Two Protocols to Reduce the Criticality Level of Multiprocessor Mixed-Criticality Systems

François Santy
PARTS Research Center
Université Libre de Bruxelles (ULB)

Gurulingesh Raravi
CISTER-ISEP Research Center
Polytechnic Institute of Porto

Geoffrey Nelissen
CISTER-ISEP Research Center
Polytechnic Institute of Porto

Vincent Nelis
CISTER-ISEP Research Center
Polytechnic Institute of Porto

Pratyush Kumar
Computer Engineering and Networks
Laboratory, ETH Zurich

Joël Goossens
PARTS Research Center
Université Libre de Bruxelles (ULB)

Eduardo Tovar
CISTER-ISEP Research Center
Polytechnic Institute of Porto

ABSTRACT

Most of the existing research on multiprocessor mixed-criticality scheduling has focused on ensuring schedulability of the task set when the criticality level of the system increases. Furthermore, upon increasing the criticality level, most of these scheduling approaches suspend the execution of the lower criticality tasks in order to guarantee the schedulability of the higher criticality tasks. Although there exists a couple of approaches to facilitate the execution of some of the lower criticality tasks using the available slack in the system, to the best of our knowledge, there is no *efficient* mechanism that allows for eventually *decreasing* the criticality level of the system in order to resume the execution of the suspended lower criticality tasks. We refer to the problem of deciding when and how to lower the criticality level of the system as the "Safe Criticality Reduction" (SCR) problem. In this work, we design two solutions that are *independent* of the number of criticality levels and the number of processors and prove their correctness. The first protocol can be applied to any fixed task priority scheduler, and an upper-bound on the suspension delay suffered by the lower criticality tasks is presented. The second protocol can be applied to any fixed job priority scheduler and hence dominates the first protocol albeit with a higher run-time overhead. To the best of our knowledge, these are the first solutions for the SCR problem on multiprocessor platforms.

Keywords

Real-time scheduling, Identical Multiprocessor, Mixed-Criticality, Decrease Criticality.

RTNS 2013 , October 16 - 18 2013, Sophia Antipolis, France
Copyright 2013 ACM 978-1-4503-2058-0/13/10$15.00.
http://dx.doi.org/10.1145/2516821.2516834.

1. INTRODUCTION

Many industrial domains such as avionics, automotives, smart manufacturing, etc. rely heavily on the use of real-time embedded systems. Typically, such systems are subject to stringent timing requirements. These systems are typically composed of a set of very specific functionalities, henceforth called *tasks*. In order to guarantee that these systems always react within some pre-determined time-bounds, each task must be thoroughly analyzed for their temporal behaviour. In particular, one must determine their worst-case execution times (WCETs) and many tools and approaches exist to perform this estimation. The rigorousness of these tools depends on the desired level of confidence that the actual execution time of a task will not exceed its estimated WCET. For example, the WCET of a task can be computed empirically by using measurement tools and run-time traces, in which case the WCET estimation is the maximum execution time observed during the simulation. Alternatively, the WCET can be estimated by using more conservative but safer approaches based on parsing and analyzing the source code, by figuring out the worst possible case when this code is executed on the target platform. Typically, WCETs determined by simulation are optimistic and thus less reliable than the WCETs estimated by statically analyzing the source code. However, the latter approach usually overly estimates the actual execution requirements of the tasks. The method to be employed to estimate the WCET of a task thus depends on the consequences of the analyzed task missing its deadlines. Therefore, each task is subject to a "risk analysis" that will decide on its *criticality level*[1]. Consequently:

- The WCET of higher criticality tasks are determined by using conservative approaches which provide safe but overly pessimistic estimates since the timeliness of these tasks is crucial for the system;
- The WCET of lower criticality tasks are determined by using less conservative approaches since these tasks tolerate occasional deadline misses.

[1] In industrial standards, the criticality of a task is sometimes referred to as its *safety integrity level* (SIL), or *Design Assurance Level* (DAL).

When designing systems with tasks of different criticality, the usual approach is to partition the system in space and time, and isolate tasks with a common criticality in a dedicated partition [1]. This eliminates any interference across different criticality levels, and thus preserves a simple modularity in the design and certification process of the entire system. However, in order to efficiently use the computing resources (which is hard to achieve with partitioned scheduling), as well as to reduce the system's SWaP (size, weight and power) due to budgetary constraints, there has been a strong push towards integrating multiple functionalities onto the same computing platform. To allow tasks with different criticality levels to run concurrently on the same computing platform, there is a need to design efficient protocols which facilitate this. Many solutions have been proposed in the past few years, which have been witnessing an ever-growing interest in the real-time scheduling community (see Section 1.1).

Among those solutions, one model has been commonly adopted [3, 7, 14, 15, 19]. Each task is modeled using multiple WCETs — one WCET for each criticality level that is less than or equal to the criticality level of the task. That is, a task of criticality level ℓ has a WCET estimate at every criticality level that is less than or equal to ℓ with the following rule: the higher the criticality level, the more conservative the WCET estimate[2]. During run-time, the module has a "current criticality level" henceforth denoted by CL. Usually, CL is initialized to the lowest criticality level. Whenever a task executes for longer than its WCET at the current criticality level CL, then CL is raised to the next criticality level. When this happens, all tasks with a criticality level less than the new CL are suspended, i.e. they are not executed anymore. This is done in order to ensure that the tasks of high criticality will always be granted enough computing resources to complete their execution on time, as their timeliness is more important than the timeliness of the less critical tasks.

