t) > t$ **then**
      if $modTask = \epsilon$ **then**
        return false
      end if
      increment the virtual deadline of $modTask$
      if $modTask \in candidates$ **then**
        $candidates \leftarrow candidates \setminus modTask$
      end if
      $modTask \leftarrow \epsilon$
      $changed \leftarrow$ **true**
      break
    end if
    if $dbf_{HI}(t) > t$ **then**
      if $candidates = \phi$ **then**
        return false
      end if
      $modTask \leftarrow$ task in $candidates$ which HI dbf increases the most between t and $t-1$
      decrement the virtual deadline of $modTask$
      if the WCET in LO mode of $modTask$ is equal to its virtual deadline **then**
        $candidates \leftarrow candidates \setminus modTask$
      end if
      $changed \leftarrow$ **true**
      break
    end if
  end for
end while

B. Partitioning heuristics

The problem of finding an assignment of tasks to processors that fits a given platform is similar to the Bin-Packing problem (BPP) which is known to be NP-hard. Variants of BPP frequently occur in computer science problems and this resulted in various heuristics being developped in the literature [18]. In this paper the focus will be put on heuristics following a strict framework (which can be thought as a metaheuristic) : items (tasks) are sorted given a specific *criteria* then they are assigned to bins in that order. The choice of which bin a given item will be assigned to is specified by an *assignment rule*. Together, the sorting criteria and assignment rule unequivocally describe the heuristic [19]. A total of four possible task ordering criteria are considered : utilization, period, deadline and density (U, P, L and D for short) and two possible orders for each of them (increasing or decreasing, respectively I and D). Additionally, the most frequent assignement rules are First Fit, Next Fit, Best Fit and Worst Fit (F, N, B and W). In the single-criticality partitioning literature, Best Fit and First Fit with Decreasing Utilization or Decreasing Density are found to display the best success ratio [19].

All variants considered, there is a total of 32 possible heuristics following this "criticality unaware" framework. In this paper a new type of heuristic ("criticality aware") is defined specifically for mixed-criticality systems. In this new kind of heuristic, the task set τ is split into HI and LO task sets, τ_{HI} and τ_{LO}. First, tasks in τ_{HI} are partitioned on all processors following one of the heuristics described above. Then, tasks in τ_{LO} are partitioned on the remaining space on the platform following another (possibly the same) heuristic. There are 1024 different heuristics following this new structure. In criticality aware heuristics, Worst Fit and Best Fit behave a little differently, as it makes more sense to use the utilization in HI mode during the assignment of tasks in τ_{HI} and the utilization in LO mode during the assignment of tasks in τ_{LO}. Following the same principle, in all heuristics LO criticality tasks of are ordered using their C_i^{LO} value as WCET (which is needed to calculate utilization and density) and HI criticality tasks with their C_i^{HI} value as WCET. Heuristics are noted using their assignement rule followed by the two letters code of their ordering criteria. For example, Best Fit Increasing Period will be noted B_{IP}. Criticality aware heuristics are noted by giving the heuristic for the LO tasks followed by a slash then the heuristic for the HI tasks, such as B_{IP}/N_{IL}.

V. Partitioning heuristics selection

One of the goals of this paper is to assess which heuristic has the best success ratio by experimentation or to what extent the kind of tested system can influence which heuristic is best. The results of such experiments heavily rely on which system generation algorithm was used (described in Section VI-A) but also on its parameters. Both heuristic evaluation methods use the same system generation parameters, which can be described as follows :

- System tailored for 4 processors
- 20 tasks, out which 8 are HI criticality tasks
- Both $U_{LO}(\tau)$ and $U_{HI}(\tau)$ randomly (and independently) distributed between 0 and the number of processors
- The minimum period $tMin = 5$ and the maximum period $tMax = 50$

A. Racing between heuristics

The large number of possible heuristics and long computation time required to test systems on many heuristics leads to a need for efficiency. In Algorithm 2, the search of the best heuristic is concentrated towards those that showed a good success ratio in past experiments. Such selection algorithms are usually called racing algorithms and are used in machine learning for model selection and parameter tuning [20]. In our context this approach enables us to quickly eliminate non significant heuristics. The search algorithm itself uses three parameters :

- $nRounds$ is the number of times the outer loop (called round) of the algorithm is executed, corresponding to the number of times the working set of heuristics is shrunk.
- $nTests$ is the number of systems generated and tested during the first round.
- $exploration$ is the factor (between 0 and 1) of reduction of the size of the working set of heuristics. Additionally the number of tested systems is multiplied by $\frac{1}{exploration}$ each round.

Algorithm 2: Racing for heuristics

Require: $nRounds$, $nTests$ and $exploration$ as defined previously.
$heuristics \leftarrow$ all possible heuristics
$curTests \leftarrow nTests$
for $r = 1$ **to** $nRounds$ **do**
  for $t = 1$ **to** $curTests$ **do**
    $system \leftarrow$ randomly generated system
    for all $h \in curHeuristics$ **do**
      partition $system$ with h
      update success ratio of h with the result
    end for
  end for
  $curHeuristics \leftarrow$ the $exploration^r$ portion of $heuristics$ with highest success ratio
  $curTests \leftarrow curTests/exploration$
end for
return $heuristics$ sorted by success ratio

The time complexity of Algorithm 2 in terms of number of partitionings is in $O(nRounds \cdot nTests \cdot |heuristics|)$. Note that the number of partitionings per round does not change. In later rounds, fewer heuristics are tested on a larger number of systems.

B. Direct elimination

Direct elimination is a slightly different form of racing. In Algorithm 3 heuristics are selected for further testing based on their ability to dominate other heuristics rather than simply success ratio. Each run begins with all heuristics being tested on one system. If at least one heuristic could schedule the system, all the heuristics that were unable to schedule it are discarded. This is repeated until only one heuristic remains in the set of heuristics (or if we have reasonable evidence that stability has been reached, see $stability$), then the whole set of heuristics is reset and the operation is repeated.

- $nRuns$ determines the number of times the complete operation (starting with all heuristics and reducing until stability) is done.
- $stability$ is the maximum amount of systems that will be tested without eliminating heuristics.

Algorithm 3: Direct heuristic elimination

Require: $nRuns$ and $stability$ as defined previously.
for $r = 1$ **to** $nRuns$ **do**
  $heuristics \leftarrow$ all possible heuristics
  while $t < stability$ and $|heuristics| > 1$ **do**
    $system \leftarrow$ randomly generated system
    $schedules \leftarrow$ mapping of all heuristics to **false**
    for all $h \in heuristics$ **do**
      partition $system$ with h
      if h can partition $system$ **then**
        $schedules[h] \leftarrow$ **true**
      end if
      update success ratio of h with the result
    end for
    if $\exists i \mid schedules[i]$ and $\exists j \mid \neg schedules[j]$ **then**
      $t = 0$
      for all $h \in heuristics$ **do**
        if $\neg schedules[h]$ **then**
          remove h from $heuristics$
        end if
      end for
    else
      $t = t + 1$
    end if
  end while
end for
return $heuristics$ sorted by success ratio

The worst case number of tests of Algorithm 3 is in $O(nRuns \cdot stability \cdot |heuristics|)$ as it will take a maximum of $nTest - 1$ tests to eliminate each heuristic individually. In practice Algorithm 3 is faster than Algorithm 2 because the actual number of tests depends on the generated systems for Algorithm 3 but not for Algorithm 2. One system is very often enough to eliminate a large portion of the set of heuristics, which means $stability \cdot |heuristics|$ is a very pessimistic estimate of the time required for one round.

VI. Experiments

A. Task set generation

Widely used standard random task set generation algorithms exist for uniprocessor single-criticality scheduling, such as UUniFast [21] for generating uniform task utilizations. But this standard way of generating task sets does not extend to more specific systems, where an ecosystem of techniques still exists. UUniFast has been extended to multiprocessor systems in [22], which is the basis of the multiprocessor mixed-criticality task set generation algorithm found in this paper. Other works in the literature have proposed techniques for generating mixed-criticality task sets, such as [14], [23] using an explicit scaling factor between the utilization in HI and LO mode.

In this paper task utilizations in LO and HI mode are generated taking the constraints specific to mixed-criticality

systems into account and in a way that aims to keep the uniformity of the generated systems intact. The generator receives guidelines on $U_{HI}(\tau)$, $U_{LO}(\tau)$, the ratio of HI tasks ($ratioHI = \frac{n}{n_{HI}}$) and the maximum and minimum period ($tMax$ and $tMin$). The generator will try to create a system meeting those guidelines as precisely as possible, although making no guarantees.

1) LO and HI utilizations: The algorithm generates n utilization values from $U_{LO}(\tau)$ for LO mode and n_{HI} utilization values from $U_{HI}(\tau)$ for HI mode using a variant of the multiprocessor UUniFast algorithm [22]. Those utilization values are then associated to one another in HI-LO couples making sure that for each HI task the HI utilization is higher than the LO utilization. If this cannot be achieved, HI utilizations are re-generated for one less HI task until one such association can be found. When a valid association is found, it is randomly shuffled with the limitation that is has to stay correct.

2) Periods, WCETs and deadlines: Periods are randomly chosen between $tMin$ and $tMax$ with a log uniform random variable biased towards lower values, as done in [22]. WCETs are then directly calculated from those periods and the utilizations. HI WCETs are checked to be at least one plus the corresponding LO WCET. Periods are then adjusted to make sure no task has utilization above or equal to one then ensuring that the total system utilization is below the given guidelines for LO and HI utilization. This is done by repeatedly choosing a random task in τ and incrementing its period until both conditions become satisfied. Finally, deadlines for each task are randomly chosen between the highest WCET (C_i^{LO} for LO tasks, C_i^{HI} for HI tasks) and the period with a logarithmic uniform random variable biased towards higher values.

B. Results

The methods explained in Section V-A and Section V-B agree on the general domination of the single-criticality heuristics F_{DU} and F_{DD} both in criticality unaware (confirming previous results in single-criticality systems [19]) and in criticality aware mixed-criticality partitioning heuristics. However the conducted experiments aggregated success ratios based on systems with utilizations in HI and LO mode ranging from 1 to the number of processors (4), giving the same weight to each system. In a realistic environment, it is expected that systems with higher utilization will be more interesting as we want to use the platform to maximum capacity (or use lighter hardware to run the same set of tasks). Three of the best heuristics have been run on new systems generated with a range of fixed HI and LO utilizations (all other parameters remaining the same) to show how their success ratio evolves with HI and LO utilization. The results for F_{DD}, F_{DD}/F_{DD} and F_{DD}/W_{DD} are respectively shown in Figure 3, 1 and 2.

The intuitive expectation is that using W_{DD} as a HI mode heuristic will give better results when $U_{LO}(\tau)$ is high and $U_{HI}(\tau)$ is low. This is motivated by the reasoning that if HI utilization is as balanced between processors as possible, it is more likely that more LO tasks will be schedulable over the whole system. However if we forget about LO tasks, W_{DD} performs worse than F_{DD} to find a good partitioning of HI tasks (the same way it performs worse in single-criticality scheduling), which means F_{DD}/F_{DD} has an advantage over F_{DD}/W_{DD} when partitioning task sets with high $U_{HI}(\tau)$ and low $U_{LO}(\tau)$. This is verified in our experiments as when

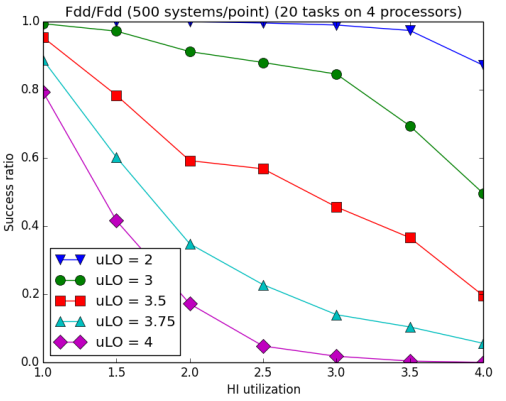


Fig. 1. Success ratio of F_{DD}/F_{DD} for various HI and LO utilizations

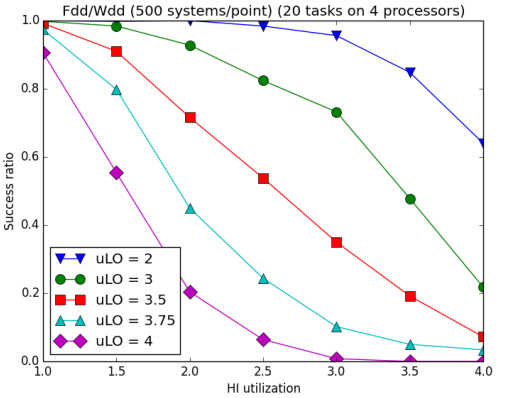


Fig. 2. Success ratio of F_{DD}/W_{DD} for various HI and LO utilizations

compared to F_{DD}/F_{DD}, F_{DD}/W_{DD} does globally better when HI utilization is lower than 2.5 if LO utilization is at least 3.5.

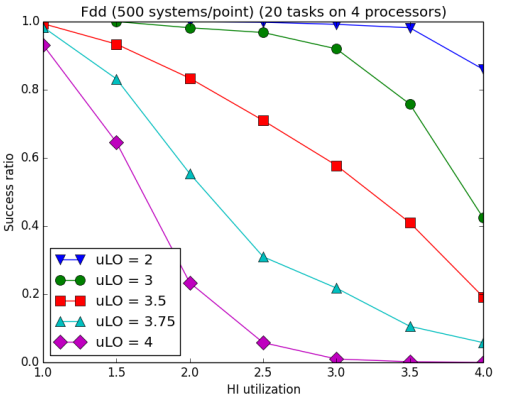


Fig. 3. Success ratio of F_{DD} for various HI and LO utilizations

F_{DD} obtains slightly better success ratio in most situations. However if we try to directly compare F_{DD} with F_{DD}/W_{DD}, we obtain surprising results.

In Figure 4 the amount of systems that are schedulable by F_{DD}/W_{DD} but not by F_{DD} is compared with the amount

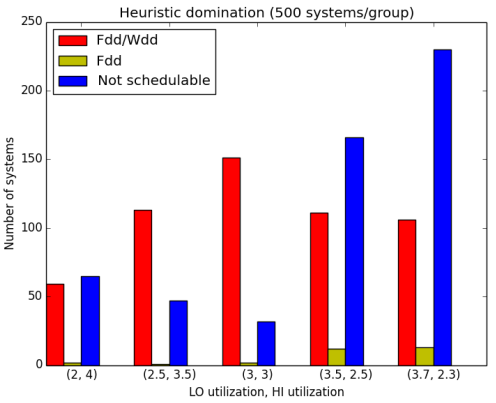


Fig. 4. Number of systems where F_{DD}/W_{DD} dominates F_{DD} and vice versa

of systems schedulable by F_{DD} but not by F_{DD}/W_{DD} for various values of $U_{LO}(\tau)$ and $U_{HI}(\tau)$. For all the chosen utilization combinations, there are very few systems that belong to the second category. This number increases as HI utilization increases and LO utilization decreases, which hints that not allowing LO tasks to be partitioned before any HI tasks (the definition of criticality aware heuristics) might not be a good choice in systems that have heavy LO tasks.

VII. Conclusion

In this paper, we have considered the problem of partitioned EDF-VD scheduling for mixed critical (HI and LO) periodic tasks on identical multiprocessors. In a mixed critical system, it is mandatory to grant HI critical tasks. We have studied different meta-heuristic that maximize the success ratio of LO critical tasks while granting HI critical tasks. We considered two partitioned scheduling approaches, one called criticality aware that first tries to assign HI tasks to processors and then LO tasks separately and one called "criticality unaware" that does not take into account the criticality of the tasks. We have adopted a race metaheuristic to select the best partitioning heuristics according to several placement and sorting criteria. We show that taking into account criticality levels by first assigning HI critical tasks leads to better success ratio for HI tasks. Furthermore, when Worst Fit with Decreasing Density succeeds to partition HI tasks, assigning LO tasks with Fist Fit Decreasing Density maximizes the success ratio of LO tasks.

References

[1] D. Johnson, "Fast algorithms for bin packing," *Journal of Computer and Systems Science*, vol. 8(3):272314, 1974.

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

[3] L. George, P. Courbin, and Y. Sorel, "Job vs. portioned partitioning for the earliest deadline first semi-partitioned scheduling," *Elsevier Journal of systems architecture, Special issue on multiprocessor real-time scheduling Ed. U.Devi and J.H. Anderson*, vol. 57, no. 5, pp. 518–535, 2011.

[4] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 239–243.

[5] N. C. Audsley, "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times," Tech. Rep. YCS-164, 1991.

[6] F. Dorin, P. Richard, M. Richard, and J. Goossens, "Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities," *Real-Time Syst.*, vol. 46, pp. 305–331, December 2010.

[7] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 13–22.

[8] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *Mathematical Foundations of Computer Science 2010*, pp. 90–101, 2010.

[9] H. Li and S. Baruah, "An algorithm for scheduling certifiable mixed-criticality sporadic task systems," in *31st IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 183–192.

[10] N. Guan, P. Ekberg, M. Stigge, and W. Yi, "Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems," in *32nd IEEE Real-Time Systems Symposium*. IEEE Computer Society, 2011, pp. 13–23.

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

[12] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin, "Mixed-criticality scheduling on multiprocessors," *Real-Time Systems*, pp. 1–36, 2013.

[13] S. K. Baruah, V. Bonifaci, G. DAngelo, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie, "Mixed-criticality scheduling of sporadic task systems," in *Algorithms–ESA 2011*. Springer, 2011, pp. 555–566.

[14] H. Li and S. Baruah, "Global mixed-criticality scheduling on multiprocessors," in *24th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2012, pp. 166–175.

[15] S. K. Baruah, "Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors," *IEEE Transactions on Computers*, vol. 53, no. 6, pp. 781–784, 2004.

[16] P. Ekberg and W. Yi, "Bounding and shaping the demand of generalized mixed-criticality sporadic task systems," *Real-Time Systems*, pp. 1–39, 2013.

[17] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 1990, pp. 182–190.

[18] R. Yesodha and T. Amudha, "A comparative study on heuristic procedures to solve bin packing problems," *International Journal*, 2012.

[19] I. Lupu, P. Courbin, L. George, and J. Goossens, "Multi-criteria evaluation of partitioning schemes for real-time systems," in *2010 IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2010, pp. 1–8.

[20] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp, "A racing algorithm for configuring metaheuristics." in *Proceedings of the Genetic and Evolutionary Computation Conference*, vol. 2. GECCO, 2002, pp. 11–18.

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

[22] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2010, pp. 6–11.

[23] N. Guan, P. Ekberg, M. Stigge, and W. Yi, "Improving the scheduling of certifiable mixed-criticality sporadic task systems," Tech Rep 2013-008, Department of Information Technology, Uppsala University, Tech. Rep., 2013.

Mixed Criticality Scheduling Applied to JPEG2000 Video Streaming Over Wireless Multimedia Sensor Networks

Alemayehu Addisu
Dip. di Informatica e Elettronica
Politecnico di Torino
C.so Duca degli Abruzzi, 24
10129 Torino, Italy
Email: alex.addisu@gmail.com

Laurent George
LIGM, Cite Descartes,
Copernicus Bat, 5 bd Descartes,
77454, Champs sur Marne,
France
Email: lgeorge@ieee.org

Vincent Sciandra
Transdev,
32 boulvard Gallieni
92130, Issy-les-Moulineaux,
France
Email: sciandra@ece.fr

Max Agueh
LACSC ECE,
37, Quai de Grenelle
75725 Paris Cedex 15,
France
Email: agueh@ece.fr

***Abstract*—In this paper, we propose a mixed-criticality scheduling scheme for selection of JPEG2000 codestream features to be transmitted over Wireless Multimedia Sensor Networks (WMSNs). We extend the application of mixed-criticality scheduling model to the wireless domain. We show that by adopting mixed-criticality scheduling scheme, an improved end-to-end response time is gained with respect to the classical case where all information exhibit the same level of importance.**

I. Introduction

Thanks to the integration of inexpensive complementary metal-oxide semiconductors (CMOS) cameras and microphones, WMSNs are emerged as interesting framework for applications such as enhanced surveillance and monitoring systems. These networks are capable of sensing multimedia content including audio, still images and videos in addition to scalar sensor data (e.g. temperature, humidity, etc ...) [1].

Several surveys were conducted on different aspects of wireless multimedia sensor networks. For example, [1] and [2] clearly highlights the state-of-the-art and main research challenges for development of WMSNs. These surveys discussed several characteristics of WMSNs, among which application scenarios, existing solutions and different open research issues. Since WMSNs are resource constrained networks (e.g limited computation power, reduced memory, narrow bandwidth, etc ...), image and video transmission over such networks is still an important challenge which needs to be addressed.

In this context, mixed-criticality scheduling is used to arbitrate flows generated from different sources. Each source uses JPEG2000 compression algorithm to encode images with multi-layer and multi-resolution features of JPEG2000. During transmission of image/video from the sources, all information do not possess the same level of criticality. Information from one source is more critical than that of other source depending on the available channel capacity. Hence, we can model our system based on the mixed-criticality properties.

The mixed-criticality nature of the system arises from the fact that while we would like to transmit all information (all layers and resolutions) under high availability of the bandwidth. However, it is important that the critical information has to be transmitted even when the bandwidth is low. We consider a communication channel with L-levels of bandwidth values and transmission of periodic images that possess different levels of criticality. Furthermore, the model is considered as a non-preemptive scheduling problem, in the sense that once the transmission of an image is started, it cannot be preempted by another with higher criticality.