When considering system safety, mixed-criticality is a natural approach, since it aims at favouring the timeliness of the functionalities that are crucial for the proper functioning of the system. Therefore, most of the current work has been oriented toward the study of protocols that allow to respond promptly and adequately to the increase of the criticality level of the system. However, there has not been much focus, yet, on the problem of safely decreasing the CL. A decrease in CL from level h to level $\ell < h$ consists in re-activating *all* the suspended tasks that have a criticality level greater than or equal to ℓ. This thus improves the timing guarantees provided by the system, in the sense that the timeliness of both the currently executing tasks *as well as* the re-activated tasks are met. However, any such decrease must be carefully done: it must be ensured that the re-activated lower criticality tasks do not interfere with the schedulability guarantees of the higher criticality tasks. We refer to this problem as the "Safe Criticality Reduction (SCR)" problem.

1.1 Related Work

Mixed-Criticality scheduling was initially introduced by Vestal [23] and since then it has gained increasing interest in the real-time research community. Mixed-criticality systems are found in almost all the industrial domains such as avionics, automotives, etc. [12, 22], where applications are subject to multiple certification requirements. Many works have addressed the scheduling problem for such systems implemented upon uniprocessor platforms [3–7, 13, 17, 20, 21]. More recently, research has been oriented towards the study of mixed-criticality scheduling upon multiprocessor platforms [14, 16, 19, 19]. Many of the previous results focus on ensuring the timeliness of higher criticality tasks, potentially at the expense of the lower criticality tasks. Although there exists a couple of approaches to facilitate the execution of some of the lower criticality tasks using the available slack in the system, to the best of our knowledge, there is no *efficient* mechanism that allows for eventually *decreasing* the criticality level of the system in order to resume the execution of the suspended lower criticality tasks. Furthermore, many results are restricted to an easier subproblem of mixed-criticality, with only two criticality levels. The work by Santy et. al. [20] comes closest to our work in the sense that it has addressed the problem under consideration, but only for task sets scheduled using fixed-task priority algorithms on uniprocessor platforms.

1.2 This Research

In this work, we address the problem of when and how to reduce the criticality level of the system so that all the lower criticality level tasks are (re-)activated without jeopardising the schedulability of the system. We refer to this problem as "Safe Criticality Reduction" (SCR) problem. To address this problem, we propose two protocols: (P1) a fixed-task priority protocol with negligible run-time overhead and with a proven upper bound on the time to switch back to lower criticality mode and (P2) a fixed-job priority protocol which dominates the first protocol (since fixed-task priority is a special case of fixed-job priority) but at the cost of higher run-time overhead. We believe that the significance of this work is as follows: 1) there exists no protocols for SCR problem in the past and hence our protocols are the first of their kind, 2) our protocols can be applied to mixed-criticality systems with any number of criticality levels, 3) our protocols are generic in the sense that the first protocol can be applied to any fixed-task priority predictable scheduler and the second protocol can be applied to any fixed-job priority predictable scheduler and finally 4) the protocols are applicable to both global and partitioned scheduling paradigms[3].

2. MODEL OF COMPUTATION

2.1 Task and platform model

We consider an identical multiprocessor platform π comprising m unit-speed processors, denoted by π_j with $j = \{1, 2, \ldots, m\}$.

We consider a mixed-criticality task set τ comprising n *constrained-deadline sporadic tasks*, denoted by τ_i with $i = \{1, 2, \ldots, n\}$. Let the set $\mathbb{L} = \{1, \ldots, L\}$ denote different criticality levels in the system with level 1 begin the least

[2] Modeling a task with a vector of monotonically increasing WCET might for instance originate from a prior probabilistic timing analysis [24]. Such an analysis computes several WCET bounds for a task, each bound being associated to a probability of that task exceeding that particular WCET. This modelisation of real-time tasks is particularly suited for functionalities being subject to mandatory certification, as certification processes often impose a specific maximum probability of failure per criticality level [12, 22].

[3] For ease of explanation, thought the paper, we explain the protocols for global scheduling algorithms and in Section 6, we discuss how the protocols can also be applied to partitioned scheduling paradigms.

critical level and level L being the most critical level. Each task τ_i is characterized by a 4-tuple $\langle L_i, C_i, T_i, D_i \rangle$ where:

- $L_i \in \mathbb{L}$ denotes the *criticality level* of task τ_i;
- $C_i \in \mathbb{N}_0^L$ denotes a vector $\langle C_i(1), C_i(2), \ldots, C_i(L) \rangle$ of size L, where $C_i(\ell)$ denotes the worst-case execution time of task τ_i at criticality level $\ell \in \mathbb{L}$;
- $T_i \in \mathbb{N}_0$ denotes the *minimum inter-arrival time* of task τ_i;
- $D_i \in \mathbb{N}_0$ denotes the *deadline* of task τ_i, with $D_i \leq T_i$.

We assume $C_i(\ell)$ is monotonically increasing for increasing values of ℓ. More precisely, for task τ_i the following two conditions hold:

- $\forall \ell \in [1, L_i)$, $C_i(\ell) \leq C_i(\ell + 1)$;
- $\forall \ell \in [L_i, L]$, $C_i(\ell) = C_i(L_i)$.

We will assume that no task is supposed to execute longer than its WCET at its own criticality level, meaning that the probability of a task exceeding its WCET at its own criticality level is zero. For convenience, we will use the notation $\tau^{(k)}$, with $k \in \mathbb{L}$, to denote the set of tasks having criticality level greater than or equal to k.