Due to rapidly increasing cost, power and thermal dissipation constraints, there is an increasing trend in embedded system towards implementing multiple functionalities upon a single shared computing platform. Typically, all these different functionalities do not possess the same level of criticality to the overall system performance. This concept of mixed-criticalities gave rise to a mixed-criticality scheduling problem. It is initially introduced by Vestal [3]. He pointed out that there is a difficulty in computing the exact Worst Case Execution Times (WCETs). The more conservative the approximation of WCETs, the high level of assurance that the execution of the task never exceeds it WCET. This confidence levels form the basis for different levels of rigorousness (level of criticality) of the system. To solve the problem of mixed-criticality scheduling, he also suggested a fixed-task-priority strategy based on "Audsley approach" [4]. Based on this approach, other authors provide an improved way to tackle the intractability of mixed-criticality scheduling. For instance, Baruah et. al. [5] proposed an algorithm called Own Criticality Based Priority (OCBP) to schedule a mixed-criticality system with a finite number of jobs. In their work, they showed that OCBP-schedulability offers performance guarantee that is superior to performance guarantee offered by the Worst Case Reservations (WCR) schedulability. In [6], the authors proposed another algorithm, called Priority List Reuse Scheduling (PLRS) to schedule certifiable mixed-criticality sporadic tasks system. They used fixed-job-priority scheduling scheme and assigned job priorities by exploring and balancing the asymmetric effects between the workload of different criticality levels.

Through simulation, they found that the run-time complexity of PLRS is polynomial time.

In our work, we model the transmission channel as a non-preemptive uniprocessor with limited amount of transmission time. The channel has *L* levels of speed, which corresponds to the amount of available bandwidth in the network. We can represent *B(l)* as the available bandwidth at criticality level *l*∈ [1,*L*]. Hence, the transmission time $C_i(l)$ can be seen as the time it takes to send data units at a rate of *B(l)* and criticality level *l*.

The remainder of this paper is organized as follows. In section 2, we discuss mixed-criticality scheduling model along with some overviews on JPEG2000 compression algorithm and available bandwidth estimation tools. Then in section 3, the application of mixed-criticality scheduling to video streaming system is given. In section 4, implementation of the proposed algorithm is detailed. Experiments and results are provided in section 5. Then, conclusion and future work are followed in section 6.

II. Model, Definitions and Overviews

A. Mixed-criticality Scheduling

1) ***MC tasks and jobs*** *:* We consider the scheduling of Mixed-criticality(MC) tasks on a non-preemptive single processor with task sets $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Furthermore, the maximum criticality of a task is bounded by *L*. Each MC task is characterized by 5-tuple $\tau = \{R_i, T_i, D_i, \chi_i, C_i\}$ where:

- $R_i \in \mathbb{N}$ is the release time of the first job of task τ_i
- $T_i \in \mathbb{N} \setminus \{0\}$ is the period of task τ_i
- $D_i \in \mathbb{N} \setminus \{0\}$ is the deadline of task τ_i, $D_i \leq T_i$
- $\chi_i \in \mathbb{N}$ is the maximum criticality of the task τ_i, $\chi_i \leq L$
- $C_i \in \mathbb{N}^{\mathbb{L}}$ is a size *L* vector of WCETs, where $C_i(l)$ is an estimation of the WCET of task τ_i at criticality level $l \in$ [1,*L*]

We assume $C_i(l)$ is monotonically non-decreasing for increasing *l*. More precisely, for task τ_i:

- $\forall\ m \in [1, \chi_i] : C_i(m) \leq C_i(m+1)$
- $\forall\ m \in [\chi_i, L] : C_i(m) = C_i(\chi_i)$

A *job* in MC system is characterized by a 5-tuple of parameters: $J_j = \{r_j, d_j, \chi_j, C_j, c_j\}$ where:

- $r_j \in \mathbb{N}$ is the release time of the job J_j,
- $d_j \in \mathbb{N}$ is the absolute deadline,
- $\chi_j \in \mathbb{N}^+$ is the criticality of the job,
- $C_j \in \mathbb{N}^{\mathbb{L}}$ is a size *L* vector of WCETs of J_j,
- $c_j \in \mathbb{N} \setminus \{0\}$ is the exact execution of the job J_j.

The idea of the MC job model is: job J_j is released at time r_j, has deadline at d_j, and needs to execute for some amount at time c_j. However, the value of c_j is not known beforehand, but only becomes revealed by actually executing the job until it signals that it has completed execution.

At any time, we call a job is *available* if its release time has passed and the job has not signalled execution completion. Let us define a notion of a *scenario*. Each job J_j requires an amount of execution time c_j within $[r_j,\ d_j]$. We call a collection of execution times $S = \{c_1, c_2, ..., c_n\}$ a *scenario* and it consists of *n* jobs.

The criticality level of a scenario *S* can be defined as the smallest integer *l* such that $c_j \leq C_j(l)\ \forall j$. If no such *l* exists, then the scenario is said to be *erroneous*, since at least one task exceeds its WCET at its own criticality.

2) ***MC-schedulability****:* In literature, it is shown that MC-scheduling problem is NP-hard in strong sense. For example in [5], Baruah et al. proved that when MC is applied to a finite set of jobs, it is not possible to find a solution in polynomial time. This condition forces the research community to come up with a *sufficient* condition that can be verified in polynomial time. In [3], Vestal determined a total ordering of the tasks in τ offline. Each task is assigned a distinct priority and jobs inherit the priority of the task that released them. At each moment, the scheduler dispatches the available job with the highest priority. The priority assignment is realized using "Audsley approach" based on the following definition with assumption of priority *n* be the lowest priority and 1 be the highest priority.

Definition: A task τ_i in a mixed-criticality task set τ is said to be *viable at the lowest priority level* if all of the following conditions hold true:

1) the lowest priority is assigned to τ_i,
2) all other tasks in τ can be assigned any priority provided that these priorities are higher than the priority assigned to τ_i and
3) every job released by τ_i meets its deadline when it is executed for at most $C_i(\chi_i)$ time units and all other tasks τ_j in τ generate jobs that run for at most $C_i(\chi_i)$ time units.

The procedure is repeatedly applied to the set of jobs excluding the lowest priority job, until all jobs are ordered, or at some iteration a lowest priority job does not exist.

Since the priority of the a task is based on its own criticality level, we can say that a task set is *Own Criticality Based Priority(OCBP)-schedulable* as long as we find a complete ordering of the tasks.

B. Worst case end-to-end response time

In real-time applications, *timeliness* is one of Quality of Service (QoS) parameters which has utmost importance. For example, in real-time video/audio streaming applications, delay matters. If the system is unable to deliver frames within a sliding window of period, the frames arrived outside the window will be discarded. So, this results in reduced Quality of Experience (QoE), i.e., less visual comfort for users. Shorter end-to-end response time helps the frames to arrive within

sliding window and it contributes to the overall performance of the system.

Several approaches can be used to determine the maximum end-to-end response time, such as *stochastic* or *deterministic* approach [7]. Furthermore, the *deterministic* approach is divided into *holistic* and *trajectory* approaches. In our work, we consider *trajectory* approach that gives better estimation of the end-to-end response time when compared to the *holistic* approach.

C. JPEG2000

JPEG2000 is an image compression standard and coding scheme. In addition to its high coding efficiency, JPEG2000 also provides with a number of highly desirable features such as seamless progressive transmission by resolution or quality, lossy to loss-less compression, random codestream access and processing, and region of interest. In [8], JPEG2000 is covered in much detailed way.

D. AVAILABLE BANDWIDTH ESTIMATION

Accuracy of available estimated bandwidth and convergence delay algorithm are research challenges in wireless network measurements. In wireless communication networks, the available bandwidth could be used as an important parameter to take decisions concerning many issues such as load control, admission control and routing.

Available bandwidth estimation methods can be divided in two major approaches:

- *Intrusive approaches* - these methods are based on end-to-end probe packets to estimate the available bandwidth on the link.
- *Passive approaches* - they use local information on the used bandwidth (e.g. the channel usage computed by sensing the radio medium) and exchange this information through *Hello* messages that are used in many routing protocols.

In [9], Prasad et. al presented four types of bandwidth estimation tools which uses intrusive approaches. All techniques are provided in their paper.

In our work, we rely on an active probing available bandwidth estimation tool called *Wireless Bandwidth estimation tool (WBest)*. This algorithm is proposed in [10]. The authors demonstrated that *WBest* has higher accuracy and faster convergence time in wireless environment with respect to other tools. They made a comparison with existing available bandwidth estimation tools such as: IGI/PTR , PathChirp, and Pathload. They recommended *WBest* for multimedia streaming applications over wireless networks. In their work, they pointed out that finding the optimal length of the trains used in the steps is a difficult matter. They proposed that 10 packet pairs for the first train and 30 packets for the second train are good choices, which yield a sufficiently accurate bandwidth estimation results. We also adopt these choices in our work.

III. JPEG2000 VIDEO STREAMING

In this section, first we define our architecture for the wireless multimedia sensor network. The wireless network is composed of Raspberry Pi platforms[11] and we name it π-*sense* network. Then we setup the mixed-criticality scheduling model for the wireless network.

A. Network Setup

The raspberry pi platform is used as sensor node. It is a credit-card-sized single-board computer with several peripheries for different purposes. EDIMAX WIFI dongle [12] is attached to USB port of the raspberry pi board. The dongle complies with wireless 802.11 b/g/n standards with data rates up to 150 Mbps and supports smart transmit power control and auto-idle state adjustment.

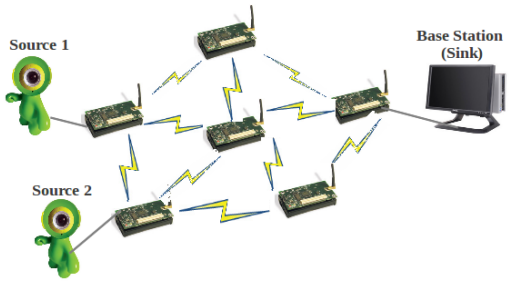


Figure 1. The π-sense wireless network

One remaining part to add to this section is the routing protocol we used in the π-sense network. In [13], Murray et al. made comparisons between *Optimized Link State Routing (OLSR)*, *Better Approach To Mobile Ad hoc Networking (BATMAN)*, and *Babel* routing protocols. They investigated the cause of performance loss or gain in multi hop ad hoc networks. They pointed out that in small networks, *Babel* offers higher throughputs due to reduced protocol overhead. In our mesh network of raspberry pi platforms, we use *Babel* routing protocol.

The whole network is depicted in figure 2. In the π-sense network, there are two sources transmitting video/image to the sink.

B. MC-wireless Model

In this section, we setup the proposed scheduling algorithm for the π-sense wireless network. Since the available bandwidth in the network varies, we took bandwidth estimates (bits/sec) in discrete format by using *WBest* estimation tool. We did this process offline so that we can know beforehand the bandwidth needed for the transmission of the images/frames.

We represent $B(l)$ as an estimate of available bandwidth at

criticality level $l \in$ [1,L]. Furthermore, we assume that $B(l)$ is monotonically decreasing for increasing criticality level l, i.e.,

- $B(l+1) \leq B(l)\ \forall l \in [1, L]$

We consider mixed-criticality periodic frame sets $f = \{f_1, f_2, ..., f_n\}$ on non-preemptive communication channel where maximum criticality of the frame is L. The channel has bandwidth of $B(l)$ units per second, $\forall\ l \in$ [1,L].

Each MC frame is characterized by a 5 tuple $f_i = \{R_i, T_i, D_i, \chi_i, C_i\}$ where:

- $R_i \in \mathbb{N}$ is the release time of the first packet of the frame f_i,
- $T_i \in \mathbb{N}^*$ is the period of the frame f_i,
- $D_i \in \mathbb{N}^*$ is the deadline of the frame f_i, $D_i \leq T_i$,
- $\chi_i \in \mathbb{N}$ is the maximum criticality of the frame f_i, $\chi_i \leq L$,
- $C_i \in \mathbb{R}^+$ is an L vector of transmission times of frame f_i.

By definition, $C_i(l)$ is the time needed to transmit frame f_i of length N_i at criticality level $l \in [1, L]$. It is defined as:

$C_i(l) = \frac{N_i}{B(l)}$

The rest of this section deals with the computation of worst case end-to-end response time (WCERT). The WCERT is computed at each level of criticality.

*1) **Notations**:* The following notations are used in our work:

- $w_{i,m}$ is the relative start time of transmission of frame f_i for the m^{th} period T_i
- $WCERT_i$ is the worst case end-to-end response time of frame f_i
- $C_i^h(l)$ is transmission time of frame f_i from node h at criticality level l
- L_{max}, L_{min} - maximum/minimum network delay between two consecutive nodes in the wireless network
- Jin_i^1 is the maximum jitter of frame f_i at source node
- $hp(i,l)$ is the set of frames having a priority higher than f_i at criticality $\chi_i \geq l$
- $lp(i,l)$ is the set of frames having a priority lower than f_i at criticality $\chi_i \geq l$
- $H_i^{1,h}$ is the maximum delay incurred by frame f_i due to $f_j \in lp(i,l)$, while going from source node (1) to sink (h)

*2) **Worst case end-to-end response**:* To use *trajectory* approach, we assume that all flows from both sources follow the same path to the sink. In π-sense network, the sources are two hops away from the sink. Hence, nodes are marked from 1(source) to 3. The end-to-end response time is obtained by summing the delays incurred on each node along the path of the flows.

Due to non-preemption effect, the transmission of a high priority frame (from source 1) can be delayed so that the lower priority frame (from source 2) finishes its transmission. The maximum delay incurred by f_i due to $f_j \in lp(i,l)$ when both of them follow the same path to the sink can be given as:

$$\begin{cases} H_i^{1,1}(l) & = max_{f_j \in lp(i,l)} \left\{C_j^1(l) - 1\right\} \\ H_i^{1,h+1}(l) & \leq H_i^{1,1}(l) + max_{f_j \in lp(i,l)} \left\{C_j^{h+1}(l)\right\} \\ & \quad - min_{f_j \in hp(i,l) \cup f_i} \left\{C_j^h(l)\right\} \\ & \quad + L_{max} - L_{min} \end{cases}$$

Property 2 gives an upper bound on the maximum delay incurred on the path $h \in [source, sink]$. In [7], Martin et al. gave proof of the upper bound on the delay.

Before obtaining the worst case end-to-end response time, we need to find the latest release time of f_i when the path consists of q nodes. It is given by:

$$\begin{cases} w_{i,t}^q(l) & = \sum_{f_j \in hp(i,l)} (1 + \lfloor \frac{w_i^q(l) + Jin_i^1(l)}{T_j} \rfloor) \text{x} C_j^{slow}(l) \\ & \quad + (1 + \lfloor \frac{t + Jin_i^1(l)}{T_i} \rfloor) \text{x} C_i^{slow}(l) \\ & \quad + \sum_{h=1}^{q} (max_{f_j \in hp(i,l) \cup f_i} \left\{C_j^h(l)\right\}) - C_i^q(l) \\ & \quad + H_i^{1,q}(l) + (q-1) \text{x} L_{max} \end{cases}$$

For the latest release time, the prove of existence of solution and the upper bound is also given in [7].

Finally, the worst case end-to-end response time is given by:

$$WCERT_i^{1,q}(l) = max_{k=0..K} \left\{w_{i,t}^q(l) + C_i^q(l) - k.T_i + Jin_i^1(l)\right\}$$

where the value of K is also provided in [7].

In the next section, implementation of MC-wireless is given. Experimental results are shown in section 5.

IV. MC-wireless Implementation

In this section, we implement mixed-criticality scheduling scheme for the π-sense WMSN. First, we assign fixed priority for the information transmitted from the two sources (as shown in figure 1). That is, at any time, source 1 gets higher priority over source 2. The priority assignment can be based on *location*, for example. Assume that source 1 is at the entrance of a building and source 2 is inside the building. It is necessary that frames from source 1 is transmitted to the sink even if the available bandwidth is dropped. In this situation, source 2 stops transmission so that source 1 transmits its frames within short period of time. When the available bandwidth is enough, both sources transmit their frames over the wireless network.

Secondly, we define criticality levels that corresponds to the values of the available bandwidth. In our work, we set number of criticality levels to 3, i.e., $l \in$ [1,3]. Hence, $B(l)$ is the available bandwidth at criticality level $l \in$ [1,3].

At source nodes, we implement JPEG2000 algorithm to encode the frames. The frames can be critical or non-critical depending on the information encoded in it. For instance, when the available bandwidth is too low, the frame is encoded with

base layer and resolution (that means it is critical frame). Hence, even if the bandwidth is low, we can still get a frame with lower quality from source 1, but source 2 is disconnected due to its lower importance.

frame	frame length(KB)	BW needed(Mbps)
f_1	5.7	$B(l) \leq 1$
f_2	7.6	$1 < B(l) \leq 1.8$
f_3	15.2	$B(l) > 1.8$

Table I
NEEDED BANDWIDTH FOR THE FRAMES

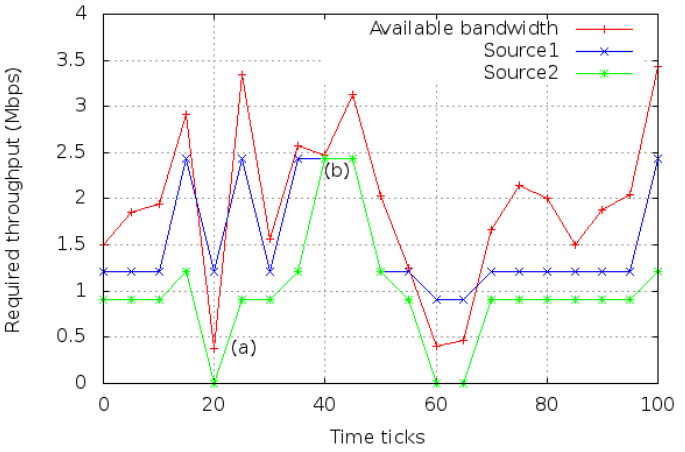


Figure 2. Required bandwidth for the sources

From table 1, we see that when the available bandwidth is greater than 1.8 Mbps, both critical and non-critical information (all layers with full resolution - i.e, f_3) can be transmitted to the sink. However, if the bandwidth is less than 1 Mbps, it is necessary to transmit only the critical information (i.e, f_1).

Figure 2 shows required bandwidth for the two sources along with estimated available bandwidth. The bandwidth values fall into range of [0.1, 3.427] Mbps. We divide the range into 3 parts. The first part is when the bandwidth is in [0.1, 1.0] Mbps. It corresponds to criticality level 3 (i.e., B(3) $\in$ [0.1, 1.0]). Then, if the available bandwidth is in (1.0, 1.8] Mbps, we say that it is at criticality level 2 (i.e., B(2) $\in$ (1.0, 1.8]). Finally, criticality level 1 corresponds to bandwidth values above 1.8 Mbps.

There are 2 important points to consider in figure 2. At point (a), the available bandwidth is dropped to 0.384 Mbps from prior value of 2.914 Mbps. This means, it is changed from criticality level 1 to 3 (i.e., the change is more than 1 step). In this situation, the appropriate selection from source 1 could be frame f_1 that requires a bandwidth of 0.912 Mbps (taking frame rate of 20 fps). However, due to comfort of user visualization, we select frame f_2 that requires 1.2 Mbps. Clearly, the time it takes to transmit f_1 is lower than that of f_2, but we have a good quality image at the end. Thus, it is a trade-off between the quality of the image and transmission time. Finally, it actually takes 158 ms (7.6KB/0.384Mbps) to transmit f_2. For source 2, we stop transmitting the frames because it has lower priority and the bandwidth is dropped too much.

Secondly, at point (b), the estimated available bandwidth is at criticality level 1. Hence , it is sufficient to transmit both critical and non-critical information from source 1. In this case, source 2 can also transmit its frames.

V. EXPERIMENTS

In this section, we present the results gained by adopting mixed-criticality scheduling to the π-sense network. The MC-wireless is compared against the classical case in which all information are considered equally important.

Let us consider a classical situation where all the information can be sent without considering the available bandwidth. This corresponds of transmission of both critical and non-critical information (i.e. f_3). In this case, even if the bandwidth is at criticality level 3, we transmit f_3. However, according to MC-wireless, we prioritize f_1 over f_3 because the the bandwidth required for f_3 is greater than that of f_1 and f_1 finishes transmission earlier than f_3. Hence, we get reduce transmission time by selecting f_1 over f_3.. This reduction in transmission time leads to an improved end-to-end response time.

The worst case end-to-end response time (WCERT) is shown in figure 3 for both classical and MC-wireless cases. From the figure, we see that MC-wireless improves the end-to-end response time due to the fact that MC classifies the information as critical/non-critical based on the available bandwidth. Experimental results of WCERTs are plotted for each criticality level, that is, WCERT_l represents the worst case end-to-end response time at criticality level $l \in [1, 3]$.

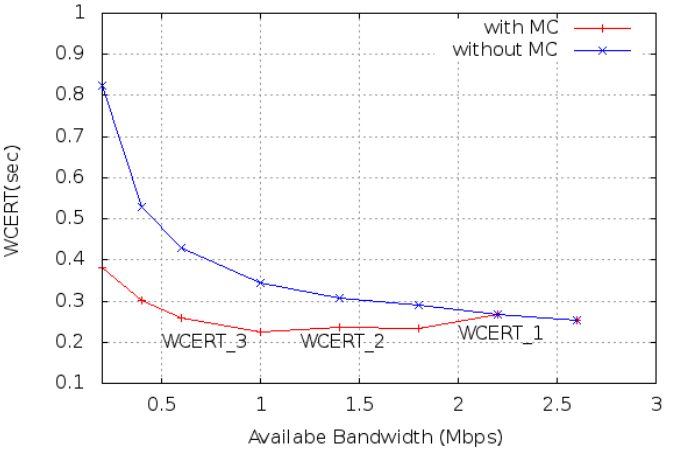


Figure 3. WCERT with and without MC

Let us consider a case when the available bandwidth is below 1 Mbps (region corresponding to WCERT_3). In this region, if we transmit all information (i.e. f_3), we end up incurring high WCERT. However, transmitting only the critical information

(i.e. f_1), we gain a reduction in WCERT (from 0.822614 sec to 0.382091 sec when the available bandwidth is 0.2012 Mbps). This trend continues until the available bandwidth becomes more than 1.8 Mbps. Hence, in the third region ($WCERT_1$), since the available bandwidth is enough to send all the information, both MC-wireless and classical approaches provide the same value of WCERT.