Each task τ_i releases a (potentially infinite) sequence of jobs with two consecutive jobs separated by at least T_i time units. Hereafter, we call such a collection of jobs generated by τ, an *instance* of τ. The k^{th} job $J_{i,k}$ released by the mixed-criticality task τ_i is characterized by a 3-tuple of parameters $\{r_{i,k}, d_{i,k}, c_{i,k}\}$ where:

- $r_{i,k} \in \mathbb{N}$ denotes the *release time* of job $J_{i,k}$;
- $d_{i,k} \in \mathbb{N}_0$ denotes the absolute *deadline* of job $J_{i,k}$ and is given by $d_{i,k} \overset{\text{def}}{=} r_{i,k} + D_i$;
- $c_{i,k} \in \mathbb{N}_0$ denotes the actual *execution time* of job $J_{i,k}$. From the specifications of τ_i, we can say that $c_{i,k} \leq C_i(L_i)$, but the exact value of $c_{i,k}$ will not be known until $J_{i,k}$ is released and completes its execution.

We will furthermore use the notations $s_{i,k} \in \mathbb{N}$ and $f_{i,k} \in \mathbb{N}_0$, to denote the *exact* time-instant at which job $J_{i,k}$ starts and finishes its execution, respectively. A job $J_{i,k}$ is said to be *active* at time t if and only if $r_{i,k} \leq t \leq f_{i,k}$.

Definition 1. (Scenario) *Given a sequence of jobs, a scenario represents the set of actual execution times for each of these jobs.*

Definition 2. (Criticality of a scenario) *The criticality of a scenario is given by the lowest criticality level, such that no job overruns its WCET at that specific criticality level:*

$$\text{scenario criticality} = \min\{\ell \,|\, c_{i,k} \leq C_i(\ell), \forall i, k\}$$

Definition 3. (Worst-case response time) *The worst-case response time (WCRT) $R_i(\ell)$ of task τ_i at criticality level ℓ is the maximum amount of time that elapses between the release of any job $J_{i,k}$ of τ_i at time $r_{i,k}$, and its completion at time $f_{i,k}$, in any scenario of criticality level ℓ.*

2.2 Scheduler specification

In this work, we consider *global, fully-preemptive* and *work-conserving* scheduling policies, according to Definition 4.

Definition 4. (Global, fully-preemptive, work-conserving scheduler) *At each time instant, a global, fully-preemptive and work-conserving scheduler dispatches the m highest priority jobs (if any) on the m processors of the platform, is allowed to interrupt a job that is executing on a given processor to assign another active job to this processor, and never idles a processor while there is an active job..*

We will furthermore consider both fixed task priority (FTP) and fixed job priority (FJP) scheduling policies, according to the definitions given below.

Definition 5. (FTP scheduler) *A scheduler is said to be* FTP *if it assigns a single static priority to each task (though different tasks may share the same priority). It follows that different jobs released by the same task will have the same priority.*

Definition 6. (FJP scheduler) *A scheduler is said to be* FJP *if it assigns a single static priority to each job, but different jobs of the same task may have different priorities.*

Note that FTP schedulers are a strict subset of FJP schedulers. In the remainder of the paper, when considering FTP schedulers, we will denote by $\texttt{hp}(\tau_i)$ the set of tasks having a priority higher than τ_i, and by $\texttt{lp}(\tau_i)$ the set of tasks having a priority lower than τ_i.

The following definitions introduce what is considered as being a schedulable mixed-criticality task set.

Definition 7. (Feasible schedule) *A schedule for a scenario of criticality ℓ is feasible if every job $J_{i,k} \,|L_i \geq \ell \; \forall i, k$ executes for $c_{i,k}$ time units between $r_{i,k}$ and $d_{i,k}$.*

Definition 8. ($\mathcal{S}$-Schedulable) *Let $\mathcal{S}$ be a scheduling policy, and τ a mixed-criticality task set. We say τ is $\mathcal{S}$-Schedulable if, for* any *scenario of criticality $\ell \in \mathbb{L}$, $\mathcal{S}$ generates a feasible schedule.*

In the remainder of the paper, we assume that the task sets on which our protocols are applied have been deemed $\mathcal{S}$-schedulable by a preliminary offline analysis stage.

3. MOTIVATION

In mixed-criticality scheduling, most of the work is built on a common assumption: *the platform π is not powerful enough to meet all the task deadlines if they all run concurrently and execute for their WCET at their own criticality level.* As a result, most methods and analysis tools presented in the literature share a common view on how the tasks are handled at runtime: an "overrun handler" is in charge of monitoring the execution of jobs and increases the current criticality level CL of the module at run-time, as described in Section 1.

If the current criticality level CL of the module can be increased dynamically at run-time, it should also be possible to decrease CL. Otherwise, if no action is taken to reduce the criticality level CL, the system will eventually be running in the highest level L, thus executing *only* the most critical tasks. Hence, the lower criticality tasks will never be executed anymore, even though there may be sufficient spare processing capacity available to execute them. This leads to inefficient usage of the computing resources. We therefore believe that the SCR problem is a crucial challenge in improving mixed criticality systems, as the overall performance of the system and its quality of service is likely to be better if the system is executing more tasks in a lower CL, rather than a subset of tasks in a "safe operating mode".

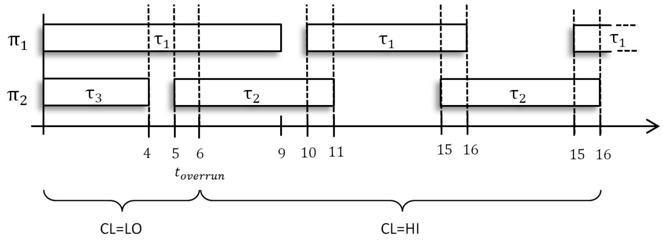

Figure 1: An example to motivate the problem under consideration.

In this work, our focus is to design an approach to determine the time instants at which CL can be reduced.

On a uni-processor computing platform, decreasing the level of the system from its current criticality level h to a criticality level $\ell < h$ at a time instant t when the processor is idle has been shown to work well [20]. However, a straightforward adaptation of this rule on multiprocessors, i.e., lowering the criticality level of the system at a time instant t when all the processors are idle is not efficient. This is because there are very few time instants at which *all* the processors are idle. It can even be shown that such a time instant t may not even exist, even for a simple dual-criticality system, as illustrated by Example 1.

Example 1. *Consider a mixed-criticality system with only two levels, $\mathbb{L} = \{\text{LO}, \text{HI}\}$. Let the task set $\tau = \{\tau_1, \tau_2, \tau_3\}$ consists of three implicit-deadline sporadic tasks with the following parameters:*

$$\tau_1 = \langle \text{HI}, [6, 9], 10, 10\rangle$$
$$\tau_2 = \langle \text{HI}, [6, 9], 10, 10\rangle$$
$$\tau_3 = \langle \text{LO}, [3, 3], 10, 10\rangle$$

Consider an identical multiprocessor platform $\pi = \{\pi_1, \pi_2\}$ with two processors. Let us consider a FTP scheduler where task τ_1 has the highest priority, task τ_3 the lowest priority, and task τ_2 has an intermediate priority. Let us assume that the first job of τ_1 is released at time $t = 0$ and its subsequent jobs are released periodically after every 10 time units. Similarly, let the first job of task τ_2 be released at time $t = 5$ and all its subsequent jobs be released exactly 10 time units apart. Finally, let the first job of task τ_3 be released at time $t = 0$ and all its subsequent jobs are released periodically after every 10 time units. The scenario illustrated by Figure 1 shows that a time instant at which both π_1 and π_2 are idle never occurs even though all the jobs released after time $t = 9$ respect their WCET at the lowest criticality level. Therefore, task τ_3 might never be reactivated.

Example 1 shows that the problem under consideration is nontrivial and in this work, we propose efficient protocols to decide when to reduce the criticality level CL of the module without jeopardizing its schedulability. Upon the re-enablement of the previously suspended tasks, both protocols make sure that no task in the system is affected anymore by the overrun that occurred previously. Section 4 describes a protocol with a negligible run-time overhead which can be applied to any FTP scheduler. Section 5 describes a protocol with a higher run-time overhead but which can be applied to a wider family of FJP schedulers.

In the following sections, we assume that:

- At least one task overruns, causing CL to reach criticality level h.
- We wish to decrease CL from level h to level $\ell < h$.

4. A PROTOCOL FOR FTP SCHEDULERS

4.1 Background Results

In [19], a method for computing an upper-bound on the WCRT of a mixed-criticality task τ_i is described. This method consists in considering the execution of a job $J_{i,k}$ released by τ_i in a time window starting at $J_{i,k}$'s release time $r_{i,k}$, and finishing Δ time units later, i.e. at time instant $r_{i,k} + \Delta$. The procedure aims at defining an upper-bound on the *workload*, the *interfering workload*, the *total interfering workload*, and the *interference* (as explained in the following definitions) suffered by task τ_i in any scenario of criticality level $\ell \in \mathbb{L}$ during that interval. The WCRT of τ_i at criticality level $\ell \in \mathbb{L}$ is then deduced from the previously computed values.

Definition 9. (Workload [19]) *The workload of task τ_j within a time window of size Δ is the cumulative length of time intervals during which the jobs released by τ_j execute within that window.*

A task $\tau_j \in \mathtt{hp}(\tau_i)$ is considered as a *carry-in* task if τ_j released a job before the start of the window, and that job executes (partly or fully) within the window. Otherwise, τ_j is considered as a *non carry-in* task. Pathan [19] showed that if a higher-priority task is a carry-in task, then its worst-case interference on the lower-priority task is higher than if it was non carry-in.

Definition 10. (Carry-in/Non Carry-In Interfering Workload [19]) *The carry-in interfering workload $\bar{\mathtt{I}}^{\mathtt{CI}}_{j,i}(\Delta, \ell)$ (resp. non carry-in interfering workload $\bar{\mathtt{I}}^{\mathtt{NC}}_{j,i}(\Delta, \ell)$) of τ_j on task τ_i in any scenario of criticality level ℓ, is the cumulative length of time interval within the time window of size Δ during which a job released by a carry-in task τ_j (resp. a non carry-in task τ_j) executes, and $J_{i,k}$ is not dispatched on any processor.*

The difference between the carry-in and non carry-in interfering workload of a task $\tau_j \in \mathtt{hp}(\tau_i)$ will be denoted by:

$$\bar{\mathtt{I}}^{\mathtt{DIFF}}_{j,i}(\Delta, \ell) \overset{\text{def}}{=} \bar{\mathtt{I}}^{\mathtt{CI}}_{j,i}(\Delta, \ell) - \bar{\mathtt{I}}^{\mathtt{NC}}_{j,i}(\Delta, \ell) \quad (1)$$

The following property shows that it is not necessary to consider *all* tasks as being carry-in tasks.