To provide the upper bound on WCERT, we calculate end-to-end response time using the *trajectory* approach. On the path from source 1 to sink, we have 3 nodes. So, we compute delays at each node and obtain the response time for criticality level 3. Since source 2 is not transmitting at this criticality level, the delay due to non-preemption is ignored.

$$WCERT_1^{1,3}(3) = Jin_1^{source1}(3) + C_1^1(3) + C_1^2(3) + C_1^3(3) + 3*(L_{max} - L_{min}) = 0.623886 + \text{K sec}$$

where $Jin_1^{source1}(3)$ = 0.17023 sec, $C_1^1(3)$ = $C_1^2(3)$ = 5.7KB/0.2012Mbps = 0.2266 sec and $C_1^3(3)$ = 5.7KB/100Mbps = 0.000456 sec (the last node and sink are connected through Ethernet cable). We can assume that the processing time ($L_{max} - L_{min}$) on each node is not significant compared to other transmission times. Furthermore, since the bandwidth of every node is not exactly known, we can bound their bandwidths by estimated available bandwidth. Some of the nodes may have higher bandwidth, but estimating WCERT by available bandwidth will give an upper bound on the WCERT time. Hence,

$$WCERT_1^{1,3}(3) = 0.623886 \text{ sec.}$$

In our experiment, WCERT is found to be 0.382091 sec and it is 0.623886 sec using trajectory approach. The upper bound of WCERT will decrease if the capacity of each channel is known (as a case in LAN network).

VI. Conclusion and Future work

In this paper, we applied mixed-criticality scheduling scheme for wireless multimedia sensor networks. We showed the gains of adopting mixed-criticality in comparison to classical cases. An improved end-to-end response time is achieved by our experiments.

An interesting extension of our work can be to apply the proposed scheduling model to a larger network. In this case, it is possible to cluster nodes based on their location in the network. Such a scalable network will have a cluster head in each group. The cluster heads manage flows by using our proposed scheduling scheme. They also form another hierarchical layer and are linked to the sinks. Hence, applying the proposed scheduling scheme at each layer of the hierarchy allows addressing scalability issues in large networks.

References

[1] I.F. Akyildiz, T. Melodia, and K.R. Chowdhury, *Wireless Multimedia Sensor Networks: Applications and Testbeds*. IEEE Communications Surveys and Tutorials, VOL.10, NO.4, Fourth Quarter of 2008.

[2] S. Misra, M. Reisslein, and G. Xue, *A Survey of Multimedia Streaming in Wireless Sensor Networks*. Proceedings of the IEEE, VOL.96, NO.10, October 2008.

[3] S. Vestal, *Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance*. IEEE Computer Society, 2007, pp. 239-243.

[4] N.C. Audsley, *Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times* . Tech. Rep YCS-164, 1991

[5] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow and L. Stougie, *Scheduling real-time mixed-criticality jobs*. IEEE Communications on Computers 61, 8(2012) 1140-1152

[6] N. Guan, P. Ekberg, M. Stigge and W. Yi, *Effective and Efficient Scheduling of Certifiable Mixed-Criticality Sporadic Task Systems*. IEEE Computer Society, 2011 pp. 183-192

[7] S. Martin, P. Minet, and L. George, *End-to-end response time with Fixed Priority Scheduling: Trajectory approach versus Holistic approach*. INRIA: France, 2004

[8] S. Taubman and W. Marcellin, *JPEG2000: Standard for Interactive Imaging* . Proceedings of the IEEE, VOL. 90, NO.8, August 2002.

[9] R.S. Prasad, M. Murray, C. Dovrolis, and K. Claffy, *Bandwidth Estimation: Metrics, measurement techniques and tools*. IEEE Network, vol 17, pp. 27-35, 2003

[10] M. Li, M. Claypool, and R. Kinicki *WBest: a Bandwidth Estimation Tool for IEEE 802.11 Wireless Networks*. Computer Science Department at Worcester Polytechnic Institute, Worcester, MA: USA

[11] Raspberry Pi : ⟨http://www.raspberrypi.org/⟩

[12] EDIMAX : ⟨http://www.edimax.com/en/produce_detail.php?pd_id=347&pl1_id=1⟩

[13] D. Murray, M. Dixon, and T. Koziniec, *An Experimental Comparison of Routing Protocols in Multi Hop Ad Hoc Networks*. Murdoch University: Australia, 2010

State-Based Mode Switching with Applications to Mixed-Criticality Systems

Pontus Ekberg, Martin Stigge, Nan Guan and Wang Yi
Uppsala University, Sweden
Email: {pontus.ekberg | martin.stigge | nan.guan | yi}@it.uu.se

***Abstract*—We present a new graph-based real-time task model that can specify complex job arrival patterns and global state-based mode switching. The mode switching is of a mixed-criticality style, enabling immediate changes to the parameters of active jobs upon mode switches. The resulting task model therefore generalizes previously proposed task graph models as well as mixed-criticality (sporadic) task models, and further allows for the modeling of timing properties not found in any of these models. We outline an EDF uniprocessor schedulability analysis procedure by combining ideas from prior analysis methods for graph-based and mixed-criticality task scheduling.**

I. Introduction

We present the Mode-Switching Digraph Real-Time (MS-DRT) task model, a model that can express complex arrival patterns of jobs and global mode switching. The tasks are represented with graphs that specify both the arrival patterns and the synchronization points (mode switches) between tasks. MS-DRT is a strict generalization of the Digraph Real-Time (DRT) task model [1] and of the common mixed-criticality sporadic task model [2], its variations [3] and generalizations [4]. MS-DRT also enables the modeling of many timing properties that have no counterpart in the above models, some examples of which are shown in Section III.

The modes in MS-DRT are system-wide, meaning that all tasks synchronously switch from one mode to another. Mode-switching logic is specified per state (vertex) of the task graphs, so that behaviors may differ depending on the local state of the tasks. The mode change protocol is of a generalized mixed-criticality style, enabling immediate changes to the timing parameters of active jobs at mode changes. As opposed to the usual mixed-criticality setting, it is possible to have cyclic mode changes in MS-DRT. In addition to being a mixed-criticality task model, MS-DRT could, for example, find applications as a timing model for statecharts [5] by considering the orthogonal components as tasks and expressing their arrival and synchronization patterns.

We outline an EDF schedulability analysis procedure for MS-DRT task systems on uniprocessors. The analysis procedure combines ideas from previously published EDF schedulability analysis methods for DRT task systems [1] and mixed-criticality sporadic task systems [6], [4]. These are all based on computing demand bound functions for tasks, and are therefore possible to combine.

A. Related Work

Mixed-criticality scheduling theory has seen a process of slowly generalized task models. After the seminal paper by Vestal [2], which described fixed-priority response-time analysis for mixed-criticality sporadic task systems, the initial research effort was based on scheduling static sequences of mixed-criticality jobs. The work by Baruah et al. [7] provides a good overview of such mixed-criticality job scheduling.

One of the scheduling theories developed for static job sequences, the OCBP scheduling approach, was then generalized for sporadic tasks systems by Li and Baruah [8]. Shortly thereafter, Baruah et al. developed a new EDF-based scheduling algorithm, called EDF-VD [9], for mixed-criticality sporadic tasks. The initial work by Vestal on response-time analysis of fixed-priority scheduling was also improved by Baruah et al. [10]. This list is by no means exhaustive, many other works have since been based on the mixed-criticality sporadic task model.

EDF-based scheduling of mixed-criticality sporadic tasks was further investigated by Ekberg and Yi [6]. This was based on very similar runtime scheduling as the previous work on EDF-VD by Baruah et al. [9], but the analysis was based on computing demand bound functions for the mixed-criticality tasks. Demand bound functions offer a handy abstraction for use in EDF-based schedulability analysis, and have been successfully applied to many varying task models outside of the mixed-criticality setting. For example, EDF scheduling analyses based on demand bound functions have been presented for task models that offer greater expressiveness than sporadic tasks regarding job arrival patterns, such as the GMF [11] and DRT [1] task models. This wide applicability of demand bound functions is what allows us analyze a combination of mixed-criticality style mode switching with more general task models for this paper.

Baruah [3] has also proposed a variation of the standard mixed-criticality sporadic task model, in which the periods of sporadic tasks rather than their execution-time estimates are subject to uncertainties. A generalization by Ekberg and Yi [4] covers the case where all parameters of the sporadic tasks may change, and the potential mode switches can be expressed as a directed acyclic graph instead of being linearly ordered. Contrary, the MS-DRT task model proposed in this paper is not constrained to sporadic behavior, and allows cyclic mode switches.

II. The Mode-Switching Digraph Real-Time Task Model

In this section we describe the syntax and semantics of the MS-DRT task model in as abstract terms as possible. Concrete examples of MS-DRT tasks, outlining some more real-world interpretations of the semantics, are presented in Section III.

A. Syntax

An MS-DRT task system is formally defined by a set of tasks $T = \{\tau_1, \ldots, \tau_n\}$ together with an associated set of modes $M(T) = \{\mu_1, \ldots, \mu_k\}$. An MS-DRT task $\tau \in T$ is defined by an ordered tuple $(V(\tau), E_{\text{cf}}(\tau), E_{\text{ms}}(\tau))$, where

- $V(\tau)$ is a set of vertices, representing *job types*,
- each vertex $v \in V(\tau)$ is labeled with an ordered tuple $\langle e(v), d(v), \mu(v)\rangle \in (\mathbb{N}_{\geqslant 0} \times \mathbb{N}_{\geqslant 0} \times M(T))$, representing worst-case execution time, relative deadline and mode of the corresponding job type, respectively,
- $E_{\text{cf}}(\tau)$ is a set of directed edges representing task control flow, such that $\mu(u) = \mu(v)$ for each $(u, v) \in E_{\text{cf}}(\tau)$,
- each edge $(u, v) \in E_{\text{cf}}(\tau)$ is labeled with a minimum inter-release separation time $p(u, v) \in \mathbb{N}_{\geqslant 0}$,
- $E_{\text{ms}}(\tau)$ is a set of directed edges representing possible mode switches, such that $\mu(u) \neq \mu(v)$ for each $(u, v) \in E_{\text{ms}}(\tau)$.

We assume that each task $\tau \in T$ satisfies the *frame separation property*, a generalization of the *constrained deadlines* concept for sporadic tasks. In other words, for each vertex $u \in V(\tau)$ and $(u, v) \in E_{\text{cf}}(\tau)$ we have $d(u) \leqslant p(u, v)$.

Note that, by the above definition, $E_{\text{cf}}(\tau)$ and $E_{\text{ms}}(\tau)$ are disjoint sets, that $(V(\tau), E_{\text{cf}}(\tau))$ is a directed graph with up to k disjoint subgraphs (one subgraph per mode of the task), and that $(V(\tau), E_{\text{ms}}(\tau))$ is a directed multipartite graph (colorable with one color per mode).

B. Semantics

An MS-DRT system consists of a number of tasks that all run in the same mode at any particular time point, i.e., the modes are system wide. While running in any mode, an MS-DRT task τ behaves in the same way as an ordinary DRT task: the task runs by traversing the graph $(V(\tau), E_{\text{cf}}(\tau))$, releasing a job every time a vertex is visited. More formally, a job is defined by a triple $(r, e, d) \in \mathbb{R}^3$, representing the job's release time, execution-time budget and absolute deadline, respectively. A valid job sequence $[(r_1, e_1, d_1), (r_2, e_2, d_2), \ldots]$ is generated by τ if there is a path $(\pi_1, \pi_2, \ldots)$ through $(V(\tau), E_{\text{cf}}(\tau))$ such that for all i

- $e_i = e(\pi_i)$,
- $d_i = r_i + d(\pi_i)$,
- $r_{i+1} \geqslant r_i + p(\pi_i, \pi_{i+1})$.

As edges in $E_{\text{cf}}(\tau)$ only go between vertices labeled with job types of the same mode, traversing $(V(\tau), E_{\text{cf}}(\tau))$ never causes the task to switch mode.

For any point in time and any mode μ, if the latest visited vertex u of each task $\tau \in T$ has an outgoing mode-switch edge $(u, v) \in E_{\text{ms}}(\tau)$ with $\mu(v) = \mu$, then an event can trigger the system to switch to mode μ. At that time point, each task τ immediately and synchronously switches to the new mode μ through one of the edges in $E_{\text{ms}}(\tau)$, chosen non-deterministically if there are several such edges. Note that the model does not specify the origin of the events triggering mode switches, but rather just says that such events can arrive at any time. Any event-triggering scheme chosen by the system designer is then valid for the model. For example, mode-switch events can be emitted due to the run-time behavior of the tasks themselves, or due to execution-time overruns of jobs. They could also be the result of errors or faults, or come from external sources.

The mode switch protocol is of a (generalized) mixed-criticality style, meaning that if the last released job of τ is still active (released, but not finished), it immediately has its parameters changed to that of the job type at the target vertex. In this way, the job types labeled on any two vertices u, v, for which $(u, v) \in E_{\text{ms}}(\tau)$, can be thought of as representing different versions of *the same job*. Note that when a mode-switch edge $(u, v) \in E_{\text{ms}}(\tau)$ is taken, no new job is released at vertex v. The job type labeled on v serves to update the parameters of a job from u that is still active as follows.

1) The job's total execution-time budget is changed from $e(u)$ to $e(v)$, but is not replenished. If the job has already executed for at least $e(v)$ time units, it is immediately considered to be finished.
2) The job's absolute deadline is changed to be $d(v)$ time units after its release time.

Jobs that are active during a mode switch are called *carry-over jobs*. Note that a job is still eligible to become a carry-over job at the time point where its execution-time budget reaches zero; this allows modeling of mode switches due to execution-time overruns.

After the switch, the tasks can go on to generate new job sequences by continuing to traverse $(V(\tau), E_{\text{cf}}(\tau))$, now only being able to visit vertices labeled with job types of the new mode. The inter-release separation constraints hold across mode switches. In other words, if the last released job of τ (active or not) was released at time t in a previous mode, then the first control-flow edge $(v, w) \in E_{\text{cf}}(\tau)$ to be followed in the new mode can be taken earliest at time $t + p(v, w)$.

A system may start with any mode as the initial one, and with any vertices with job types of that mode as the initial vertices of the tasks.[1]

III. Examples

In this section we present some example tasks, showing a few of the properties that can be modeled with the MS-DRT task model. In the figures we draw the control-flow edges as straight arrows and the mode-switch edges as wiggly arrows. The colors help reading, but carry no semantic information.

[1] In practice, systems will likely have just one or a few well-defined initial states. However, allowing the system to start in any reachable state does not negatively affect schedulability. As the goal of the model is to over-approximate all the possible behaviors of the system, we found it counterproductive to add syntax to needlessly restrict the behaviors of the model.

Example III.1 (Dual-criticality tasks)**.** *Figure 1 shows some tasks that are similar to ordinary mixed-criticality tasks [2], but with some additional semantics that can not be expressed in the original model. Upon an execution-time overrun, the system would switch from mode* LO *to mode* HI*.*

- τ_1 *is equivalent to a dual-criticality sporadic task that gets its execution-time budget increased at a switch to the high-criticality mode (*HI*).*
- τ_2 *will instead drop any active job at a mode switch, and after a delay start a less intensive sporadic workload. Recall that the inter-release separation constraints hold transparently across mode switches, so the extra dummy job at u_3 is introduced to ensure that u_4 is visited no earlier than 100 time units after the mode switch as opposed to 100 time units after the last job release at u_1.*
- τ_3 *will stop releasing new jobs after a mode switch, but must finish any active job that it has at that time; the time given to finish the last job is increased to 70 time units instead of the 30 time units that are normally given.*
- τ_4 *is a direct extension of a two-vertex DRT task to the dual-criticality semantics with different execution-time estimates.*
- τ_5 *represents overhead from the mode switch (e.g., reordering priority queues) by creating a one-off job that must be executed immediately after the switch. Note that the "worst-case" behavior of τ_5 is to defer the release of a job at z_1 until the time point where a switch to* HI *occurs, and then immediately transforming it with the parameters at z_2. It is often the case that the model will have many possible behaviors that are irrelevant for the concrete system, as long as the behaviors of the system are a subset of those of the model it is not an issue for performing safety analysis.*

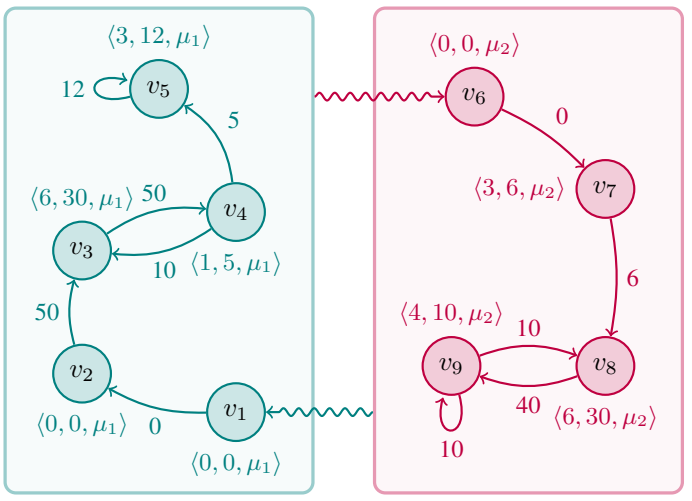


Fig. 2. An example task with coarse-grained mode switching.

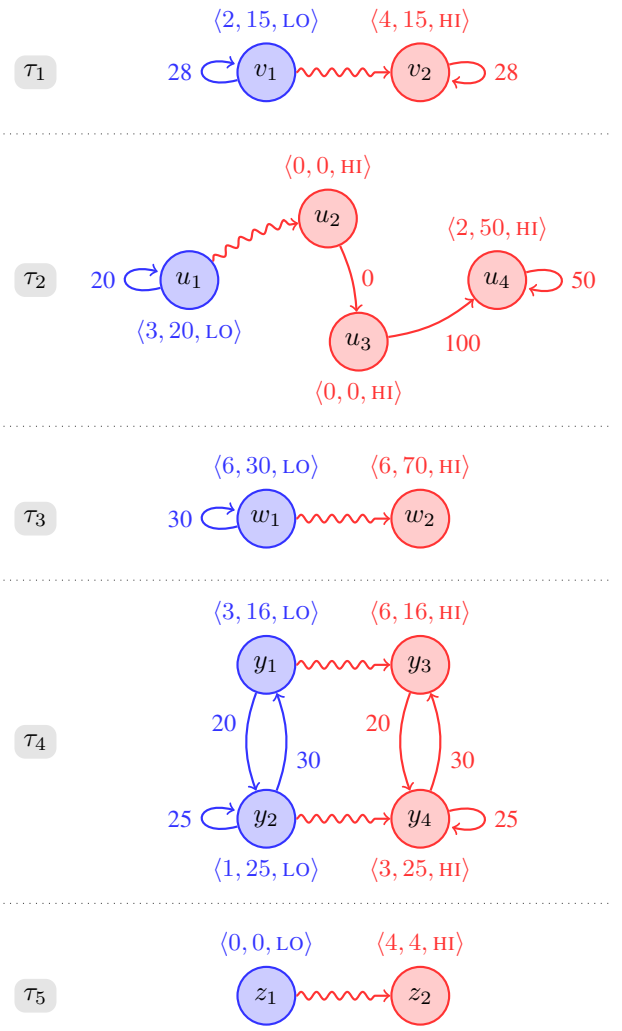


Fig. 1. Some example tasks of the ordinary dual-criticality style.

Example III.2 (Coarse-grained mode switching)**.** *More traditional, coarse-grained mode switching can also be modeled in MS-DRT. Figure 2 shows a task that only has a single entry point per mode. The rectangular boxes are syntactic sugar expressing that the outgoing edges have copies from each of the vertices inside the box. From any of the vertices $v_1, \ldots, v_5$, the task can switch to mode μ_2 by going to v_6. Any active job is dropped at v_6, and some initial work has to be performed immediately at v_7. The task can switch to v_1 in mode μ_1 from any of the vertices $v_6, \ldots, v_9$, in the process dropping any active job and delaying at least 50 time units before continuing to release the first non-dummy job at v_3. Note that while the model dictates that all active jobs of the task, if any, are dropped at a mode switch, a perfectly valid behavior of the system that is modeled would be to only switch to a new mode when the task actually has no active jobs.*

Example III.3 (Period-adapting tasks)**.** *In this example we model two tasks, shown in Figure 3, that periodically read some sensor values and release jobs to process the readings. Depending on the values that are read, the resulting jobs have*

different execution-time requirements. Some readings result in "small jobs" with an execution-time requirement of 1 time unit, and other readings result in "big jobs" that take up to 4 time units to complete.

The tasks are together adapting to the current sensor values by switching to different modes where the periodicity of the readings are matched against the execution-time requirements of the resulting jobs. In mode μ_1, both tasks are reading and processing small jobs at the highest possible pace in v_1 and u_1, respectively. If τ_1 reads a sensor value implying a big job, it goes to the dummy job v_5 and triggers a mode switch to mode μ_2, upon which the dummy job is transformed into a big job at v_2. At the same time, τ_2 must also switch to μ_2 and goes to u_3, where the current (small) job continues with the same execution-time budget. The period of the sensor readings in μ_2 is changed to 5 in order to match the total execution-time requirements of the jobs. If τ_1 later reads a sensor value implying a small job, it will go to v_9 and trigger a mode switch back to μ_1. Task τ_2 will switch back to μ_1 through u_8, allowing it to finish its current job with the deadline given to it before reverting back to the smaller period of sensor readings. Similarly, the system switches to mode μ_3 if τ_2 reads sensor values resulting in big jobs, and to μ_4 if both tasks do.

IV. Analysis

In this section we will briefly outline an EDF schedulability analysis of MS-DRT task systems on uniprocessors. It is based on ideas from previously published EDF schedulability analyses of regular DRT task systems [1] and mixed-criticality sporadic task systems [6], [4].

Following the analysis for the generalized mixed-criticality sporadic task model [4], we define the *mode structure* $G(T)$ of an MS-DRT task system T as the directed graph (V, E) where $V = M(T)$ is the set of modes and E contains edges for the possible mode switches.[2] That is, $(\mu_i, \mu_j) \in E$ if and only if each task $\tau \in T$ has vertices u, v such that $(u, v) \in E_{\text{ms}}(\tau)$ and $\mu(u) = \mu_i$ and $\mu(v) = \mu_j$. For example, Figure 4 shows the mode structure for the tasks in Example III.3.

To reduce the complexity of the schedulability analysis we analyze each mode and mode switch separately. For each mode μ_j we will analyze its schedulability during all possible time intervals that do *not* include a mode switch. For every mode switch $(\mu_i, \mu_j) \in E$ we will analyze the schedulability of μ_j for all possible time intervals that start with a switch from μ_i to μ_j, over-approximating any workload that can be carried over from the previous mode.