Property 1. (From [8] and [19]) *The total interfering workload is upper-bounded by considering at most $m - 1$ (recall that m is the number of processors) carry-in tasks within the time window of any lower priority task, when considering global FTP scheduling of constrained-deadline sporadic task sets.*

Definition 11. (Total Interfering Workload [19]) *The total interfering workload $\bar{\mathtt{I}}_i(\Delta, \ell)$ of tasks $\tau_j \in \mathtt{hp}(\tau_i)$ on task τ_i in any scenario of criticality level ℓ, over any time window of size Δ, is the sum of interfering workload of all the higher priority tasks within that window. The total interfering workload suffered by task τ_i is computed as follows:*

$$\bar{\mathtt{I}}_i(\Delta, \ell) \overset{\text{def}}{=} \sum_{\tau_j \in \mathtt{hp}(\tau_i)} \bar{\mathtt{I}}^{\mathtt{NC}}_{j,i}(\Delta, \ell) + \sum_{\tau_j \in \mathtt{hp}_{m-1}(\tau_i)} \bar{\mathtt{I}}^{\mathtt{DIFF}}_{j,i}(\Delta, \ell) \quad (2)$$

where $\mathtt{hp}_{m-1}(\tau_i)$ is the set of at most $m - 1$ carry-in tasks belonging to $\mathtt{hp}(\tau_i)$ that have the largest value of $\bar{\mathtt{I}}^{\mathtt{DIFF}}_{j,i}(\Delta, \ell)$.

Definition 12. (Interference [19]) *The interference suffered by a task τ_i in any scenario of criticality level ℓ, and*

Algorithm 1: FTP Protocol

```
$f_0^{\checkmark} := t_{\texttt{overrun}}$;
for $i$=1,...,$n$ do
    if $\tau_i$ has an active job $J_{i,k}$ at time $f_{i-1}^{\checkmark}$ then
        $f_i^{\checkmark} := f_{i,k}$;
    else
        $f_i^{\checkmark} := f_{i-1}^{\checkmark}$;
    end
end
```

during a time interval of length Δ, is the cumulative length of time intervals during which the m processors are busy executing tasks belonging to $\texttt{hp}(\tau_i)$. An upper-bound on the interference suffered by τ_i over an interval of length is given by $\left\lfloor \frac{\bar{\mathtt{I}}_i(\Delta,\ell)}{m} \right\rfloor$.

Finally, and from the above definitions, since in any scenario of criticality level ℓ, $J_{i,k}$ is allowed to execute for at most $C_i(\ell)$ time units, the WCRT $R_i(\ell)$ of task τ_i at criticality level ℓ is obtained by determining the least fixed point of the following function:

$$R_i(\ell) \overset{\text{def}}{=} C_i(\ell) + \left\lfloor \frac{\bar{\mathtt{I}}_i(R_i(\ell),\ell)}{m} \right\rfloor \tag{3}$$

Note that since the task set is $\mathcal{S}$-schedulable (see Section 2), it must be the case that $R_i(\ell) \leq D_i \ \forall \ell \in \mathbb{L}$. In the remainder of this section, we will denote by $\mathtt{I}_i^*(\Delta,\ell) \leq \bar{\mathtt{I}}_i(\Delta,\ell)$ the *actual* total interfering workload suffered by τ_i over an interval of length Δ. Moreover, we assume that the tasks in τ are prioritized in the order of their indices. That is, if $i < j$ then τ_i has a higher priority than τ_j.

4.2 Description of the Protocol

The first protocol, formalized by Algorithm 1, and whose correctness is proved in Section 4.4, works as follows: Suppose an overrun occurs at time $t_{\texttt{overrun}}$ and that no job exceeds its WCET at criticality level ℓ after $t_{\texttt{overrun}}$. For every task τ_i in a decreasing order of priority, the protocol identifies a time instant $f_i^{\checkmark}$ satisfying Condition 1.

Condition 1. *Time $f_i^{\checkmark}$ is such that:*

1. *$f_i^{\checkmark} \geq f_{i-1}^{\checkmark}$ (with $f_0^{\checkmark} = t_{\texttt{overrun}}$);*
2. *τ_i has no active job at time $f_i^{\checkmark}$.*

Then, as soon as such an instant has been found for the lowest priority task τ_n, the criticality of the system can safely be decreased to level ℓ and all the suspended tasks with a criticality greater than or equal to ℓ can be reactivated.

More precisely, starting at time $t_{\texttt{overrun}}$, the protocol (Algorithm 1) identifies the earliest time instant $f_1^{\checkmark}$ satisfying Condition 1. That is, the protocol checks whether task τ_1 (the highest priority task belonging to τ) has an active job $J_{1,k}$ at time $t_{\texttt{overrun}}$. If it is the case, then the protocol waits for that job to complete its execution at time $f_1^{\checkmark} = f_{1,k}$. Otherwise, $f_1^{\checkmark} = t_{\texttt{overrun}}$. The protocol then looks for the earliest instant after $f_1^{\checkmark}$ where τ_2 has no active job (i.e., the earliest instant $f_2^{\checkmark} \geq f_1^{\checkmark}$ satisfying Condition 1). Again, if at time $f_1^{\checkmark}$, τ_2 has no active job, then $f_2^{\checkmark} = f_1^{\checkmark}$. Otherwise, the protocol simply waits for the active job of τ_2 to complete. These steps are iteratively performed by Algorithm 1 for every task $\tau_i \in \tau$ in their priority order (i.e. it identifies such a time instant for task τ_i only when it has previously identified such a time instant for task τ_{i-1}). The re-enablement of the previously suspended tasks having a criticality at least equal to ℓ can then take place when an instant $f_n^{\checkmark}$ satisfying Condition 1 has been found.