The analysis is based on finding *demand bound functions* for each task. The demand bound functions must safely over-approximate the total execution demand of all jobs from the task that together can have their entire scheduling windows (from release time to deadline) within a time interval of a given length. Let $\text{dbf}_{\mu_j}(\tau, \ell)$ denote a demand bound function for τ in mode μ_j for any time interval of length ℓ that do not contain

[2] In [4] the mode structure is constrained to be a directed acyclic graph, here it can be any directed graph.

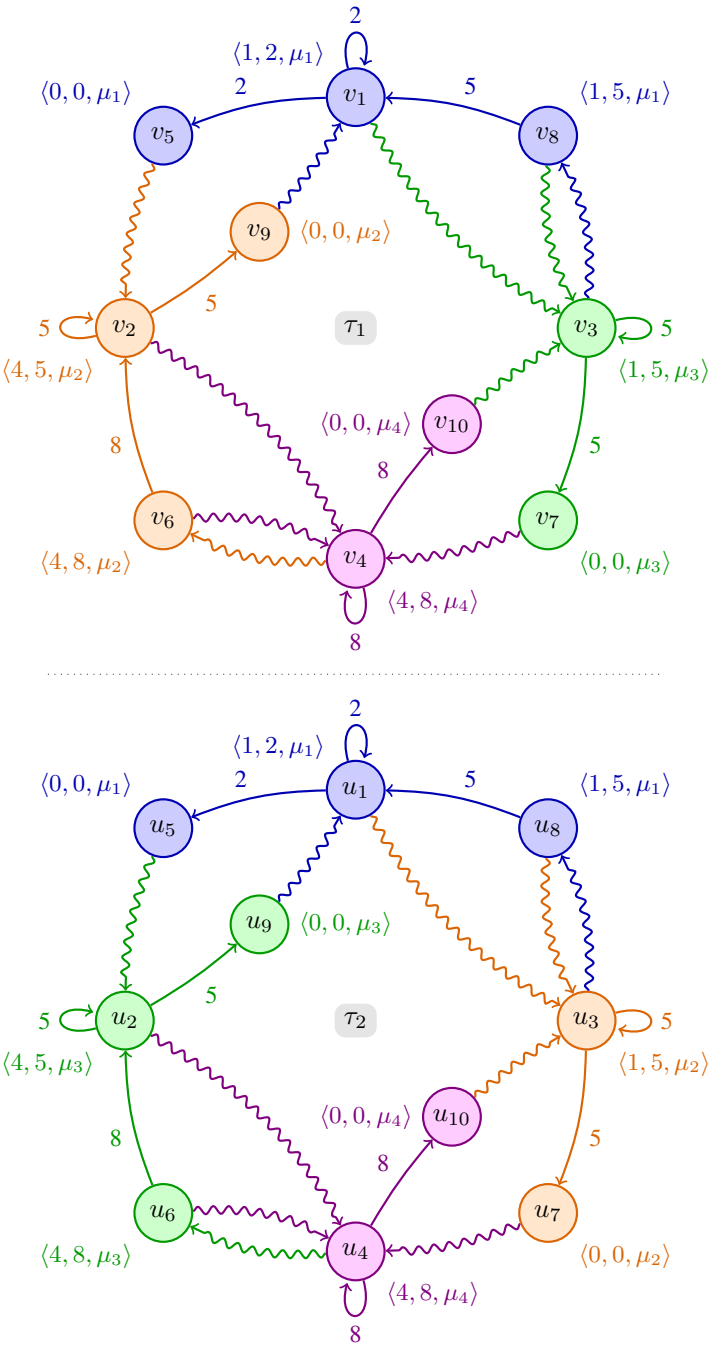


Fig. 3. Two tasks that dynamically adapt to each other's requirements.

a mode switch. Let $\text{dbf}_{\mu_i \to \mu_j}(\tau, \ell)$ instead denote a demand bound function for task τ in mode μ_j for any time interval of length ℓ that start with a switch from μ_i to μ_j. Only the latter kind of demand bound function will be over-approximate. The pessimism there comes from assuming the worst-case behavior in the previous mode when bounding the demand of carry-over jobs; this is the price that we pay for analyzing the modes in relative separation. Experimental results from [4], where a similar source of pessimism exists, shows that the resulting analysis procedure still offers good performance compared to

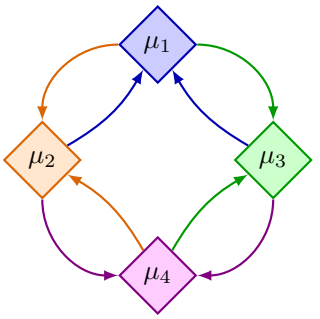


Fig. 4. The mode structure of the tasks in Example III.3.

other methods. Considering modes in separation also enables us to handle systems with cyclic mode switches with relative ease, as we only need to look to the immediately previous mode to over-approximate the demand of carry-over jobs.

The demand bound functions described above can be used in a schedulability test as expressed by the following proposition.

Proposition IV.1. *An MS-DRT task system T with mode structure $G(T) = (V, E)$ is schedulable by EDF if the following two conditions hold for all $\mu_j \in V$:*

$$\forall \ell \geqslant 0 : \quad \sum_{\tau \in T} \mathrm{dbf}_{\mu_j}(\tau, \ell) \leqslant \ell, \tag{1}$$

$$\forall \mu_i \in \mathrm{pred}(\mu_j), \forall \ell \geqslant 0 : \quad \sum_{\tau \in T} \mathrm{dbf}_{\mu_i \to \mu_j}(\tau, \ell) \leqslant \ell, \tag{2}$$

where $\mathrm{pred}(\mu_j) \stackrel{\text{def}}{=} \{\mu_i \mid (\mu_i, \mu_j) \in E\}$.

Conceptually, the above proposition checks the schedulability of each mode in complete isolation. It can be thought of as $|M(T)|$ separate schedulability tests. For each mode μ_j, we ensure in condition (1) that the demand of workload released entirely inside μ_j is schedulable and in condition (2) that workload including carry-over jobs from all possible predecessor modes is schedulable. Considering this, a proof of the above proposition can be made very similar to, e.g., the proof of Theorem 1 in [11] and is omitted for space reasons.

A. Computing demand bound functions

The computation of demand bound functions for MS-DRT tasks is based on the method for computing such functions for regular DRT tasks [1]. A quick review of this method is presented below.

1) Demand bound function computation for DRT: For any path $\pi = (\pi_1, \ldots, \pi_m)$ through a DRT graph, the execution-time demand $e(\pi)$ and deadline $d(\pi)$ of π is defined as

$$e(\pi) \stackrel{\text{def}}{=} \sum_{i=1}^{m} e(\pi_i),$$

$$d(\pi) \stackrel{\text{def}}{=} \sum_{i=1}^{m-1} p(\pi_i, \pi_{i+1}) + d(\pi_m).$$

To compute a (precise) demand bound function of a DRT task for a time interval length ℓ, we must find the maximum $e(\pi)$ for all paths π through the graph with $d(\pi) \leqslant \ell$.

As the number of paths through a graph grows exponentially with the path length, a path abstraction called *demand tuples* was introduced in [1] to alleviate this problem. Each demand tuple can abstractly represent several concrete paths that are equivalent for the purposes of computing the demand bound function. The most basic demand tuple abstraction of a path $\pi = (\pi_1, \ldots, \pi_m)$ is simply a triple $\langle e(\pi), d(\pi), \pi_m \rangle$ with the execution demand, deadline and final vertex of the path. The same demand tuple would abstract all paths that share these three properties. The demand tuples can be used instead of concrete paths for traversing the task graph by extending them with more vertices. If (u, v) is an edge of the graph and $\langle e, d, u \rangle$ a demand tuple for path π, then $\langle e', d', v \rangle$ with $e' = e + e(v)$ and $d' = d - d(u) + p(u, v) + d(v)$ is a demand tuple for π extended by v.

The problem of finding the maximum $e(\pi)$ for any path π with $d(\pi) \leqslant \ell$ can then be transformed into finding $\max\{e \mid \langle e, d, v \rangle$ demand tuple with $d \leqslant \ell\}$. To generate all demand tuples required to compute the demand bound function for all relevant values of the interval length ℓ, we first find an upper bound $\ell_{\max}$ on the values of ℓ that must be considered for schedulability. We then create demand tuples $\langle e(v), d(v), v \rangle$ for all vertices (or 0-length paths) v, and then iteratively extend each demand tuple $\langle e, d, v \rangle$ with more vertices as explained above, as long as $d \leqslant \ell_{\max}$. Duplicate demand tuples can be discarded on the fly, and other optimizations can be applied as well [1]. It was shown in [1] that a pseudo-polynomial $\ell_{\max}$ can be found assuming that the utilization of the task set is bounded by a constant strictly smaller than 1. There are therefore at most pseudo-polynomially many demand tuples to consider, and the process of computing the demand bound function is of pseudo-polynomial time complexity.

2) Demand bound function computation for MS-DRT: The above method for computing demand bound functions can be used for finding the intra-mode demand bound functions $\mathrm{dbf}_{\mu_j}(\tau, \ell)$ for MS-DRT tasks τ. The initial demand tuples $\langle e(v), d(v), v \rangle$ are created from vertices $v \in V(\tau)$ with $\mu(v) = \mu_j$, and the tuples are extended using edges from $E_{\mathrm{cf}}(\tau)$. In fact, one can apply the demand bound function computation for DRT tasks directly on the subgraph $(V(\mu_j, \tau), E_{\mathrm{cf}}(\tau))$, where $V(\mu_j, \tau) \stackrel{\text{def}}{=} \{v \in V(\tau) \mid \mu(v) = \mu_j\}$. Computing a bound $\ell_{\max}$ on the values of ℓ that must be considered for mode μ_j in Proposition IV.1 can be done exactly as in [1], using the set $\{(V(\mu_j, \tau), E_{\mathrm{cf}}(\tau)) \mid \tau \in T\}$ as the set of DRT tasks.

It is more difficult to compute the demand bound functions $\mathrm{dbf}_{\mu_i \to \mu_j}(\tau, \ell)$ for time intervals starting with a mode switch, as these must account for possible carry-over jobs from the previous mode. To achieve this, we adapt the ideas from [6], [4] for bounding the demand of carry-over jobs for mixed-criticality sporadic tasks. Consider a task τ that switches from mode μ_i to μ_j through an edge $(u, v) \in E_{\mathrm{ms}}(\tau)$, as illustrated in Figure 5.

We will assume that mode μ_i is schedulable when computing $\mathrm{dbf}_{\mu_i \to \mu_j}(\tau, \ell)$, i.e., that all deadlines are met in μ_i. By

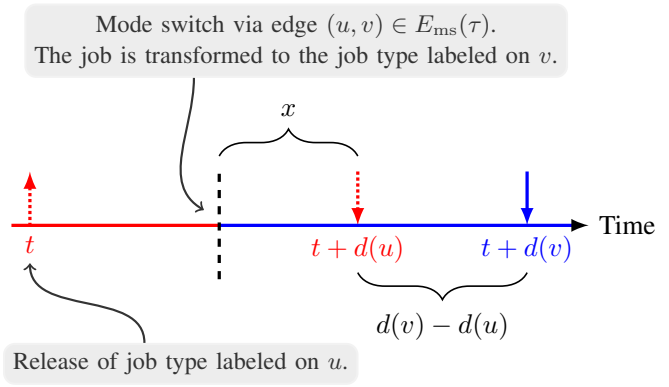


Fig. 5. Task τ switches from μ_i to μ_j trough an edge $(u, v) \in E_{\text{ms}}(\tau)$. The parameters of the active job are updated with the parameters of the job type labeled on vertex v. The job's remaining execution-time budget in μ_i is at most x time units, assuming that μ_i is schedulable.

reasoning similar to that in [4], this is a safe assumption for the purposes of finding demand bound functions to be used for schedulability analysis as in Proposition IV.1. With this assumption, we can conclude that if the mode switch happens x time units before the carry-over job's absolute deadline in μ_i, the job can have at most x time units left of its execution-time budget in μ_i at that time point. The job's remaining execution time budget as it enters mode μ_j is then at most $e(x, u, v) \stackrel{\text{def}}{=} \min(x, e(u)) + e(v) - e(u)$ time units. The length of the time interval until its new absolute deadline in μ_j is $d(x, u, v) \stackrel{\text{def}}{=} x + d(v) - d(u)$ time units. For the purposes of computing the demand bound function, we consider the carry-over job in μ_j as a new job released at the time of the mode switch, with execution-time budget and deadline as above. Such a job, if it exists, must be the first job to contribute demand to a time interval starting at a mode switch. A straightforward approach to computing $\text{dbf}_{\mu_i \to \mu_j}(\tau, \ell)$ is then to create the initial demand tuples

$$\langle e(x, u, v), d(x, u, v), v \rangle$$

for all $(u, v) \in E_{\text{ms}}(\tau)$ and $x \in [0, e(u)]$ such that $\mu(u) = \mu_i$ and $\mu(v) = \mu_j$. Note, however, that $x = e(u)$ corresponds to the "worst-case" demand in the sense that a smaller x leads to $e(x, u, v)$ decreasing by the same amount as $d(x, u, v)$. If the demand of the entire task set can be too high for some interval length ℓ_1 with $x \leqslant e(u)$, then it can also be too high for another interval length $\ell_2 \geqslant \ell_1$ when $x = e(u)$.[3] It is therefore enough for our purposes to create the above demand tuples with $x = e(u)$.

If the task has no active job at the time of the switch from μ_i to μ_j, then the first job to contribute demand in μ_j must be of a job type labeled on a vertex w such that $(v, w) \in E_{\text{cf}}(\tau)$ and $(u, v) \in E_{\text{ms}}(\tau)$ for some u, v with $\mu(u) = \mu_i$ and $\mu(v) = \mu_j$. For all such vertices w, we need to create demand tuples $\langle e(w), d(w), w \rangle$. These demand tuples, together with the ones for the carry-over jobs above, are all that are needed for safely abstracting the demand of all 0-length paths that start at a switch from μ_i to μ_j. These initial demand tuples can then be extended with additional vertices in the same manner as for regular DRT tasks, and ultimately be used to construct a $\text{dbf}_{\mu_i \to \mu_j}(\tau, \ell)$ for use in Proposition IV.1.

[3] This reasoning only holds under the assumption of a unit-speed dedicated processor as in Proposition IV.1. If we instead use some supply bound function (that is piecewise-linear between integer points) as a model of the computing platform, we might have to create demand tuples for all integers $x \in \{0, \ldots, e(u)\}$.

V. Conclusions

We have presented MS-DRT, a task model that supports the modeling of complex arrival and synchronization patterns with state-based mode changes. The mode switching protocol is of a mixed-criticality style, implying that MS-DRT generalizes both previous graph-based and mixed-criticality (sporadic) task models. MS-DRT also enables the modeling of many types of systems that fall outside of what is usually considered for mixed-criticality scheduling, some examples of such systems were shown in Section III. We have outlined how EDF schedulability analysis for MS-DRT can be performed by combining ideas from previous methods that use demand bound functions. We believe that MS-DRT offers a type of expressiveness not seen in commonly used models of mode-switching systems. At the same time, it offers the possibility of schedulability analysis that is significantly more efficient than for powerful timing models such as timed automata [12].

The schedulability analysis for mixed-criticality sporadic tasks that we adapted for MS-DRT can be greatly improved by a parameter-tuning preprocessing procedure [6], [4]. This tuning artificially decreases some of the relative deadlines in order to shift demand between the demand bound functions of different modes. A similar procedure can be applied to MS-DRT tasks, although the process of tuning is considerably more involved. As future work we plan to tackle this challenge, as well as to work out the remainder of the schedulability analysis in more technical detail.

References

[1] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "The digraph real-time task model," in *RTAS*, 2011, pp. 71–80.
[2] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *RTSS*, 2007, pp. 239–243.
[3] S. Baruah, "Certification-cognizant scheduling of tasks with pessimistic frequency specification," in *SIES*, 2012, pp. 31–38.
[4] P. Ekberg and W. Yi, "Bounding and shaping the demand of generalized mixed-criticality sporadic task systems," *Real-Time Systems*, 2013.
[5] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231 – 274, 1987.
[6] P. Ekberg and W. Yi, "Bounding and shaping the demand of mixed-criticality sporadic tasks," in *ECRTS*, 2012, pp. 135–144.
[7] S. Baruah, K., V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *IEEE Transactions on Computers*, Jul. 2011.
[8] H. Li and S. Baruah, "An algorithm for scheduling certifiable mixed-criticality sporadic task systems," in *RTSS*, 2010, pp. 183 –192.
[9] S. Baruah, V. Bonifaci, G. D'Angelo, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie, "Mixed-criticality scheduling of sporadic task systems," in *ESA*, 2011, pp. 555–566.
[10] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *RTSS*, 2011, pp. 34 –43.
[11] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Systems*, vol. 17, pp. 5–22, 1999.
[12] E. Fersman, P. Krcal, P. Pettersson, and W. Yi, "Task automata: Schedulability, decidability and undecidability," *Information and Computation*, vol. 205, no. 8, pp. 1149 – 1172, 2007.

Time-Triggered Mixed-Critical Scheduler

Dario Socci, Peter Poplavko, Saddek Bensalem and Marius Bozga
UJF-Grenoble 1
CNRS VERIMAG UMR 5104,
Grenoble, F-38041, France
{*Dario.Socci* | *Petro.Poplavko* | *Saddek.Bensalem* | *Marius.Bozga*}*@imag.fr*

***Abstract*—Modern safety-critical systems, such as avionics, tend to be mixed-critical, because integration of different tasks with different assurance requirements can effectively reduce their costs. Scheduling is one of the main challenges of such systems. In this work we show that a generalization of the Time Triggered (TT) scheduling paradigm, Single Time Table per Mode (STTM) dominates other approaches like Fixed Priority or Fixed Priority per Mode (FPM). We also propose an algorithm to transform any FPM priority assignment to an equivalent set of STTM tables.**

I. Introduction

Advances in technology is leading towards an increasing trend in integration of multiple functionalities on a single chip. Integration is an effective way of reducing cost and power consumption of embedded systems. On the other hand, for safety-critical domains, such as avionics, this is leading to the integration of tasks with significantly *asymmetric* safety requirements on a single assembly of processing resources.

Such system have called into existence a special Mixed-Critical System (MCS) scheduling theory, that has been developed at least since 2007 [1]. This theory treats the asymmetric safety requirements by adequate scheduling methods, which leads to much more efficient resource usage compared to classical scheduling approaches [2]. In particular, MCS-aware scheduling methodologies were demonstrated in [3] to significantly outperform traditional approaches such as reservation-based techniques.

A major branch of the MCS scheduling theory is the *certification-cognizant* mixed-critical scheduling, which assumes that all tasks are hard real-time and supports the certification prescribed by safety-critical standards such as $DO-178B$ [4]. Although this approach follows these prescriptions in a rather simple pragmatic way, it faces NP-complete problems even under basic assumptions [3].

Following a significant volume of previous MC scheduling work (*e.g.,* [3], [5], [6], [7]) in this paper we consider the basic problem of single-core scheduling for a finite set of jobs whose exact arrival times are known *a priori*. As stated in [6], this assumption applies without restrictions when generating schedules for *time-triggered architecture*, in this case one can just apply a finite job algorithm, like the one presented in this paper, to a hyperperiod of periodic tasks. We also restrict ourselves to dual-critical problems. This restriction is often assumed in literature ([5], [6], [7], [8], [9]). Dual-critical system are also of practical interest, such applications as *Unmanned aerial vehicles* (UAVs) assume two criticality levels: *safety critical* and *mission critical*.

The Own-Criticality Based Priority (OCBP) [5] is theoretically the best among all *Fixed Priority per job* (FP) scheduling algorithms for MCS. Recent extensions of the fixed job priority policy [8], [9] perform a switch between different priority tables for different modes. This Fixed Priority per Mode per job (FPM) policy can lead to better results due to their higher flexibility. In particular Mixed-Critical Earliest Deadline First (MCEDF) [10] has been proven to dominate OCBP, and hence FP scheduling, for dual-critical problems.

In [6] Baruah *et al.* propose a Time Triggered (TT) version of OCBP. This scheduling algorithm uses one static table per criticality mode. We will call this approach Single Time Table per Mode (STTM). The Time Triggered algorithms are important because they are easy to certify, since the time intervals in which each job executes are statically known *a priori*, while in an approach like FPM the jobs can interact in different ways depending on the execution times. The only unknown variable in STTM scheduler is the time when a switch will occur, but there are only a small number[1] of precomputed instants of time where this can happen, while in FPM these are infinite. Thus, even if STTM is a dynamic scheduler, all the possible executions can be easily enumerated, making certification easier. Also some commercial systems implement TT as default scheduling mechanism. In general, it is NP-complete problem to decide whether optimal scheduling policy (OPT) exists.

This work focuses on STTM algorithms, giving two main contributions: we prove that STTM approach dominates FPM and we give an algorithm that allows to transform an FPM priority assignment into a set of STTM tables. The following gives a relation between the sets of schedulable instances for dual-critical problems:

$$FP \overset{1}{\subsetneq} MCEDF \overset{2}{\subsetneq} FPM \overset{3}{\subsetneq} STTM \subseteq OPT \qquad (1)$$

Inclusion 1 is proved by dominance of MCEDF over OCBP [10]. Next, Inclusion 2 is true by definition of MCEDF. We can easily prove that the inclusion is strict under the hypothesis that $P \neq NP$. In fact under the restrictive hypothesis that all arrival times are equal to zero, we have that FPM is optimal, but the problem remains NP-complete even under this assumption [3]. If we assume by contradiction that MCEDF=FPM, then MCEDF could solve NP-complete problems in polynomial time. Example A.1 shows an instance that is FPM-schedulable but not MCEDF-schedulable.

The research leading to these results has received funding from **CERTAINTY** – European Community's Seventh Framework Programme [FP7/2007-2013], grant agreement no. 288175.

[1] equal to the number of HI jobs in a dual-critical system

Inclusion 3 is the main contribution of this paper. We will prove in Section IV that there exists an algorithm that can transform any FPM priority assignment into STTM tables. The strictness of the inclusion is shown by Example IV.1. Note that since the FPM policy is completely defined by a finite set of *basic* scenarios [3], such scenarios could be used as tables for a TT-like scheduling. This is theoretically feasible, but it has little practical interest, since it would potentially require a huge number of tables, whereas we require only two tables. Hence in this paper we actually propose a TT extension for MCEDF or any other FPM policy.

II. Background

Consider a set of hard real-time jobs having different *levels of criticality*. It is common in literature to model different criticality requirements by giving different worst-case execution times (WCETs) for the same job. In *dual-criticality* systems we have the highly level, denoted as 'HI', and the low critical (normal) level, denoted as 'LO'. Every job gets a pair of WCET values: the LO WCET and the HI WCET. One important remark is that both HI and LO jobs are hard real-time, so both *must* terminate their executions before the deadlines. But only HI jobs undergo certification. This means that the designer is confident that the jobs will never exceed their LO WCET, calculated by exhaustive measurements and adding some practical margin. However, it is required to prove to the certification authorities that the HI jobs will meet the deadlines even under the unlikely event that some jobs would execute at their HI WCET, calculated by more safe and pessimistic formal WCET estimation tools, required for certification.

A. Formalism

In a dual-criticality MCS, 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
- $A_j \in \mathbb{Q}$ is the arrival time, $A_j \geq 0$
- $D_j \in \mathbb{Q}_+$ is the deadline, $D_j > A_j$
- $\chi_j \in \{\text{LO}, \text{HI}\}$ is job's criticality level
- $C_j \in \mathbb{Q}_+^2$ 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 the jobs with the same parameters. We assume that $C_j(\text{LO}) \leq C_j(\text{HI})$. We also assume that the LO-criticality 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* $\mathbf{J}$ of the MC-scheduling problem is a set of K jobs. A *scenario* of an instance $\mathbf{J}$ is a vector of execution times of all jobs: $(c_1, c_2, \ldots, c_K)$. If at least one c_j exceeds $C_j(\text{HI})$, the scenario is called *erroneous*. The *criticality of scenario* $(c_1, c_2, \ldots, c_K)$ is the least critical χ such that $\forall j\ ,\ c_j \leq C_j(\chi)$. A scenario is *basic* if for each j either $c_j = C_j(\text{LO})$ or $c_j = C_j(\text{HI})$.

A (preemptive) schedule is a mapping from physical time to $\mathbf{J} \cup \{\perp\}$, where $\perp$ denotes no job. Every job should start at time A_j or later and run for no more than $C_j(\text{HI})$ time units. The online state of a run-time scheduler at every time instance consists of the set of terminated jobs, the set of *ready jobs*, *i.e.,* jobs that have arrived in the past and did not terminate yet, the progress of ready jobs, *i.e.,* how much each of them has executed so far, 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 schedule is *feasible* if the following conditions are met:

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 then its LO WCET, than all critical (HI) jobs must terminate before their deadline, whereas non-critical (LO) jobs may be even dropped.*

An instance $\mathbf{J}$ is *clairvoyantly schedulable* if for each non-erroneous scenario, when it is known in advance (hence *clairvoyantly*), one can specify a feasible schedule. By default, the scheduling is non-clairvoyant.

Based on the online state, a *scheduling policy* deterministically decides which ready job is scheduled at every time instant. A scheduling policy is *optimal* (or *correct*) for the given instance $\mathbf{J}$ if for each non-erroneous scenario it generates a feasible schedule. We assume without loss of generality that the scheduling policies are *monotonic per scenario*, which means one can check their optimality by simulating for all basic scenarios [3]. A *mode-switched* scheduling policy uses χ_{mode} in the scheduling decisions, *e.g.,* to drop the LO jobs, otherwise it is *mode-ignorant*. An instance $\mathbf{J}$ is *MC-schedulable* if there exists an optimal scheduling policy for it. A *fixed-priority* scheduling policy is a mode-ignorant monotonic policy that can be defined by a priority table PT, which is a K-sized vector specifying all jobs (or, optionally, their indexes) 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 selects the highest-priority job in PT. If a scheduling policy cannot be defined by a static priority table, it is called *dynamic-priority*.

A Time-Triggered (TT) table is a static, pre-computed table that defines at every instant of time which job must be scheduled. We define a Single Time Table per Mode (STTM) scheduling as an extension of TT scheduling that associates to each mode a single TT table.

In this paper, we consider the construction of STTM table starting from *fixed priority per mode* (FPM) policy, that is a fixed-priority scheduling that has a different PT for each mode. We assume that the scheduler is preemptive. For the given job set $\mathbf{J}$, this policy assumes two fixed-priority tables PT_{LO} before the mode switch and PT_{HI} after the mode switch. We assume that in the HI mode the LO jobs are dropped and hence excluded from PT_{HI}[2].

The basic scenario **LO** is the scenario where all jobs execute for time $C(\text{LO})$. Under the FPM policy, the basic scenario **HI-***J* is the scenario where job J is the first job that switches into HI mode after having executed for time $C(\text{LO})$.

[2]This assumption is legal according to Condition 2 and can only improve the schedulability

After the switch, job J and all non terminated HI jobs execute for time $C(\text{HI})$.

If we simulate the FPM policy for **LO** and all **HI-*J*** scenarios, we obtain function E^{LO} and E^{HI-J}, defined as $E^{LO|HI-J} : Time \rightarrow \{\perp\} \cup \mathbf{J}$ that specify for every time instance the job that runs at given time instance or $\perp$ when the processor is idle, when one job preempts another, or when a job starts/terminates. Note that this leads to *open* intervals of job activity and closed or single-point intervals of idle processor. These functions could be used for time-triggered scheduling, where we start in **LO** table defined by E^{LO} and switch to **HI-*J*** table defined by E^{HI-J} whenever a given HI job J switches to HI mode. But this would require an individual table per HI job, which is of little practical value.

III. Transformation Algorithm

Our goal is to obtain one single table **HI*** for the switch to the HI mode by *any* HI job. In this section we will show how to build such a table, while in Section IV we will show its correctness. We propose a method to generate this table by simulating fixed-priority for HI jobs with $C(\text{HI})$ times using priority table PT_{HI} and assuming that a HI job can be disabled at any time when all three *enabling* rules defined below are false. Whenever a non-terminated HI job is (temporarily) disabled, a lower-priority HI job can execute (priority inversion).

Before we give the rules, let us give some supplementary definitions.

Let $T_j^{LO}(t)$ (resp. $T_j^{HI*}(t)$) be the cumulative execution progress of job J_j by time t in table **LO** (resp. **HI***). We call a HI job that has executed for more than its $C(\text{LO})$ a *switched job*. We say that such a job switches at time t if $T_j^{LO}(t) = C_{LO}$. It is *non-switched* otherwise.

The method to generate **HI*** is as follows:

- at any time t, we execute the highest priority (according to PT_{HI}) *enabled* HI job
- a job J_j is *enabled* at time t if:
  - the job has arrived: $t > A_j$
  - the job has not yet executed for its HI WCET: $T_j^{HI*}(t) < C_j(\text{HI})$
  - at least one of the following *rules* is true:

$$T_j^{LO}(t) = C_j(\text{LO}) \quad (2a)$$
$$T_j^{HI*}(t) < T_j^{LO}(t) \quad (2b)$$
$$T_j^{HI*}(t) = T_j^{LO}(t) \wedge E^{LO}(t) = j \quad (2c)$$

Informally, Rule (2a) allows switched jobs to run as soon as possible, while Rules (2b) and (2c) assure that a job will not run in **HI*** for more time than in **LO** before the switch.

Example III.1. *Let us consider the following instance as an example:*

Job	*A*	*D*	χ	*C(LO)*	*C(HI)*
1	*0*	*12*	*HI*	*3*	*5*
2	*6*	*11*	*HI*	*2*	*4*
3	*7*	*8*	*LO*	*1*	*1*
4	*1*	*4*	*HI*	*1*	*2*

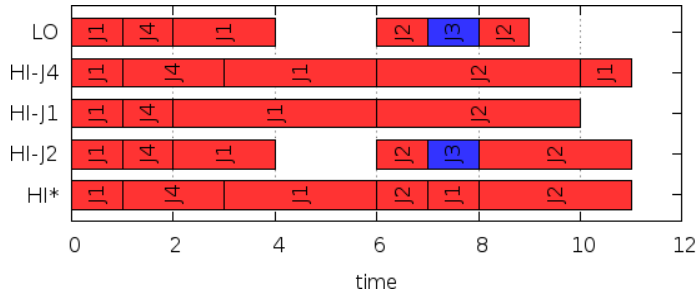


Fig. 1. Basic scenarios and TT tables

and assume the following FPM priority assignment[3]*:*

$$PT_{LO} = J_4 \prec J_1 \prec J_2 \prec J_3$$
$$PT_{HI} = J_4 \prec J_1 \prec J_2$$

*Figure 1 presents all the basic scenarios and the table **HI***. Consider this table. At time 0, only J_1 has arrived, and it is enabled by Rule (2c). At time 1, J_4 arrives, it has higher priority then J_1 and it is enabled by Rule (2c), so it is chosen by the algorithm to be executed. At time 2 for job J_4 Rule (2c) will be false, but Rule (2a) will become true, so we will continue execute it until time 3. At time 3 J_4 will terminate, so J_1 will be enabled by Rule (2b) until time 5 and by Rule (2a) from 5 on. So J_1 will continue its execution till time 6, when J_2 arrives. J_2 is enabled by Rule (2c), and it has higher priority than J_1, so it will be executed until time 7. At this instant Rule (2c) becomes false for J_2, disabling it. So we execute J_1. At time 8 J_1 terminates and J_2 is enabled by Rule (2c). At time 9 Rule (2c) is false for J_2, while Rule (2a) becomes true. So J_2 continues its execution until time 11, when it terminates.*

*It is easy to verify the correctness of TT scheduling that uses **LO** and **HI*** as tables. In fact in table **LO** all the jobs meet the deadline. When there is a switch, at time t, from **LO** to **HI***, all HI job J_j must have from time t a quantity of time reserved for them in **HI*** equal to $C_j(HI) - T_j^{LO}(t)$. In our example, if there is a switch in the **LO** table at time 2, caused by job J_4, then J_1, J_4 and J_2 will have enough remaining time reserved in **HI*** (respectively $4 = C_1(HI) - T_1^{LO}(2) = 5 - 1$, $1 = C_4(HI) - 1$ and $4 = C_2(HI) - 0$), and will terminate before their deadlines. In this case we will drop job J_3, since we do not care about LO jobs when in HI mode. Similarly, in the case of a switch at time 4, caused by J_1, then J_1 and J_2 will have respectively $3 = C_1(HI) - 2$ and $4 = C_2(HI) - 0$. Note that in this case J_1 will have one time unit more then it actually needs. Finally, if there will be a switch at time 9, caused by job J_2, this job will have 2 other time units, terminating at time 11, meeting its deadline.*

IV. Proof of Dominance

We will prove in this section the Inclusion 3 of Equation (1). At the same time we also provide an algorithm to construct STTM tables from FPM priority tables, generated by an algorithm such as MCEDF.

The following is the main claim of this paper:

[3]Can be computed using MCEDF [10]

Theorem IV.1. *If the FPM policy leads to a feasible schedule, then a switched time triggered schedule that uses **LO** and **HI*** as, respectively, LO-mode and HI-mode table, is a feasible schedule as well.*

To prove it, we first show that:

Lemma IV.2. *If at any time we switch from **LO** to **HI***, then all the unterminated jobs will have enough time reserved in **HI*** to terminate their work.*

First, let us comment that, according to our rules to construct **HI***, no HI jobs get disabled forever because eventually Rule (2a) becomes true, since all LO jobs eventually terminate. Thus, all HI jobs get a total time $C(\text{HI})$ reserved in **HI***. Consequently, if a job switches at time t, then this and any other job is guaranteed to get $C(\text{HI}) - T_j^{HI*}(t)$, but needs to get at least $C(\text{HI}) - T_j^{LO}(t)$.

Therefore the lemma can be equivalently stated as follows:

*no non-switched HI job makes more progress in **HI*** than in **LO**.*

Formally:

$$\forall t \ , \ T_j^{LO}(t) < C_j(\text{LO}) \Rightarrow T_j^{LO}(t) \geq T_j^{HI*}(t)$$

Proof of Lemma IV.2: At time $t = 0$ the lemma thesis is obviously true, and with progress of time it can be invalidated only during the time when a job is scheduled in **HI***. However, as long as $T_j^{LO}(t) < C_j(\text{LO})$ job J_j can only be scheduled when either (2b) or (2c) is true, but they both imply that we have $T_j^{LO}(t) \geq T_j^{HI*}(t)$. ■

Let $TT_J^{HI(LO|HI-J')}$ be the termination time of J in **HI*** (respectively, **LO**, **HI-***J'*).

Theorem IV.3. *Let J^{least} be the least priority job in PT_{HI}, then $\exists J' \ : \ TT_{J^{least}}^{HI*} \leq TT_{J^{least}}^{HI-J'}$*

Let us first give some definitions and support lemmas. A *busy interval* in some table (be it **LO**, **HI-***J* or **HI*** table) is a maximal continuous interval of time where some jobs are enabled for execution, where for table **HI*** we apply special rules defined earlier which can disable a job temporarily. When such rules are not applied, the busy intervals are obviously *open* intervals, because they are composed of union of (intersecting) open intervals between arrival and termination of different jobs. We state without proof that even with the extra rules we defined earlier for **HI***, the busy intervals remain to be open intervals.

For convenience, we use the term 'busy interval' also for the set of jobs that are enabled at least once inside the busy interval, and denote it BI, *e.g.,* BI^{HI*} for busy intervals in **HI***. Note that for this table, unlike the other tables, it is not always so that the total interval duration is exactly equal to the total work of jobs in BI, because there are rules that can temporarily disable a job after its arrival and before its termination. Therefore, the total work of jobs in BI^{HI*} can exceed the length of the busy interval. This also means that a job may belong to several busy intervals of **HI***.

In between BI, there are closed, sometimes single-point, *idle intervals*. For **HI***, we would like to distinguish an idle interval as a *hole* if inside this interval there are HI jobs that have arrived and not yet terminated, and are disabled because neither of the rules (2a), (2b), (2c) is true. The idle intervals that are not holes, are called *empty* intervals, *i.e.,* those where the job queue is empty.

For instance in Figure 1 in **HI*** there are two busy intervals: (0,8) and (8,11), thus we have a hole of size 0 at time 8. This happens because we have that immediately before time 8 J_1 is enabled by Rule (2a) while J_2 is disabled. On the other hand, at time 8 J_1 is disabled (because it terminates) while J_2 is enabled by Rule (2c).

The following proposition is well-known for fixed-priority policies, but needs to be re-established because we added the rules that can disable jobs.

Lemma IV.4. *If J^{least} is the least priority job in PT_{HI} then it terminates at the end of some busy interval BI^{HI*}.*

Proof: Let us assume by contradiction that J^{least} terminates inside a busy interval at time t. This means that at time t there is another enabled job (by definition of busy interval). If that is so, then J^{least}, having the least priority, should not be running at time t. ■

Lemma IV.5. *Let $BI^{HI*} = (a, b)$ be a busy interval in **HI***. At time a, the set of non-terminated HI jobs is the same in tables **LO** and **HI***, and for all of them holds that at time a the cumulative execution progress in **LO** is the same as in **HI***.*

Proof: Consider time a. The lemma thesis is obvious for any job that did not arrive yet, so in the sequel we consider only those jobs that have arrived.

If a job J is non-terminated in **LO** then it is non-terminated in **HI*** as well by Lemma IV.2. In addition, by the same lemma we have:
(I) $T_J^{HI*}(a) \leq T_J^{LO}(a)$.

On the other hand, if job J is non-terminated in **HI*** then the fact that it is not enabled at time a (by lemma condition) implies that Rule (2a) is false and hence the job is non-terminated in **LO** as well. Combined with the earlier observations, we conclude that the sets of non-terminated jobs at time a in these two tables are equal. In addition, also Rule (2b) is false, which means:
(II) $T_J^{HI*}(a) \geq T_J^{LO}(a)$.
Combining (I) and (II) we have the equality of the cumulative times. ■

Corollary IV.6. *Let $BI^{HI*} = (a, b)$ be a busy interval in which some job switches. Let J_s be the first such job, and let t^s be the time at which the switch occurs.*

*Then during the interval (a, t^s) tables **HI***, **HI-**J_s and **LO** are identical*

Proof: Notice that **HI-**J_s and **LO** are equal by construction in $(0, t^s)$ and hence in (a, t^s) as well. Let us compare **LO** and **HI***. At time a the set of non terminated jobs in these two tables are equal. In interval (a, t^s) no job switched yet, therefore all the jobs that run in **HI*** should satisfy Rule (2c), which is due to the fact that the other two rules require a switch

to have occurred. As long as Rule (2c) holds, the **HI*** table replicates the **LO** table, and because it fills time interval (a, t_s) continuously, as $t_s \in BI^{HI*}$, we have proved our thesis.

■

Proof of Theorem IV.3: Let $BI^{HI*} = (a, b)$ be the busy interval in which J^{least} terminates. By Lemma IV.4, $TT^{HI*}_{J^{least}} = b$. By Lemma IV.5, job J^{least} is not yet switched at start of this interval, and since this job terminates at the end of BI^{HI*}, we know also that it switches inside this interval as well, so Corollary IV.6 applies for this interval.

Let us assume that $BI^{HI*} = (a, b)$ is followed by an empty interval, *i.e.,* an idle interval which appears due to termination of all HI jobs that have arrived so far. Because in this case all the jobs of BI^{HI*} have terminated by time b, we have:

$$b = a + \sum_{j \in BI^{HI*}} \left(C_j(\text{HI}) - T_j^{HI*}(a)\right)$$

Let J_s be the first job to switch in BI^{HI*}, at time t^s. By Lemma IV.5 and Corollary IV.6, we have that the same jobs, with the same remaining execution time as in **HI*** will run from time a in **HI-**J_s before the switch and, by construction after the switch as well. Therefore $BI^{HI*} = BI^{HI-J_s}$ and J^{least}, being the least-priority job, will terminate at time b in both tables.

Let us now examine the other case, in which $BI^{HI*} = (a, b)$, the busy interval where J^{least} terminates, is followed by a hole, *i.e.,* the idle interval which appears because at time b the rules for table **HI*** have disabled the non-terminated jobs. Also in this case J^{least} by our hypothesis and Lemma IV.4 will terminate at time b, but in this case by construction not all jobs of BI^{HI*} terminate by time b:

$$b < a + \sum_{j \in BI^{HI*}} \left(C_j(\text{HI}) - T_j^{HI*}(a)\right) \tag{3}$$

Let J_s be the first job to switch in BI^{HI*}, at time t^s. Again by Lemma IV.5 and Corollary IV.6 we observe the same initial state and subsequent behavior in tables **HI*** and **HI-**J_s of all non-terminated HI jobs during the time interval $(a, t^s]$. So we conclude that all jobs of BI^{HI*} run in **HI-**J_s after time a continuously, at time a their total remaining work is equal to:

$$\sum_{j \in BI^{HI*}} \left(C_j(\text{HI}) - T_j^{HI*}(a)\right)$$

In line with equation (3), in order to complete this workload, table **HI-**J_s has to continue execution after time b. New jobs may arrive before the termination of the busy interval BI^{HI-J_s}. this busy interval executes all these jobs, J^{least} being the last one to terminate. So we have:

$$BI^{HI*} \subseteq BI^{HI-J_s}$$

and

$$TT^{HI-J_s}_{J^{least}} \geq a + \sum_{j \in BI^{HI*}} \left(C_j(\text{HI}) - T_j^{HI*}(a)\right) \tag{4}$$

Combining (3) and (4), and observing that $TT^{HI*}_{J^{least}} = b$, we have that also in this case in **HI-**J_s the least-priority job terminates no earlier than in **HI***. This completes the proof of Theorem IV.3. ■

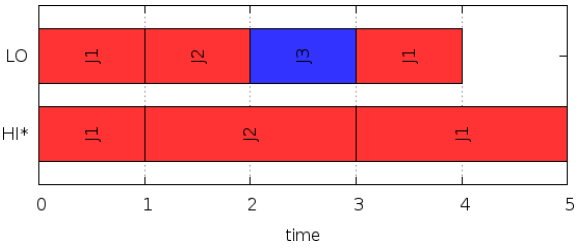


Fig. 2. The TT tables for the instance of Example IV.1

Proof of Theorem IV.1: From Lemma IV.2 we know that in any possible scenario all the HI jobs will have enough processor resource to terminate. The termination time of J^{least} is guaranteed to meet the deadline due to the hypothesis that it meets deadline in the FPM policy and Theorem IV.3. Now let us prove that also the HI jobs with higher priority in PT_{HI} meet their deadlines. Let $J^{\overline{least}}$ be the next least priority HI job after J^{least} in the PT_{HI} table. Let **J** be the currently examined problem instance and let $\overline{\mathbf{J}}$ be the instance obtained from **J** by reducing the criticality of J^{least} to LO. It is easy to show that the HI-mode table $\overline{\mathbf{HI}*}$ obtained for this new instance coincides with **HI*** except that the intervals where J^{least} is running are idled. So, $J^{\overline{least}}$ will terminate in **HI*** at the same time as in $\overline{\mathbf{HI}*}$, where by Theorem IV.3 applied to instance $\overline{\mathbf{J}}$ it will terminate no later than the latest termination under FPM policy. Obviously, also the latest termination of the FPM policy for job $J^{\overline{least}}$ is the same for both **J** and $\overline{\mathbf{J}}$. Because by our hypothesis this policy is feasible we conclude that $J^{\overline{least}}$ meets its deadline. Iterating this reasoning recursively, we argue that all HI jobs meet their deadline in **HI***, and thus we have our thesis. ■

Theorem IV.1 proves that FPM $\subseteq$ STTM. To prove the strictness of the inclusion, we give the following counter-example:

Example IV.1. *Let* $\mathbf{J}_d$ *be the following instance:*

Job	*A*	*D*	χ	*C(LO)*	*C(HI)*
1	*0*	*5*	*HI*	*2*	*3*
2	*1*	*3*	*HI*	*1*	*2*
3	*0*	*3*	*LO*	*1*	*1*

No FPM policy would schedule it. The only correct scheduling policy for $\mathbf{J}_d$ *is to execute* J_1 *for 1 time unit, then* J_2*. If* J_2 *terminates after 1 time unit, we execute* J_3 *and then* J_1 *again, otherwise we drop* J_3 *and execute* J_1*. It is easy to see that this is not an FPM schedule, as* J_1 *changes its priority* w.r.t. J_3 *in the LO scenario. This scheduling policy can be implemented using STTM tables as shown in Figure 2*

V. Conclusions and Future Work

In this paper we proposed a method to transform any FPM priority assignment into a set of STTM tables. This has a practical importance since safety critical systems are designed with a TT (also known as *static*) scheduler. A TT scheduler has a behavior that is completely precomputed. This makes certification of such system much easier. Although STTM is not static, it has a finite number of switches that can be trivially

checked the same way as TT. From a theoretical point of view, we proved that STTM dominates FPM.