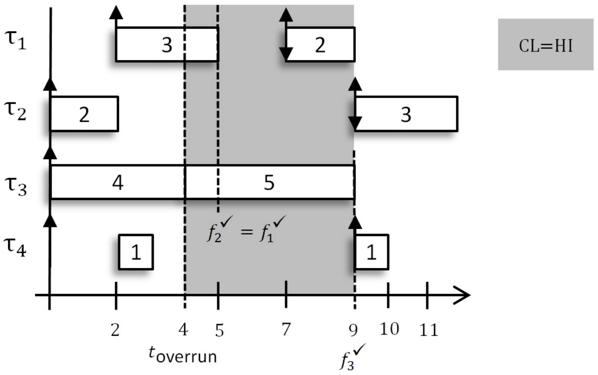

Figure 2: An example to illustrate the protocol described in Section 4.2.

Note that our reasoning is based on the assumption that an overrun is unusual. In particular, we make the assumption that if the protocol identified a time instant satisfying Condition 1 for every task $\tau_1, ..., \tau_i$, then these tasks will not overrun their WCET at level ℓ until the protocol was able to decrease CL from level h to level ℓ. If this happens nevertheless, then the protocol is aborted and has to restart all over from the beginning.

Moreover, in the remainder of the section, we assume that if the protocol identifies a time instant $f_i^{\checkmark}$ satisfying Condition 1 for task τ_i, then this means that the protocol has already identified such a time instant for every tasks $\tau_1, \tau_2, ..., \tau_{i-1}$.

4.3 An Example

Consider a mixed-criticality system with only two criticality levels, $\mathbb{L} = \{\text{LO}, \text{HI}\}$. Let the task set $\tau = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ consists of four implicit-deadline sporadic tasks with the following parameters:

$$\begin{aligned}
\tau_1 &= \langle \text{HI}, [4,5], 5, 5 \rangle \\
\tau_2 &= \langle \text{HI}, [3,4], 9, 9 \rangle \\
\tau_3 &= \langle \text{HI}, [4,9], 11, 11 \rangle \\
\tau_4 &= \langle \text{LO}, [1,1], 5, 5 \rangle
\end{aligned}$$

Consider an identical multiprocessor platform $\pi = \{\pi_1, \pi_2\}$ with two processors. Let us use a FTP scheduler where tasks are prioritized according to their indices (i.e., task τ_1 has the highest priority and task τ_4 the lowest) and consider the scenario illustrated by Figure 2 where tasks τ_2, τ_3 and τ_4 release a job at time $t = 0$, and task τ_1 releases a job at time $t = 2$. At time $t = 4$, τ_3 does not signal its completion, which results in CL being increased from level LO to level HI. Task τ_4 is therefore suspended. At time $t = 4$, the protocol is launched and checks whether task τ_1 has an active job. Since the first job of task τ_1 is still executing, the protocol must wait for that job to finish at time $t = 5$. Since at time $t = 5$, task τ_2 has no active job, the protocol can skip τ_2 and consider τ_3 instead. Since at time $t = 5$, τ_3 has an active job, the protocol must wait for that job to finish at time $t = 9$. Because τ_3 is the lowest priority task among the tasks in $\tau^{(\text{HI})}$, it follows that no other task in τ will have an active job (the lower criticality tasks have been suspended at time $t = 4$). Thus, at time $t = 9$, the criticality of the system can be decreased to level LO, thereby reactivating the task τ_4.

4.4 Proof of Correctness

Lemma 1. *Let $t_{\texttt{overrun}}$ be the last time at which a job overran its WCET at criticality level ℓ. Let us assume that the protocol identifies an instant $f_{i-1}^{\checkmark} \geq t_{\texttt{overrun}}$ respecting Condition 1 for task τ_{i-1} and no job with an execution time greater than its WCET at criticality level ℓ is released after $t_{\texttt{overrun}}$. From time $f_{i-1}^{\checkmark}$ onwards, the actual total interfering workload $\mathtt{I}_i^*(\Delta, \ell)$ suffered by task τ_i over any window of size Δ is less than or equal to $\bar{\mathtt{I}}_i(\Delta, \ell)$.*

Proof. From Definition 11, we know that $\bar{\mathtt{I}}_i(\Delta, \ell)$ is an upper-bound on the total interfering workload suffered by task τ_i in any scenario of criticality level ℓ. More precisely, we have $\mathtt{I}_i^*(\Delta, \ell) \leq \bar{\mathtt{I}}_i(\Delta, \ell)$ if no job released by a task in $\mathtt{hp}(\tau_i)$ within the interval Δ has an execution time greater than its WCET at criticality level ℓ. We now prove that this is the case from time $f_{i-1}^{\checkmark}$ onwards.

The proof is by induction on the task priorities. In particular, assuming that Lemma 1 holds for task τ_{i-1}, we prove that it also holds for task τ_i.

Base case. Let us consider that τ_{i-1} is the highest priority task (i.e. $\tau_{i-1} = \tau_1$), and that $\tau_i = \tau_2$. It follows that the jobs released by task τ_1 are the only jobs interfering with the execution of task τ_i from time $f_{i-1}^{\checkmark}$ onwards. But since task τ_1 has no active job at time $f_1^{\checkmark}$, and because we assumed that all the jobs released by τ_1 after $f_1^{\checkmark}$ have an execution time that is less than or equal to their WCET at level ℓ, it follows that all the jobs interfering with the execution of τ_2 from time $f_1^{\checkmark}$ onwards have an execution time that is less than or equal to their WCET at level ℓ. Hence, $\mathtt{I}_2^*(\Delta, \ell) \leq \bar{\mathtt{I}}_2(\Delta, \ell)$ for any window of size Δ starting at time $f_1^{\checkmark}$.