In future work we plan to extend this algorithm for more than two levels of criticality. Also, it is necessary to investigate the mixed-critical scheduling of task graphs, where there are data dependencies between jobs.

REFERENCES

[1] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Real-Time Systems Symposium*, RTSS'07, pp. 239–243, IEEE, 2007.

[2] D. d. Niz, K. Lakshmanan, and R. Rajkumar, "On the scheduling of mixed-criticality real-time task sets," in *Real-Time Systems Symposium*, RTSS'09, pp. 291–300, IEEE, 2009.

[3] 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.

[4] L. A. Johnson, "DO-178B: Software considerations in airborne systems and equipment certification.," in *Radio Technical Commission for Aeronautics.*, RTCA, 1992.

[5] S. K. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium*, RTAS'10, pp. 13–22, IEEE, 2010.

[6] 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.

[7] T. Park and S. Kim, "Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems," in *Intern. Conf. on Embedded software*, EMSOFT '11, pp. 253–262, ACM, 2011.

[8] 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.

[9] P. Ekberg and W. Yi, "Bounding and shaping the demand of mixed-criticality sporadic tasks," in *Euromicro Conf. on Real-Time Systems*, ECRTS'12, pp. 145–154, IEEE, 2012.

[10] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Mixed critical earliest deadline first," in *Euromicro Conf. on Real-Time Systems*, ECRTS'13, pp. 93–102, IEEE, 2013.

APPENDIX

The following example proves that $MCEDF \subsetneq FPM$.

Example A.1. *Consider the following instance:*

Job	*A*	*D*	χ	$C(LO)$	$C(HI)$
1	*0*	*8*	*LO*	*5*	*5*
2	*0*	*10*	*HI*	*2*	*3*
3	*0*	*11*	*HI*	*2*	*5*

applying MCEDF[4] *to it we will have the following FPM priority assignment:*

$$PT_{LO} = \quad J_2 \prec J_1 \prec J_3$$
$$PT_{HI} = \quad J_2 \prec J_3$$

It is easy to show that this priority assignment is not correct. In fact, if J_2 executes for $C_2(LO)$ and J_3 executes for $C_3(HI)$, then J_3 will terminate at time 12, missing its deadline.

[4]refer to [10] to know how to compute these priorities

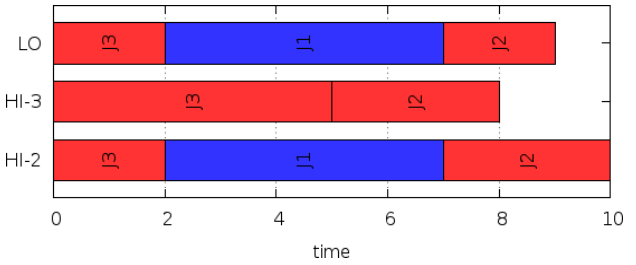


Fig. 3. Basic scenarios for Example A.1

On the other hand, the following FPM assignment:

$$PT_{LO} = \quad J_3 \prec J_1 \prec J_2$$
$$PT_{HI} = \quad J_3 \prec J_2$$

is correct. This can be checked on the charts of Figure 3, where the basic scenarios are reported.

Schedule Table Generation for Time-Triggered Mixed Criticality Systems

Jens Theis and Gerhard Fohler
Technische Universität Kaiserslautern, Germany

Sanjoy Baruah
The University of North Carolina, Chapel Hill, NC, USA.

***Abstract*—Recent research in real-time scheduling for mixed criticality systems has centered on an event-triggered/priority-driven approach to scheduling. Current practice in many safety-critical domains, however, favors a time-triggered approach.**

We present here an effective and flexible approach for applying mode changes for time-triggered systems to handle mixed criticality job sets.

It is based on a heuristic search algorithm for constructing schedule tables for the different criticalities: a change of criticality levels results in a change in the schedule table that is being used. The sequence of steps in case of a change of criticality level is known beforehand.

We present a search-tree based framework for our heuristic search, and derive a heuristic function for guiding the search that significantly reduces the search space for backtracking by swapping search decisions over several levels in the search tree.

I. Introduction

The common approach to design embedded real-time systems with safety critical requirements, subject to certification, has been complete spatial and temporal isolation between activities in the system (e.g., in the ARINC standard [1] for avionics). In many modern platforms, however, the impact on performance and resource utilization of such strict separation approaches can no longer be justified by certification efforts, even more so as over pessimistic assumptions are mandated.

In order to certify a system as being correct, the certification authorities (CAs) mandate certain assumptions about the worst-case behavior of the system during run-time; these assumptions, e.g., for execution time, are typically far more conservative than the assumptions that the system designer would use during the process of designing, implementing, and testing the system if subsequent certification were not required. However, while the CAs are only concerned with the correctness of the safety-critical part of the system, the system designer is responsible for ensuring that the entire system is working correctly, including the non-critical parts. Traditional scheduling criteria as deadlines, utilization, etc. have proved inadequate for accommodating these contrasting demands of low and high critical applications.

In the time-triggered (TT) paradigm [13] of real-time scheduling, activities in the system are triggered by the progression of time only. A schedule for the entire duration of a system's execution is constructed prior to run-time. The scheduling decision that is made at each instant during run-time is completely determined by examining this pre-computed schedule, represented, e.g., in a schedule table. The schedule tables typically used for TT systems are particularly easy to verify; hence, they have been popular in safety critical systems subject to certification.

In mixed criticality systems as described above, however, the inflexibility of TT scheduling poses additional challenges: tasks with different assumptions cannot be fit into a single schedule table or changed, should the need arise during system operations.

While current practice in many safety-critical application domains is centered on time-triggered (TT) scheduling, much recent research on scheduling for mixed criticality systems has focused on even-triggered/priority-based scheduling.

In [5], an effort was made to show that the results obtained by this mixed criticality research could indeed be applied to TT-scheduled systems. It advocated the creation of two different schedule tables, one based upon the system designers' job parameters and the other using the CAs' parameters. At run-time the system starts operation with the schedule table that was constructed assuming that the system designers' parameters are correct. If a violation of the designer assumptions is detected during run-time, the run-time dispatcher switches tables and henceforth begins using the schedule table that was constructed assuming that the CAs' parameters are the correct ones.

Challenges in the construction of these schedule tables include the need to build two matching schedule tables, which allow for feasible and consistent switching from one to the other at run-time while ensuring, e.g., that even during a switch, the computational demands of individual jobs that are active during the switch are met. Since jobs with two different WCET values, one for each criticality level (mode), have to be considered, standard mode change schedule-generation algorithms (such as the ones presented in [11], [12]) cannot be directly applied. The simple table-generation algorithm of [5] was based on identifying a common priority ordering of the jobs for both schedules, with the priority of each job being based on its criticality level. This priority ordering is inflexible, and the resulting resource utilization is less than may be achieved by dynamic scheduling approaches such as EDF.

We believe that [5] was to a large measure successful in the sense that (i) mixed criticality scheduling principles were transferred to the TT domain; (ii) a TT-based framework for implementing mixed criticality systems was designed; and (iii) proof-of-concept algorithms for generating the schedule tables needed by this framework were derived. However, TT scheduling typically uses very sophisticated search-based algo-

This work has been supported in part by the European project DREAMS under project No. 610640.

rithms for schedule table generation, thereby achieving good resource utilization in practice despite the poor worst case bounds that can be derived for pathological workload instances. Hence, a true integration of TT and mixed criticality scheduling requires that it must be shown that schedule tables can be generated that are far more resource-efficient than the ones generated by the method in [5].

In this paper, we present an algorithm for constructing the schedule tables and for run-time execution that results in far more efficient resource utilization and enhanced flexibility; in contrast to the highly simplified proof-of-concept tables obtained by the techniques of [5], these tables can be used for actual system implementation. We have developed a tree-based search algorithm based on a heuristic function called the *leeway* (described in Section V), and a sophisticated and innovative backtracking algorithm based on this heuristic, that we believe shows this.

Related work. It is beyond the scope of this paper to discuss all work on mixed criticality scheduling; instead, we refer the interested reader to the recent survey by Burns and Davis [9].

In [4], Baruah et al. proved that mixed criticality scheduling is NP-hard in the strong sense even for two criticality levels — this provides some justification for seeking heuristic approaches to schedule construction. Further, [4] considered two classes of mixed criticality scheduling algorithms. Reservation-based scheduling policy, e.g., the straightforward approach by assigning $WCET := max\,(WCET(\chi_i))$ for all criticality levels χ_i, is a very pessimistic approach. Depending on the difference between the minimum and maximum WCET of the criticality levels, huge differences between the reserved WCET and the WCET of the actual criticality level are possible. Hence, this leads to an under-utilization of the system. The advantage of this approach is low complexity schedulability test. Additionally [4] considered the class of priority-based scheduling policies. As an example, they used the "Audsley-approach" (see [2], [3]) of assigning priorities; this assignment of fixed priorities to jobs compromises resource utilization and reduces flexibility.

In [10], de Niz et al. presented a preemptive, priority based algorithm. In this algorithm, high criticality tasks are protected from interference by low criticality tasks. This Rate-Monotonic-based algorithm determines the point in time when the priority has to be increased such that the deadline can be met with the high criticality WCET.

Park and Kim presented in [15] an online algorithm based on EDF for scheduling mixed criticality jobs. Based on the deadlines of the jobs, they created intervals and calculated slacks for these intervals. An earlier completion of a high criticality job, i.e., actual execution is less than WCET, increases the "remaining slack". Further, the available time when all high criticality jobs are guaranteed with high criticality WCET is considered as so-called "empty slack". Low criticality jobs are only executed if "remaining slack" or "empty slack" is greater than zero.

Organization. The remainder of the paper is structured as follows: in Section II, we present terms and the basic task model. Section III shows the basic idea of handling mixed criticality jobs based on our time-triggered approach with mode changes. The allocation of jobs to modes, depending on their criticality levels, is shown in Section IV. In Section V, we show the algorithm to construct the time-triggered schedule tables. We discuss our backtracking heuristic and the consequences for the scheduling process in Section VI. The evaluation of our methods is shown in Section VII. Finally, Section VIII concludes the paper.

II. Terminology and Notation

In the following, we present the terms that we use in the rest of the paper. We assume a system with two criticalities: low (LO) and high (HI). Unless otherwise specified we will represent relative time values (e.g. WCETs) by using upper case variables, whereas lower case variables represent absolute time values (e.g. release times). The dual criticality jobs J_i with $i \in \{1, .., n\}$ are characterized by the 5-tuple $\langle \chi_i, r_i, d_i, C_i(\text{LO}), C_i(\text{HI})\rangle$, with $\chi_i \in \{\text{LO}, \text{HI}\}$: criticality level of job i, $r_i \in \mathbb{R}^+$: release time of job i, $d_i \in \mathbb{R}^+$: absolute deadline of job i with $d_i > r_i$, $C_i(\text{LO})$: LO criticality WCET (as estimated by system designer), and $C_i(\text{HI})$: HI criticality WCET (as estimated by certification authority).

For LO criticality jobs, we assume $C_i(\text{LO}) = C_i(\text{HI})$, whereas for HI criticality jobs $C_i(\text{LO}) \leq C_i(\text{HI})$. Note that the assumption that $C_i(\text{LO}) = C_i(\text{HI})$ for low criticality jobs supposes the presence of run-time mechanisms for monitoring the amount of time that a job has executed, and preventing a job from executing beyond an allocated "budget". Such mechanisms are commonly found in most real-time operating systems. A low criticality job would then be allocated an execution budget equal to its LO criticality WCET, its $C_i(\text{LO})$ value.

The demand $g(t_1, t_2)$ in an interval $[t_1, t_2]$ is defined as the amount of processing time requested by jobs that are activated in that interval. The demand calculation considers all jobs whose release time is after the beginning of the interval ($r_i \geq t_1$) and must be completed before the end of the interval ($d_i \leq t_2$). It is known that it is sufficient to check the intervals from zero to the latest deadline in the system [6].

III. TT Systems with Mode Changes

In this section, we state the requirements placed on the scheduling of mixed criticality job sets and how we can accommodate these requirements within the framework of a time-triggered system. HI criticality jobs are subject of certification under pessimistic assumptions of the CAs. Furthermore, the designer has to ensure that the entire job set is feasible under his less pessimistic assumptions. Time-triggered schedule tables allow to simplify the certification process by their complete determinism. A feasible schedule table represents a constructive proof of timing correctness. The disadvantage of schedule tables is their inflexibility. Baruah and Fohler showed in [5] that constructing one schedule table per criticality level can fulfill CAs' requirements. The challenge in the construction of the schedule tables is to meet the requirements of mixed

original jobs					split jobs				
job	release time	deadline	WCETs	criticality	job	release time	deadline	WCET	criticality
J_i	r_i	d_i	$C_i(\text{LO}) = C_i(\text{HI})$	LO	J_i	r_i	d_i	$C_i(\text{LO})$	LO
J_i	r_i	d_i	$C_i(\text{LO}) \leq C_i(\text{HI})$	HI	J_i^{LO}	r_i^{LO}	d_i^{LO}	$C_i^{\text{LO}}(\text{LO})$	HI
					J_i^{Δ}	r_i^{Δ}	d_i^{Δ}	$C_i^{\Delta}(\text{HI})$	HI

TABLE I
OVERVIEW OF JOB PARAMETERS

criticality jobs and guaranteeing CAs' requirements when switching between the two schedule tables. A possible solution for this can be accomplished via the mechanism of *mode changes*. By the use of mode change schedulers (e.g. [11]), we can accommodate for the afore-mentioned demands. [5] describes the requirements for a mode-change-based scheduler for certification-cognizant mixed criticality jobs as a proof of concept. Two modes are used, one for each criticality level, to accommodate for the requirements of LO and HI criticality behavior. In the LO criticality mode LO-*table*, correct system behavior for the entire job set is guaranteed, based on the system designer's assumptions. Assumptions of the CAs are incorporated into the HI criticality mode HI-*table*. Note that for the purposes of certification it is sufficient to guarantee the correct execution of only the HI criticality jobs, but these must be guaranteed based on the pessimistic assumptions about the WCET of the CAs. However, the table-generation technique presented in [5] includes all jobs, both the LO-criticality and the HI-criticality ones, in the HI-*table*; this leads to an under-utilization of system resources.

We will construct two schedule tables, one for LO and one for HI criticality mode, such that switching from LO-*table* to HI-*table* is possible at every point in time, so-called *switch through property* [11]. At run-time, the system starts execution of the LO criticality schedule table. Violation of the system designer's assumptions (exceeding $C_i(\text{LO})$ of a HI criticality job) leads to directly switching to the HI criticality mode HI-*table*. In case such a switch occurs, we must guarantee that no HI criticality job misses its deadline. Furthermore, the CAs' pessimistic assumptions must be guaranteed for HI criticality jobs. According to the problem statement in [5], switching back to the LO criticality mode is not specified.

IV. Allocation of Jobs to Modes

Our method works on jobs, hence, it is also able to handle task sets as each individual task is composed of a sequence of recurring jobs. In this section, we describe how we split the jobs to separate the WCET of the LO criticality level and the additionally needed WCET in HI criticality case. After doing so, each resulting job is specified only with a single WCET.

LO criticality jobs J_i, which only are present in LO-*table*, are not split and the WCET is set to $C_i(\text{LO})$. The remaining parameters of these jobs remain unchanged. Each HI criticality job J_i is split into a job J_i^{LO}, which is the portion present in both tables and a job J_i^{Δ}, which is the portion that is additionally needed in HI criticality case. The calculation of the WCETs, which we will use as input for our scheduler, are shown in equations (1) - (3).

$$J_i : C_i(\text{LO}) \leftarrow C_i(\text{LO}),\ i \in \{1,..,n\} \wedge \chi_i = \text{LO} \quad (1)$$

$$J_i^{\text{LO}} : C_i^{\text{LO}}(\text{LO}) \leftarrow C_i(\text{LO}),\ i \in \{1,..,n\} \wedge \chi_i = \text{HI} \quad (2)$$

$$J_i^{\Delta} : C_i^{\Delta}(\text{HI}) \leftarrow C_i(\text{HI}) - C_i(\text{LO}),\ i \in \{1,..,n\} \wedge \chi_i = \text{HI} \quad (3)$$

For the split jobs we derive now new parameters based on the original parameters of the HI criticality job. The release time r_i^{LO} of J_i^{LO} is equal to original release time r_i. The deadline of J_i^{LO} must be early enough such that there remains enough time to schedule the additionally needed WCET in the HI criticality case. As a result, the deadline of J_i^{LO} is set to $d_i^{\text{LO}} := d_i - C_i^{\Delta}(\text{HI}) = d_i - (C_i(\text{HI}) - C_i(\text{LO}))$.
The earliest release time of J_i^{Δ} is possible when its corresponding job J_i^{LO} is scheduled directly at the beginning of the execution window. Hence, we set the release time of a job J_i^{Δ} to $r_i^{\Delta} := r_i + C_i^{\text{LO}}(\text{LO}) = r_i + C_i(\text{LO})$. The deadline of J_i^{Δ} is equal to the deadline of its corresponding original job $d_i^{\Delta} := d_i$. This results in maximum slack for both J_i^{LO} and J_i^{Δ}. To avoid that J_i^{Δ} is scheduled before J_i^{LO}, ***we add a precedence constraint*** between them: $J_i^{\text{LO}} \prec J_i^{\Delta}$. As a consequence, we can guarantee with J_i^{Δ} that in HI criticality case the additionally needed WCET is scheduled. The two resulting jobs (J_i^{LO} and J_i^{Δ}) of a split HI criticality job (J_i) keep their criticality level HI. Table I gives an overview of the original and the split jobs' parameters.

The LO criticality table LO-*table* contains all the jobs in the set $S(\text{LO})$ while the HI criticality table HI-*table* contains all the jobs in the set $S(\text{HI})$ with:

$$S(\text{LO}) = \{J_i, J_k^{\text{LO}}\},\quad (i \in \{1,..,n\} \wedge \chi_i = \text{LO}) \wedge (k \in \{1,..,n\} \wedge \chi_k = \text{HI}) \quad (4a)$$

$$S(\text{HI}) = \{J_i^{\text{LO}}, J_i^{\Delta}\},\quad (i \in \{1,..,n\} \wedge \chi_i = \text{HI}) \quad (4b)$$

V. Construction of the Scheduling Table

The schedule table is divided into slots, which means that this is the granularity of our scheduler, hence, it is preemptive at slot borders. Construction of the scheduling tables is done concurrently for both tables, i.e. first slot i – which refers to the time interval $[i, i+1)$ – is scheduled in both tables (first LO-*table*, then HI-*table*), before in the next step slot $(i+1)$ is scheduled. The length of the schedule table is determined by the last deadline of the job set or the hyper-period in case of a periodic task set. Scheduling decisions are represented by a *search tree* which is based on iterative deepening [14]. Each scheduling decision for a slot in both tables (i.e. a pair of selected jobs) is represented by an edge in the search tree. Based on the history of decisions a node represents a possible partial schedule of

both tables at a given point in time. In each node of a path, several scheduling decisions can be taken, leading to different (partial) schedules. All combinations of possible scheduling decisions form the complete search tree. Leaf nodes represent feasible and infeasible complete schedules. The selection of jobs in both tables uses a heuristic which is based on EDF and the criticality levels. If a scheduling decision leads to infeasible schedule tables, we use backtracking to search for another schedule table. In "classic" backtracking, an exhaustive search checks all possible decisions in a search tree which is extremely complex. We use a heuristic for backtracking which is based on the demand of HI criticality jobs to reduce the complexity of backtracking.

A. Low Criticality Scheduling Decisions

In the LO criticality mode LO-*table*, we select the job of the set $S(\text{LO})$ with the earliest deadline which is ready.

We introduce the concept of the *leeway* $\delta(s_c)$ of the current slot s_c, see equation (5), as a heuristic function for our backtracking mechanism. The calculation of the leeway depends on the criticality level of the selected job: for LO criticality jobs J_i, the leeway represents the difference between the deadline of the selected job and the current time (i.e. end of current slot). For HI criticality jobs J_i^{LO}, the leeway represents the difference between the deadline of the selected job (J_i^{LO}) and the current point in time ***reduced*** by the remaining demand of all J_k^{Δ} with $k \in \{1, .., n\}$ which have to be scheduled in HI-*table* until the deadline of J_i^{Δ}. The demand $g^{\Delta}(t)$ at time t represents the demand in the interval $[0, t]$ accumulated by all jobs J_i^{Δ}. The function $g_{sched}^{\Delta}(t)$ keeps track of the already scheduled demand of HI criticality jobs J_i^{Δ}. If there is no job scheduled in a slot, we define the leeway to be infinity.

$$\delta(s_c) = \begin{cases} d_i - (s_c + 1) & \text{if } \chi_i = \text{LO} \\ [d_i - (s_c + 1)] & \\ \quad - \left[g^{\Delta}(d_i^{\Delta}) - g_{sched}^{\Delta}(s_c)\right] & \text{if } \chi_i = \text{HI} \\ \infty & \text{else} \end{cases} \tag{5}$$

Slot s_c in the HI criticality schedule table HI-*table* has not been scheduled yet, hence, $g_{sched}^{\Delta}(s_c)$ does not include the scheduled demand of HI criticality jobs J_i^{Δ} in the current slot in HI-*table*. For each slot in the mode LO-*table*, we calculate the leeway. A non-negative leeway means it can be possible to schedule remaining jobs J_i^{Δ} in the HI criticality mode. Thus, we continue the scheduling process by scheduling the current slot in mode HI-*table*. If the leeway is negative, then independent of succeeding scheduling decisions, all paths will lead to leaves representing infeasible schedules. As a consequence, we start backtracking based on our heuristic (see section VI). Based on the heuristic function (leeway), a preceding node is searched for backtracking, i.e., we change the scheduling decision for that slot.

B. High Criticality Scheduling Decisions

Based on the decision in LO-*table*, we select a job for mode HI-*table*. If the scheduled job in the current slot in mode LO-*table* has criticality level HI, i.e. J_i^{LO}, then we schedule the same job J_i^{LO} in the current slot in mode HI-*table*. If the scheduled job is a LO criticality job J_i (or no job is scheduled) then we select a HI criticality job J_k^{Δ} with fulfilled precedence constraints and earliest deadline. After scheduling a HI criticality job J_i^{Δ}, we increase the amount of already-scheduled demand $g_{sched}^{\Delta}(s_c)$ by one slot. After scheduling the current slot, we check whether a deadline miss of a HI criticality job J_i^{Δ}. occurred. In this case, the scheduling process is aborted. After scheduling the HI criticality mode HI-*table*, we continue with scheduling the next slot.

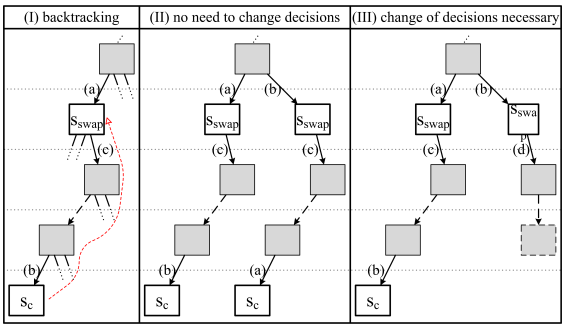


Fig. 1. Backtracking and consequences for the scheduling decisions

VI. Backtracking Heuristic

If the scheduler calculates a negative leeway for a slot, we start backtracking with our heuristic based on the leeway. In Figure 1 column (I), scheduling decision (b) for current slot s_c led to a negative leeway and hence, all succeeding scheduling decisions will yield infeasible schedule tables. As a consequence, we may skip the search for a feasible schedule in this part of the search tree. By this doing this, we save the time to check all succeeding decisions.

Based on the heuristic function (leeway), we look for a promising predecessor node to continue the scheduling process with a different scheduling decision for that node (see dotted arrow in Figure 1 column (I) from s_c to s_{swap}). We start in the current slot s_c and proceed upwards in the tree structure, checking based on the leeway for a slot s_{swap} at which we can swap the scheduling decision. The conditions for this swapping slot are: The swapping slot must be later than the release time of job i scheduled by decision (b). Furthermore, the leeway of a candidate for the swapping slot must be greater than or equal to the difference in number of slots between the current slot and the candidate for the swapping slot, i.e. $\delta(s_{swap}) \geq s_c - s_{swap}$. If these conditions are fulfilled, we can delay scheduling decision (a) for the swapping slot.

Once we have found a slot which fulfills the swapping conditions, we swap scheduling decisions (b) and (a), hence, decision (b) is now taken for s_{swap} and decision (a) for s_c (see Figure 1 column (II)). The scheduling decisions after the swapping slot – e.g. decision (c) – remain unchanged. After swapping of the decisions of the two slots, we have to recalculate the leeways of these slots.

By swapping scheduling decisions (a) and (b), it is possible that fulfilled precedence constraints are not fulfilled anymore and/or scheduled demand of HI criticality jobs J_i^{Δ} is changed. In this case, we cannot continue with scheduling decision (c)

after s_{swap} and we continue the scheduling process after s_{swap} with a possibly different decision (d) for the next slot (see Figure 1 column (III)). As a result, the current slot is set to the swapping slot and we continue the scheduling process from that slot.

Depending on the scheduled jobs in the current slot and the swapping slot, there are different cases which we have to consider when making swapping decisions. In the following, we present these cases and the consequences for the scheduling process. We refer to slots in mode LO-*table* by s^{LO} and slots in mode HI-*table* by s^{HI}.

Case 1: In both slots s_c^{LO} and s_{swap}^{LO}, a LO criticality job is scheduled. In s_{swap}^{HI}, no job is scheduled (Figure 2). We swap slots in LO criticality mode LO-*table* and update the leeway of s_c and s_{swap}. The swapping slot in the HI criticality mode remains unchanged. In a last step, we schedule the current slot in the HI criticality mode. Swapping of this case does not change any precedence constraints and this refers to Figure 1 column (II).

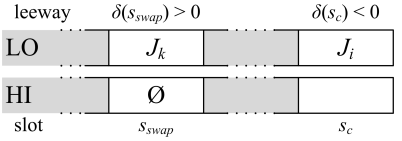


Fig. 2. Backtracking case 1: slots before backtracking

Case 2: In both slots s_c^{LO} and s_{swap}^{LO}, a LO criticality job is scheduled. In s_{swap}^{HI}, a HI criticality job J_m^{Δ} is scheduled (Figure 3). We swap slots in LO criticality mode LO-*table* and update the leeway of s_c and s_{swap}. Swapping two LO criticality jobs does not change precedence constraints, and hence, the scheduled job in s_{swap}^{HI} remains unchanged. In the last step, we schedule the current slot in the HI criticality mode. This case refers to Figure 1 column (II).

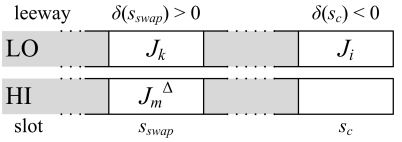


Fig. 3. Backtracking case 2: slots before backtracking

Case 3: In slot s_c^{LO}, a LO criticality job J_i and in slot s_{swap}^{LO}, a HI criticality job J_k^{LO} are scheduled. In s_{swap}^{HI}, the same HI criticality job J_k^{LO} as in s_{swap}^{LO} is scheduled (Figure 4).We swap slots in LO criticality mode LO-*table* and update the leeway of s_c and s_{swap}. Now, we must check whether fulfilled precedence constraints have been changed by swapping. If fulfilled precedence constraints have been changed then we re-schedule slot s_{swap}^{HI} and set the swapping slot as current slot and continue the scheduling process. This refers to Figure 1 column (III). If fulfilled precedence constraints have not been changed then we swap s_{swap}^{HI} and s_c^{HI} (unscheduled yet) and re-schedule s_{swap}^{HI} based on the fulfilled precedence constraints at that time – as described in subsection V-B. This refers to Figure 1 column (II).

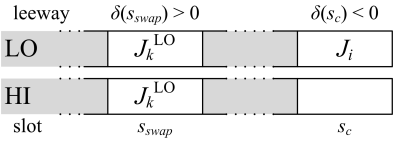


Fig. 4. Backtracking case 3: slots before backtracking

Case 4: In slot s_c^{LO}, a HI criticality job J_i^{LO} and in slot s_{swap}^{LO}, a LO criticality job J_k are scheduled. In s_{swap}^{HI}, no job is scheduled (Figure 5). We swap slots in LO criticality mode LO-*table*. In slot s_{swap}^{HI}, we scheduled the same job J_i^{LO} as in s_{swap}^{LO} (after swapping). As a consequence, we update $g_{sched}^{\Delta}(s)$ and leeway $\delta(s)$ for $s \in \{s_{swap}, .., s_c\}$. In the last step, we schedule the current slot in the HI criticality mode. This case refers to Figure 1 column (II).

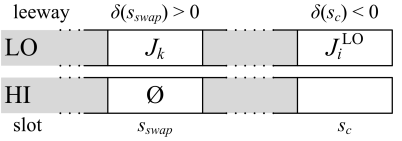


Fig. 5. Backtracking case 4: slots before backtracking

Case 5: In slot s_c^{LO}, a HI criticality job J_i^{LO} and in slot s_{swap}^{LO}, a LO criticality job J_k are scheduled. In s_{swap}^{HI}, a HI criticality job J_m^{Δ} is scheduled (Figure 6). First, we check whether scheduling J_m^{Δ} in s_c^{HI} leads to a deadline miss. If yes, then we have to search for another swapping slot. If no, then we swap slots in LO criticality mode LO-*table*. Then, we swap s_{swap}^{HI} and s_c^{HI} (unscheduled yet) and schedule in s_{swap}^{HI} the same job J_i^{LO} as in slot s_{swap}^{LO} (after swapping in LO-*table*). As a consequence, we update $g_{sched}^{\Delta}(s)$ and leeway $\delta(s)$ for $s \in \{s_{swap}, .., s_c\}$. This case refers to Figure 1 column (II).

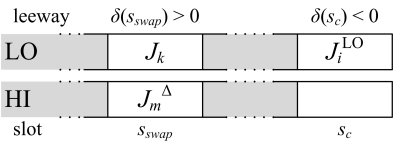


Fig. 6. Backtracking case 5: slots before backtracking

Case 6: In both slot s_c^{LO} and s_{swap}^{LO}, a HI criticality job J_i^{LO} and J_k^{LO} are scheduled. In s_{swap}^{HI}, the same HI criticality job J_k^{LO} is scheduled as in slot s_{swap}^{LO} (Figure 7). We swap slots in LO criticality modes and update the leeway of s_c and s_{swap}. Now, we must check whether fulfilled precedence constraints are not fulfilled after swapping anymore. If fulfilled precedence constraints have been changed, then we schedule J_i^{LO} in slot s_{swap}^{HI}, set the swapping slot as current slot, and continue the scheduling process. This refers to Figure 1 column (III). If fulfilled precedence constraints have not been changed, then we schedule J_i^{LO} in slot s_{swap}^{HI}. This refers to Figure 1 column (II).

VII. Evaluation

In this section, we show first evaluation results. We evaluated our algorithm by generating job sets with uniformly distributed utilizations by means of the UUniFast algorithm [8]. We generate a set of periodic tasks and unroll them into a set

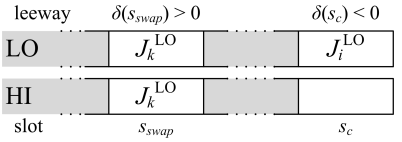


Fig. 7. Backtracking case 6: slots before backtracking

of jobs which are then handled by our algorithm. We use these synthetic workloads to show the correctness for general workloads. Rounding to full slot size values causes errors in the obtained utilization, hence, we specified that resulting job sets have errors less than 3%. We generated the mixed criticality job sets with the following constraints: the demand of all jobs with $C(\mathrm{LO})$ and the demand of HI criticality jobs with $C(\mathrm{HI})$ (original job parameters before splitting) have to be feasible by demand.

As input parameters, we used the utilization of all tasks with $C(\mathrm{LO})$ and the ratio of HI criticality jobs within each job set, which is in line with [7]. We evaluated the set of utilizations from 10% to 80% in steps of 10% with a ratio of HI criticality jobs of 50%. The range of LO criticality WCETs is $C_i(\mathrm{LO}) \in [1; 15]$. For the WCETs of HI criticality jobs, we used a high-scale factor $hsf = 3$, i.e. $C_i(\mathrm{HI}) \in [C_i(\mathrm{LO}); hsf \cdot C_i(\mathrm{LO})]$. The range of relative deadlines D_i was chosen within the interval $[45; 120]$ For each combination of input parameters, we generated $N = 1,000$ random job sets and scheduled them with our scheduler. We evaluated also other ratios of high criticality jobs and high scale factors, but for space reason, we only show a representative example.

Table II shows the success ratio of scheduling the generated job sets. The results are plotted in Figure 8. Due to $hsf = 3$, it is possible to obtain a utilization for all jobs under CAs' assumption of nearly 100% for low criticality utilizations above 30%. This leads to a drop of the success ratio, whereas the fixed priority approach is affected more strongly.

success ratio	$\sum_{i=1}^{n} U_i(\mathrm{LO})$							
	10%	20%	30%	40%	50%	60%	70%	80%
swapping	100.0	100.0	100.0	90.9	14.6	1.1	0.2	0.0
FPS	100.0	100.0	99.6	70.2	3.5	0.0	0.0	0.0

TABLE II
RESULTS: SUCCESS RATIO FOR $N = 1000$ AND $hsf = 3$

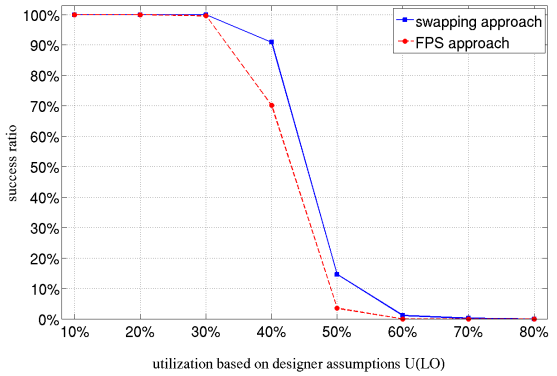


Fig. 8. Comparison between FPS approach [5] and our approach

VIII. CONCLUSION

Due to its run-time simplicity, extreme determinism, and ease of validation, the TT approach to real-time scheduling is heavily favored in industrial practice in many safety-critical application domains. In [5] an effort was made to extend results from the recently emergent field of mixed criticality scheduling to time-triggered scheduling, by proposing a TT-based framework for implementing mixed criticality systems, and presenting proof-of-concept algorithms for generating the schedule tables needed by this framework.

In this paper, we have extended and generalized the work described in [5]. We presented an algorithm for handling mixed criticality applications in time-triggered systems replacing these proof-of-concept methods with an algorithm for run-time execution and construction of schedule tables for efficiency and flexibility, providing realistic applicability. Our algorithm for the construction of the schedule tables is search-based; it is implemented as a tree search with backtracking. We devised two heuristics, one for the construction of the schedule tables and another for backtracking, based on the demand of HI criticality applications. These heuristics allow for a reduction of the search space and the time-complexity for scheduling decisions and backtracking; in addition to immediately yielding a constructive proof of the correctness of the schedule tables.

Due to the search tree based scheduling, the algorithm will be augmented to consider further constraints, in future. For example, extending the search not only to find a feasible schedule but also a schedule minimizing the preemptions.

REFERENCES

[1] *ARINC 653-1 Avionics application software standard interface*. 2003.
[2] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical report, The University of York, 1991.
[3] N. C. Audsley. *Flexible Scheduling in Hard-Real-Time Systems*. PhD thesis, Department of Computer Science, University of York, 1993.
[4] S. Baruah, V. Bonifaci, G. D'Angelo, Haohan Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 2012.
[5] S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Real-Time Systems Symposium*, 2011.
[6] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems Journal*, 1990.
[7] S.K. Baruah, A. Burns, and R.I. Davis. Response-time analysis for mixed criticality systems. In *Real-Time Systems Symposium (RTSS)*, 2011.
[8] Enrico Bini and GiorgioC. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30:129–154, 2005.
[9] A. Burns and R. Davis. Mixed criticality systems: a review. www-users.cs.york.ac.uk/ burns/review.pdf.
[10] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *RTSS*, 2009.
[11] G. Fohler. Changing operational modes in the context of pre run-time scheduling. *IEICE Transactions on Information and Systems*, 1993.
[12] G. Fohler. *Flexibillity in Statically Scheduled Hard Real-Time Systems*. PhD thesis, Technische Universität Wien, 1994.
[13] H. Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Springer, 2011.
[14] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 1985.
[15] T. Park and S. Kim. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *International Conference on Embedded Software*, 2011.

Mixed Criticality Scheduling in Time-Triggered Legacy Systems

Jens Theis and Gerhard Fohler
Technische Universität Kaiserslautern, Germany
Email: {jtheis,fohler}@eit.uni-kl.de

***Abstract*—Research on mixed criticality real-time scheduling has centered on an event-triggered (ET)/ priority-driven approach to scheduling. Regarding the time-triggered (TT) approach, which seems to have greater acceptability with certification authorities for safety critical domains, only first results have been presented, showing proof-of-concept of TT mixed criticality scheduling algorithms and comparing their resource utilization guarantees to those of ET ones. The algorithm is based on the offline construction of two coordinated schedule tables and an online mechanism. As a consequence, existing schedule tables for single criticality, possibly certified, have to be discarded.**

Here, we present an algorithm for the addition of mixed criticality scheduling to legacy TT systems. It leaves the existing schedule table unchanged, only provides analysis and adds a simple online mechanism to handle the changed criticality.

I. Introduction

In many domains such as avionics, industrial control, or health care, there is increasing demand for sharing computing platforms among applications with different importance and certification assurance levels. In the mixed criticality real-time scheduling model of [7], it is assumed that in order to certify a system as being correct, the certification authorities (CAs) mandate certain assumptions about the worst-case behavior of the system during runtime; these assumptions, e.g., for execution time, are typically far more conservative than the assumptions that the system designer would use during the process of designing, implementing, and testing the system if subsequent certification were not required. The difference in pessimism between designer's assumptions (likely at runtime) and CAs' mandate (most likely not at runtime) could be used to add non-critical activities and increase utilization. However, while the CAs are only concerned with the correctness of the safety-critical part of the system, the system designer is responsible for ensuring that the entire system is working correctly, including the non-critical parts. The current body of work in this area has focused on the event-triggered (ET)/priority-driven approach to scheduling.

Current practice in many domains, including (the safety-critical components of) automotive and avionics systems, which must meet multiple assurance requirements up to the highest criticality levels (e.g., DAL A in RTCA DO-178B or SIL4 in EN ISO/IEC 61508), however, favors a time-triggered approach (TT). In such TT systems, non-interference of safety-critical components by non-critical ones is ensured by strict isolation between components of different criticalities; Although such isolation facilitates the certification of the safety-critical functionalities, it can cause very low resource utilization.

A first result [1] shows proof-of-concept of mixed criticality real-time scheduling based on the TT approach. It is based on the offline construction of two coordinated schedule tables and an online mechanism to handle a change in criticality. In *legacy* TT systems, i.e. with existing, certified tables, this algorithm cannot be applied as it requires existing schedule tables to be changed, incurring substantial effort for recertification. At runtime, the low-criticality schedule table is executed until a high criticality job shows high criticality behavior and then the system switches to the high criticality schedule table. This solution shows a low runtime overhead but at cost of inflexibility.

The ET approach mixed critical EDF with mode switches was presented in [6]. In this approach, also two priority tables are created based on the deadlines of the jobs. When a high-criticality job exceeds its low criticality worst-case execution time (WCET), the system is switched to high criticality state with the high criticality priority table.

In this paper, we present a method to add the handling of criticality changes to existing schedule tables for legacy TT systems. It analyzes the existing table and properties of the high-criticality job set offline. A simple online mechanism then executes the jobs according to the existing table, manages a change of criticality, and then continues to execute the high-criticality job set. In case the existing schedule table is not suitable for the given mixed criticality job set, indications for its modification can be given. While in this case recertification may become necessary, the efforts will be lower than reconstruction of the schedule table from scratch.

Our method is based on slot-shifting [3] which was originally designed to add flexibility to TT systems with acceptable runtime overheads [5]. It takes the original task set and a constructed scheduling table[1] as input. As the table is constructed offline, complex constraints, such as distributed systems, end-to-end deadlines, precedence, etc. can be considered. It analyzes the table and the constraints to determine unused resources and leeways, which are represented as *spare capacities* offline. These can be used to provide flexibility and handle firm aperiodic tasks at runtime. Here, we build upon the offline analysis part and spare capacities to handle changes in criticality.

The remainder of this paper is structured as follows: Section II presents terms and notation used in this paper.

This work has been supported in part by the European project DREAMS under project No. 610640.

[1] It does not depend on a particular offline table construction algorithm.

are incorporated into the online phase, i.e. the actual runtime scheduling, which is presented in Section IV. In Section V, we recapitulate the results and the applicability of our solution. Finally, Section VI concludes the paper.

II. Terminology and Notations

In this paper, we assume a dual-criticality system with the criticality levels LO and HI. Unless otherwise defined, we represent absolute times (e.g. release times) by lower case variables and relative times (e.g. WCET) by upper case variables.

The dual criticality jobs J_i with $i \in \{1, .., n\}$ are characterized by the 5-tuple $\langle \chi_i, r_i, d_i, C_i(\text{LO}), C_i(\text{HI}) \rangle$, with

- $\chi_i \in \{\text{LO}, \text{HI}\}$: criticality level of job i
- $r_i \in \mathbb{R}^+$: release time of job i
- $d_i \in \mathbb{R}^+$: absolute deadline of job i with $d_i > r_i$
- $C_i(\text{LO})$: LO criticality WCET
- $C_i(\text{HI})$: HI criticality WCET

which is in line with the denomination in the work of Baruah and Fohler [1]. For LO criticality jobs, we assume $C_i(\text{LO}) = C_i(\text{HI})$, whereas for HI criticality jobs $C_i(\text{LO}) \leq C_i(\text{HI})$.

Slot-shifting [3] uses *slots* as granularity for scheduling decisions. A slot $s = i$ is the time interval $[i; i+1)$. This interval contains the worst case time to schedule jobs Δt_S and the minimum guaranteed time to execute jobs Δt_E. The length of Δt_S and Δt_E is determined by the designer depending on the system under development. The WCET is given as multiples of Δt_E. The length of a slot is defined as $|s| = \Delta t_S + \Delta t_E$.

Another important concept of slot-shifting are *capacity intervals* which are used to manage the execution of jobs. Section III-A explains this concept in detail. The general notation of capacity intervals is: start and end of a capacity interval I_i is denominated by: $start(I_i)$ and $end(I_i)$, respectively. A capacity interval I_i represents the time window $[start(I_i); end(I_i))$. As a result, the length of a capacity interval I_i is calculated by $|I_i| = end(I_i) - start(I_i)$. I_c represents the current capacity interval, i.e. in which capacity interval the current point in time is located.

III. Offline Phase

In the offline phase of slot-shifting, we can resolve complex constraints – e.g. precedence constraints – which is not the scope of this paper. The interested reader is referred to [3]. Slot-shifting works on job-level, i.e. the instances of a periodic task are scheduled as single jobs. Scheduling on job-level allows for scheduling of time-triggered and event-triggered tasks and jobs. In this section, we show how to determine capacity intervals and how to calculate spare capacities which form the basis for the slot-shifting runtime scheduler.