Inductive step. The jobs released by the task in $\mathtt{hp}(\tau_i)$ are the only ones interfering with the execution of task τ_i from time $f_{i-1}^{\checkmark}$ onwards. Because Algorithm 1 considers the tasks in their priority order, we know from Condition 1 that for every task in $\tau_j \in \mathtt{hp}(\tau_i)$, $f_j^{\checkmark} \leq f_{i-1}^{\checkmark}$. Furthermore, by the induction hypothesis, it holds that no job with an execution time greater than its WCET at criticality level ℓ interferes with the execution of τ_j after $f_j^{\checkmark}$, and thus after $f_{i-1}^{\checkmark}$. Furthermore, because we made the assumption that no job released by a task after $f_{i-1}^{\checkmark} \geq t_{\texttt{overrun}}$ exceeds its WCET at level ℓ, it results that all the jobs interfering with the execution of τ_i after $f_{i-1}^{\checkmark}$ have an execution time not greater than their WCET at level ℓ. Hence, $\mathtt{I}_i^*(\Delta, \ell) \leq \bar{\mathtt{I}}_i(\Delta, \ell)$ for any window of size Δ starting at time $f_{i-1}^{\checkmark}$. □

Corollary 1. *Let $t_{\texttt{overrun}}$ be the last time at which a job overran its WCET at criticality level ℓ. When the protocol identifies an instant $f_n^{\checkmark} \geq t_{\texttt{overrun}}$ satisfying Condition 1 for task τ_n, then from time $f_n^{\checkmark}$ onwards, the actual total interfering workload suffered by any task $\tau_i \in \tau$ is less than or equal to $\bar{\mathtt{I}}_i(\Delta, \ell)$.*

Proof. Since, by definition of the highest priority task, the highest priority task τ_1 never suffers from any interference from lower priority tasks, this corollary is obviously true for τ_1. Furthermore, because according to Algorithm 1, we have $f_i^{\checkmark} \leq f_n^{\checkmark}$ for $1 \leq i \leq n$, this corollary directly follows from Lemma 1 for any task τ_i such that $1 < i \leq n$. □

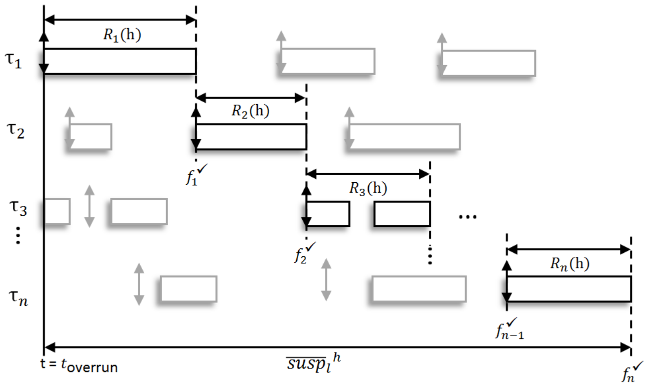

Figure 3: Upper-bound on the re-enablement of suspended tasks.

From Corollary 1, we will now show that every suspended task $\tau_k \in \tau^{(\ell)}$ can be re-enabled at time $f_n^{\checkmark}$.

Theorem 1. *Upon an overrun at time $t_{\texttt{overrun}}$, when the protocol identifies an instant $f_n^{\checkmark} \geq t_{\texttt{overrun}}$ satisfying Condition 1 for the lowest priority task $\tau_n \in \tau$, then every suspended task $\tau_k \in \tau^{(\ell)}$ can be re-enabled.*

Proof. From Corollary 1, we know that from time $f_n^{\checkmark}$ onwards, every job $J_{i,k}$ released by a task $\tau_i \in \tau^{\ell}$ will suffer a total interfering workload that is not greater than $\bar{\mathtt{I}}_i(\Delta, \ell)$ in any window of size Δ. Hence, from Expression 3, every job will completes its execution after no more that $R_i(\ell)$ time units. Furthermore, since the system was deemed $\mathcal{S}$-schedulable, we have that $R_i(\ell) \leq D_i\ \forall i$. Consequently, every task $\tau_i \in \tau^{\ell}$ can be safely re-enabled. □

4.5 Upper Bound on the Suspension Delay

In this section, we prove that using the protocol presented in Section 4.2, it is possible to upper-bound the suspension delay suffered by lower criticality tasks. This means that even though a task can exceptionally overrun its WCET at a specific criticality level ℓ, the system will eventually be able to recover from the generated overload.

Theorem 2. *Let us assume that an overrun occurs at time $t_{\texttt{overrun}}$, i.e. a task τ_i with $L_i > h$ exceeds its WCET at criticality $h - 1$, thus causing the system to increase* CL *to h. If no other task overran its WCET at criticality level $\ell < h$ after $t_{\texttt{overrun}}$, then an upper-bound $\overline{\mathtt{susp}}_\ell^h$ on the time required to re-enable the suspended tasks $\tau_i \in \tau^{(\ell)}$ is given by*

$$\overline{\mathtt{susp}}_\ell^h = \sum_{\tau_i \in \tau^{(h)}} R_i(h) \qquad (4)$$

Proof. Since all the tasks that have a criticality smaller than h are suspended at time $t_{\texttt{overrun}}$, it follows that the only tasks that may have an active job within the time interval $\left[t_{\texttt{overrun}}, t_{\texttt{overrun}} + \overline{\mathtt{susp}}_\ell^h\right)$ are the ones in $\tau^{(h)}$. Therefore, from line 6 of Algorithm 1, we have:

$$\forall \tau_i \in \tau \setminus \tau^{(h)},\ f_i^{\checkmark} = f_{i-1}^{\checkmark} \qquad (5)$$