A. Capacity Intervals

We divide the schedule into disjoint *capacity intervals* I_i with $i \in \{0, .., m\}$ based on the release times and deadlines of the jobs. It is important to highlight that capacity intervals are not identical to the execution windows, i.e. the time between release and deadline of a job. In the following, capacity intervals are briefly referred to as intervals. Each deadline of a job marks the end $end(I_i)$ of an interval I_i. Each job $J_k, k \in \{1, .., n\}$ is assigned to an interval with $end(I_i) = d_k$. Jobs with the same deadline belong to the same interval. The earliest start time $est(I_i)$ of an interval I_i is determined by the minimum of all release times of jobs assigned to this interval:

$$est(I_i) = \min_{\forall J_k \in I_i} (r_k) \tag{1}$$

The start of an interval is determined by the maximum of its earliest start time and the end of the previous interval:

$$start(I_i) = max\,(end(I_{i-1}), est(I_i)) \tag{2}$$

The gaps between the determined intervals above are defined as *empty intervals*, i.e. there is no job assigned to them. An interval I_i is called *independent* if there is no interval I_e with $e < i$ and $end(I_e) > est(I_i)$ and there is no interval I_l with $i < l$ and $end(I_i) > est(I_l)$. The length $|I_i|$ of an interval is calculated by equation 3.

$$|I_i| = end(I_i) - start(I_i) \tag{3}$$

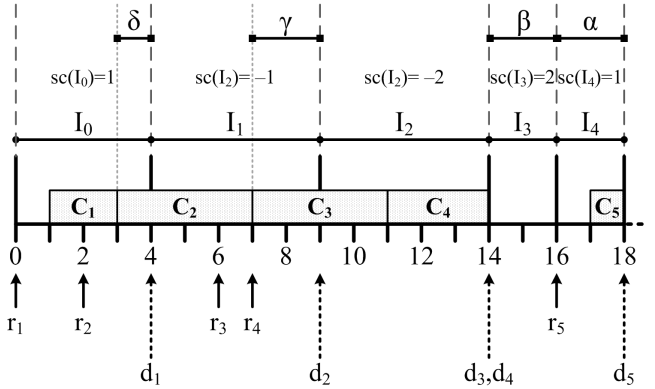
Based on the observation that demand-based schedulability tests need only to check intervals until deadlines of jobs [2], capacity intervals, which partition considered demand based on job deadlines, simplify the maintenance of demand requirements and scheduling of the demand, respectively, at runtime.

B. Spare Capacities

In the following, we explain the general concept of *spare capacities* based on non-mixed-criticality jobs (as shown in the original slot-shifting [3]). After that, we present how this is applied to mixed criticality job sets. The spare capacity $sc(I_i)$ of an interval I_i represents the amount of available resources within this interval after guaranteeing TT jobs. The difference between the length of the interval and the amount of demand of jobs assigned to this interval determines the amount of available resources. Further, it is possible that the demand within an interval is greater than the length of the interval such that the spare capacity will be negative. In the following, we show how spare capacities are calculated in detail and in combination with section IV-A, we show the consequences of negative spare capacities. We calculate spare capacities beginning with the last interval I_m until the first interval I_0.

Figure 1 shows an example to illustrate the calculation of spare capacities. As shown in section III-A, we determine the

intervals $I_0 - I_4$. In this example, we schedule the jobs as late as possible to exemplify the calculation of spare capacities. Jobs cannot be scheduled before their release time; otherwise, the schedule is not feasible and the job set is not schedulable. In the figure, time span α represents an independent interval. The spare capacity of an independent interval is calculated by the difference between the length of the interval and the sum of WCETs of jobs assigned to this interval. Time span β shows an empty interval, i.e. there are no jobs assigned to it, and hence, the spare capacity of this interval is the length of the interval. The consequences of negative spare capacities are illustrated by time span γ. The amount of demand assigned to I_2 is more than the length of the interval. As a result, interval I_1 has to lend capacity to the succeeding interval I_2. The consequence of the negative spare capacity is called *borrowing*. Time span δ shows the consequences of borrowing propagation, i.e. interval I_1 lent capacity to I_2 such that I_1 itself has to borrow capacity from the earlier interval I_0. Eventually, interval I_0 is long enough to lend capacity to I_1 and schedule the jobs assigned to I_1. Thus, the spare capacity of I_0 is non-negative.



J_i	r_i	d_i	C_i
J_1	0	4	1
J_2	2	9	4
J_3	6	14	4
J_4	7	14	3
J_5	16	18	1

Figure 1. Spare capacity calculation example: for simplicity of the example without different criticality levels

We can test the schedulability based on the spare capacities, under the condition of obeying release times and deadlines. Spare capacities of independent intervals have to be non-negative. Further, after borrowing and borrowing propagation there must be an earlier interval with non-negative spare capacity.
Equation (4) shows the calculation of spare capacities for non-mixed-criticality jobs, which is in the following applied to mixed criticality job sets. The calculation includes the requirements described by Figure 1.

$$sc\left(I_i\right) = |I_i| - \sum_{J_k \in I_i} C_k + min\left(sc\left(I_{i+1}\right), 0\right) \qquad (4)$$

For the mixed criticality case, we use the calculations of the spare capacities presented above. We calculate two spare capacity values for each interval: $sc^{\text{LO}}(I_i)$ and $sc^{\text{HI}}(I_i)$. The calculation of spare capacities is done by equations (5) and (6). Spare capacities $sc^{\text{LO}}(I_i)$ represent available capacities based on designer's assumptions, i.e. considering $C(\text{LO})$, with all jobs. Additionally, $sc^{\text{HI}}(I_i)$ represents available resources based on CAs' assumptions, i.e. considering $C(\text{HI})$, with only HI criticality jobs.

$$sc^{\text{LO}}\left(I_i\right) = |I_i| - \sum_{J_k \in I_i} C_k(\text{LO}) + min\left(sc^{\text{LO}}\left(I_{i+1}\right), 0\right) \quad (5)$$

$$sc^{\text{HI}}\left(I_i\right) = |I_i| - \sum_{\substack{J_k \in I_i \\ \wedge \chi_k = \text{HI}}} C_k(\text{HI}) + min\left(sc^{\text{HI}}\left(I_{i+1}\right), 0\right) \quad (6)$$

As Figure 1 (intervals I_0 and I_4) shows, slot-shifting allows for non-work-conserving scheduling based on spare capacities. Further, slot-shifting allows for scheduling of strictly periodic jobs, i.e. jobs that have to be executed directly at their periodic release.

IV. Online Phase

At runtime, we execute mixed criticality jobs based on the spare capacity in the current interval I_c and the deadlines of the jobs. In this section, we present how to select the next job for execution. Further, we show the update procedures for spare capacities depending on the job execution.

A. Decision Mode and Selection Function

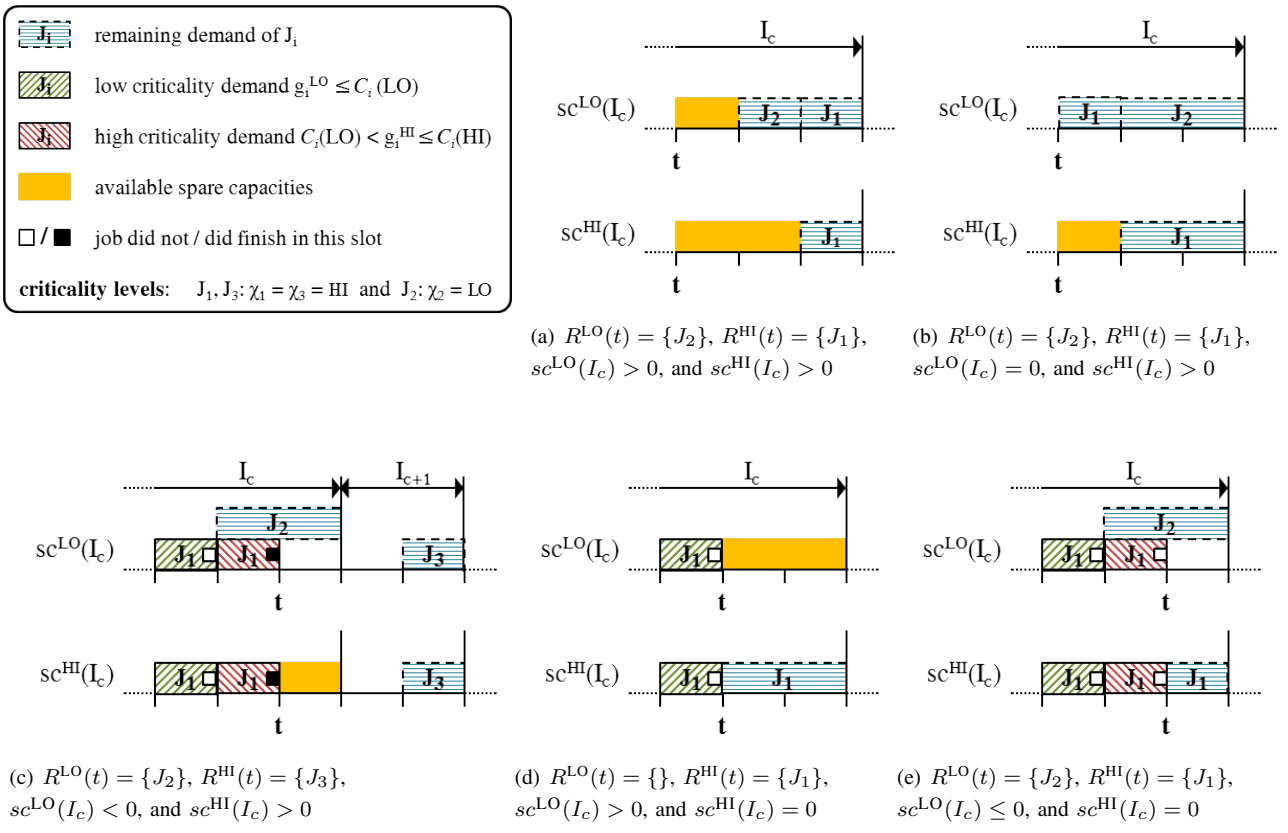
The decision mode of the slot-shifting runtime scheduler is preemptive at slot borders. We use three ready queues: $R^{\text{LO}}(t) = \{J_i | r_i \leq t \wedge \chi_i = \text{LO}\}$ contains TT LO criticality jobs. Further, $R^{\text{HI}}(t) = \{J_i | r_i \leq t \wedge \chi_i = \text{HI}\}$ is used for TT HI criticality jobs. An implementation with only one queue is also possible, but for simplicity of explanation, we will present the algorithm based on two queues. We define $R(t) := R^{\text{LO}}(t) \cup R^{\text{HI}}(t)$ which represents all guaranteed jobs. The scheduler selects the next executing jobs based on the ready queues, the intervals and the spare capacities. In contrast to standard slot-shifting, LO criticality spare capacities can be negative in the current interval I_c. This can occur when a HI criticality job executes for more than $C(\text{LO})$ and we have to continue its execution. As long as the LO criticality spare capacity in the current interval is negative, i.e. $sc^{\text{LO}}(I_c) < 0$, we can only execute HI criticality jobs.
Based on these observations, we can distinguish the following list of all possible decision cases at time t.

a) $R(t) = \{\}$: slot is idle because there are no ready jobs.

b) $R(t) \neq \{\} \wedge sc^{\text{HI}}(I_c) > 0$:

1) $sc^{\text{LO}}(I_c) > 0$:
We select the ready job with the earliest deadline in $R(t)$. Figure 2(a) illustrates this situation with an example.

2) $sc^{\text{LO}}(I_c) = 0$:
We select the job with the earliest deadline in $R(t)$ because there are available resources for HI criticality jobs and thus, no need to prioritize them. On the contrary, there is no leeway for LO criticality demand



(a) $R^{\text{LO}}(t) = \{J_2\}$, $R^{\text{HI}}(t) = \{J_1\}$, $sc^{\text{LO}}(I_c) > 0$, and $sc^{\text{HI}}(I_c) > 0$

(b) $R^{\text{LO}}(t) = \{J_2\}$, $R^{\text{HI}}(t) = \{J_1\}$, $sc^{\text{LO}}(I_c) = 0$, and $sc^{\text{HI}}(I_c) > 0$

(c) $R^{\text{LO}}(t) = \{J_2\}$, $R^{\text{HI}}(t) = \{J_3\}$, $sc^{\text{LO}}(I_c) < 0$, and $sc^{\text{HI}}(I_c) > 0$

(d) $R^{\text{LO}}(t) = \{\}$, $R^{\text{HI}}(t) = \{J_1\}$, $sc^{\text{LO}}(I_c) > 0$, and $sc^{\text{HI}}(I_c) = 0$

(e) $R^{\text{LO}}(t) = \{J_2\}$, $R^{\text{HI}}(t) = \{J_1\}$, $sc^{\text{LO}}(I_c) \leq 0$, and $sc^{\text{HI}}(I_c) = 0$

Figure 2. Examples situation when selecting the next job for decision cases with $R(t) \neq \{\}$ for an exemplary point in time t — J_1, J_3: HI **criticality jobs,** J_2: LO **criticality job**

and thus, we have to select a guaranteed job. Figure 2(b) illustrates this situation with an example.

3) $sc^{\text{LO}}(I_c) < 0$:
We select the job with the earliest deadline in $R^{\text{HI}}(t)$. There is not enough available LO criticality spare capacity in the current interval to complete remaining LO criticality jobs and hence, they are skipped. Figure 2(c) illustrates this situation with an example.

c) $R(t) \neq \{\} \wedge sc^{\text{HI}}(I_c) = 0$:

1) $sc^{\text{LO}}(I_c) > 0$:
The job with the earliest deadline in $R^{\text{HI}}(t)$ is selected because we have to execute a HI criticality job to guarantee completion with CAs' assumptions. Figure 2(d) illustrates this situation with an example.

2) $sc^{\text{LO}}(I_c) \leq 0$:
We select job with the earliest deadline in $R^{\text{HI}}(t)$ because of the reasons mentioned in case c1. Figure 2(e) illustrates this situation with an example.

d) $R(t) \neq \{\} \wedge sc^{\text{HI}}(I_c) < 0$: The HI criticality spare capacity cannot be less than zero. This could only happen if we execute a HI criticality job J_i for more than $C_i(\text{HI})$ which we assume is prevented by the system.

B. Spare Capacity Maintenance

As a consequence of the process to select the next executing job, we have to update the spare capacities depending on the criticality level and type of the job.

No execution: If an idle slot has been scheduled, we decrease both $sc^{\text{LO}}(I_c)$ and $sc^{\text{HI}}(I_c)$ by one slot.

Guaranteed job execution: If a guaranteed job J_i, either TT or firm ET, has been scheduled, then we have to differentiate whether J_i is assigned to the current interval I_c or to a later interval I_k. Further, the fact whether a HI criticality job exceeded $C(\text{LO})$, i.e. showed HI behavior, influences the maintenance of spare capacities.

A) $J_i \in I_c$

1) J_i did not exceed $C_i(\text{LO})$:
In both spare capacity calculations, $sc^{\text{LO}}(I_c)$ and $sc^{\text{HI}}(I_c)$, the scheduled demand has already been considered and hence, both spare capacities in the current interval remain unchanged.

2) J_i exceeded $C_i(\text{LO})$:
As the job executes for more than the considered amount of LO criticality execution time, we have to decrease the LO criticality spare capacity in the current interval by one slot. On the contrary, for the HI criticality spare capacity $sc^{\text{HI}}(I_c)$, this scheduled execution has already been considered and thus, $sc^{\text{HI}}(I_c)$ is unchanged.

B) $J_i \in I_k$ with $I_k \neq I_c$

1. J_i did not exceed $C_i(\text{LO})$:
The scheduled demand has not been considered in $sc^{\text{LO}}(I_c)$. As a consequence, we decrease $sc^{\text{LO}}(I_c)$ by one slot. Additionally, we increase $sc^{\text{LO}}(I_k)$ where the demand has been originally considered in the spare

capacity calculations. In other words, one slot of execution is swapped between the current interval I_c and the assigned job interval I_k.
Furthermore, there is the aspect of borrowing which has to be considered: If $sc^{\text{LO}}(I_k)$ was less than zero before increasing by one in the current step, then I_k was borrowing capacity from at least one earlier interval. Thus, we have to increase the spare capacities by one if there was borrowing or borrowing propagation in one or several of the intervals from I_c to I_{k-1}.
For the HI criticality spare capacities, we have to apply the same procedure as for the LO criticality ones.

2. J_i exceeded $C_i(\text{LO})$:
The demand has not been considered in the LO criticality spare capacities, neither in $sc^{\text{LO}}(I_c)$ nor in $sc^{\text{LO}}(I_k)$. Thus, only the spare capacity in the current interval $sc^{\text{LO}}(I_c)$ is decreased by one.
For the HI criticality spare capacities, we have to apply the same procedures as in step B1; which are:
The scheduled demand has not been considered in $sc^{\text{HI}}(I_c)$ and thus, we have to decrease $sc^{\text{HI}}(I_c)$ by one slot. Further, $sc^{\text{HI}}(I_k)$, where the demand has been originally considered, is increased by one.
Still, there may be the aspect of borrowing which has to be considered: If $sc^{\text{HI}}(I_k)$ was less than zero before increasing by one in the current step, then I_k was borrowing capacity from at least one earlier interval. Thus, we have to increase the spare capacities by one if there was borrowing or borrowing propagation in one or several of the intervals from I_c to I_{k-1}.

After execution of a job in the current slot and before the scheduling process continues in the next slot, we have to compare the actual execution time of the job with the WCET for the LO and the HI criticality case if the job finished in the current slot. If TT jobs complete earlier than their specified WCET ($C(\text{LO})$ for LO and $C(\text{HI})$ for HI criticality spare capacities), the difference between actual execution time and specified WCET can be added to the spare capacities of the corresponding intervals, as shown in [4].

As a conclusion, we can make efficient use of the available resources by the update mechanism presented above. Further, the LO criticality and HI criticality spare capacities allow for flexibility to react to the actual job behavior.

V. Discussion

In contrast to [1], there is no need to construct two schedule tables for the different requirements of the designer and the CAs. We only need to calculate two sets of spare capacities for the existing intervals whereas the intervals are identical for designer's and CAs' assumptions. At runtime, the certified schedule table is unchanged. Based on the spare capacities the runtime mechanism handles the requirements of LO and HI criticality jobs within the constraints of the offline computed, certified schedule table. This simplifies offline preparation and certification.

The runtime overhead of slot-shifting has been compared to Linux' Completely Fair Scheduler and Litmus RT, with the conclusion that the overhead of slot-shifting is in the same order as the reference schedulers [5]. The necessary changes to extend the implemented slot-shifting version to our mixed criticality slot-shifting affect only the selection of the next job to execute and the update of the spare capacities. In both job selection and spare capacity update, only a second spare capacity has to be checked and updated, respectively.

The presented algorithm can be easily applied to resume the LO criticality state after a switch to HI. The available spare capacities are used to execute HI criticality jobs that exceed the WCET based on the designer assumptions such that we do not need to drop LO criticality jobs as long as there are enough spare capacities available to not harm the execution of HI criticality jobs. If there are no spare capacities left, we continue the execution with the restricted job set of only HI criticality jobs to guarantee them. The maintenance of spare capacities within the capacity intervals allows for switching back to the LO criticality system state as soon as we can guarantee the execution of all jobs based on designer assumptions. This is achieved by updating the LO criticality spare capacities although only HI criticality jobs are executed. The LO criticality spare capacities indicate the point in time when the WCET of LO criticality jobs can be guaranteed for them, again. The maximum number of spare capacity updates per slot occurs when there is borrowing propagation and hence, is bounded by the number of jobs. This is based on the fact that job deadlines refer to end of intervals. More intervals than jobs are only possible if there are empty intervals but empty intervals are not affected by borrowing and borrowing propagation. As a consequence, the complexity of the selection function and the spare capacity maintenance can be bounded by a linear function.

As a result, we can make use of TT legacy systems and add flexibility without harming certified schedule tables at runtime. Additionally, the implementation of slot-shifting shows an applicable runtime overhead such that slot-shifting represents a valid choice for safety-critical TT legacy systems with mixed criticality job sets.

VI. Conclusion

In this paper, we have presented a method for including mixed criticality real-time scheduling following the model of [7] to legacy TT systems. Earlier results for TT systems [1] require the offline construction of two coordinated schedule tables from scratch for mixed criticality task sets. In case a schedule table already exists and has been certified, this approach, necessitates complete recertification.

In contrast, the method presented in this paper takes an existing schedule table and the properties of the HI criticality job set as input. It performs analyses and provides a simple online mechanisms to include additional criticality job sets. The jobs of the original schedule table are executed as before. When a change of criticality arises, the online mechanism manages it, and then continues to execute the HI criticality jobs. In case the existing schedule table is not suitable for the given mixed criticality job set, indications for its modification can be given. While in this case recertification may become necessary, the efforts will be lower than reconstruction of the schedule table from scratch.

Future work will include analysis of the existing work regarding limitations of the presented method and comparison

to pure TT and ET approaches. Additionally, the inclusion of aperiodic jobs will be in the focus of our work. Further, the extension to arbitrary criticality levels will be included in future research.

REFERENCES

[1] Sanjoy Baruah and Gerhard Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Real-Time Systems Symposium*, 2011.

[2] Sanjoy K. Baruah, Louis E. Rosier, and Rodney R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems Journal*, 1990.

[3] Gerhard Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. *Real-Time Systems Symposium*, 1995.

[4] Damir Isovic and Gerhard Fohler. Handling mixed sets of tasks in combined offline and online scheduled real-time systems. *Real-Time Systems Journal*, 2009.

[5] Stefan Schorr and Gerhard Fohler. Integrated time- and event-triggered scheduling an overhead analysis on the arm architecture. In *International Conference on Embedded and Real-Time Computing Systems and Applications*, 2013.

[6] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Mixed critical earliest deadline first. In *Euromicro Conference on Real-Time Systems*, 2013.

[7] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. *Real-Time Systems Symposium*, 2007.