Furthermore, in the worst-case, each task $\tau_i \in \tau^{(h)}$ released a job right at time $f_{i-1}^{\checkmark}$. In a scenario of criticality level h, this job could finish its execution at most $R_i(h)$ time units later (i.e., the response time of the job is at most equal to its WCRT at criticality level h). Hence, from line 4 of

Algorithm 1, we have:

$$\forall \tau_i \in \tau^{(h)},\ f_i^{\checkmark} \leq f_{i-1}^{\checkmark} + R_i(h) \qquad (6)$$

Because all the suspended tasks in $\tau^{(\ell)}$ can be re-enabled when an instant $f_n^{\checkmark}$ is found for task τ_n, $\overline{\texttt{susp}}_\ell^h$ is an upper-bound on $f_n^{\checkmark}$. Using Equations 5 and 6, we get that $f_n^{\checkmark} \leq \sum_{\tau_i \in \tau^{(h)}} R_i(h)$, thereby leading to:

$$\overline{\texttt{susp}}_\ell^h = \sum_{\tau_i \in \tau^{(h)}} R_i(h)$$

□

An illustration of the computation of the upper-bound $\overline{\texttt{susp}}_\ell^h$ is given by Figure 3.

5. A PROTOCOL FOR FJP SCHEDULERS

5.1 Background Results

5.1.1 *Reference schedule*

The concept of reference schedule was first introduced in [2] for uniprocessor platforms, and later extended to identical multicore platforms in [18]. In these works [2,18], a reference schedule is used to detect the time instants at which the speed of the cores could safely be reduced (in order to save energy without missing any deadlines). In our work, we use the concept of reference schedule at each criticality level $\ell \in \mathbb{L}$, as explained in Definition 13, to detect the time instants at which the criticality level of the system can be lowered.

Definition 13. (Reference schedule at level ℓ) *The reference schedule at criticality level ℓ for a task set τ considering a scheduling algorithm $\mathcal{S}$, is the schedule generated by $\mathcal{S}$ for the jobs released by τ under the assumption that all these jobs execute exactly for their WCET at level ℓ.*

Let us consider that an algorithm $\mathcal{S}$ is used to schedule a mixed-criticality task set τ. At run-time, if the current criticality level of the module is ℓ, then $\mathcal{S}$ will schedule every task τ_i with $L_i \geq \ell$ under the assumption that none of these tasks will exceed its WCET at criticality level ℓ. Yet, at some point in time, a task τ_i with $L_i > \ell$ might overrun, thus causing the system to increase the criticality level from ℓ to h. In that case, the algorithm $\mathcal{S}$ drops all the tasks with their criticality level lower than h, and only schedules those that have a criticality level at least equal to h. It follows that the *actual* schedule produced by $\mathcal{S}$ is different from the *reference* schedule that would have been produced by $\mathcal{S}$ if no task had exceeded its WCET at criticality level ℓ.

The motivation for having such a (reference) schedule which the system can refer to is the following: if at some point in time, the actual schedule diverges from the expected reference schedule, the system can compare both schedules to determine when the system is "back-on-track", i.e. when the system has recovered from an occasional overrun by a higher criticality task. Since we consider L different criticality levels, the system has to keep track of L reference schedules, as depicted by Figure 4.

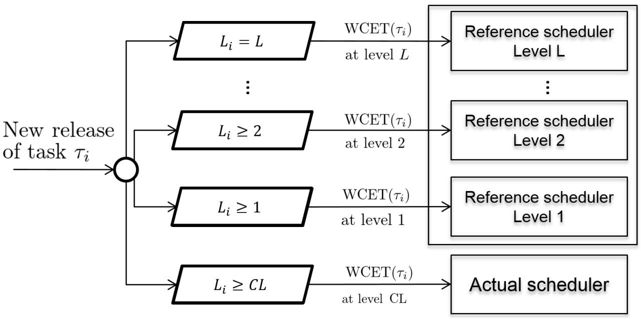

Figure 4: L reference schedulers, one for each criticality level.

5.1.2 *Predictability*

In this section, we describe the well-known concept of predictability and prove some of its properties (Lemma 3 and Lemma 4) that will be used later to prove the correctness of our protocol (Theorem 3 in Section 5.3). First, we introduce some of the terms that are extensively used in the rest of the section.

Definition 14. (Traditional Task Set) *A traditional task set is a task set in which each task τ_i is specified by a 3-tuple $\langle C_i, T_i, D_i \rangle$ where:*

- *$C_i \in \mathbb{N}_0$ denotes the WCET of task τ_i;*
- *$T_i \in \mathbb{N}_0$ denotes the minimum inter-arrival time of task τ_i;*
- *$D_i \in \mathbb{N}_0$ denotes the deadline of task τ_i.*

Definition 15. (Worst-Case Scenario of a Traditional Task Set) *The worst-case scenario of a task set is a scenario in which every job released by any task of the task set executes exactly for its worst-case execution time.*

Note that for a sporadic task set, there may be several worst-case scenarios (depending on the release times of the jobs).

We now define the notion of predictability and list some of its properties that are relevant for this work.

Definition 16. (Predictability, from Ha and Liu [11]) *Let $\mathcal{S}$ be a scheduling algorithm, and let $J = \{J_1, J_2, \ldots, J_n\}$ be a set of n jobs, where each job $J_i = (r_i, c_i)$ is characterized by an arrival time r_i and a execution requirement c_i. Let s_i and f_i denote the time at which job J_i starts and completes its execution (respectively) when J is scheduled by $\mathcal{S}$. Now, consider any set $J' = \{J'_1, J'_2, \ldots, J'_n\}$ of n jobs obtained from J as follows. J'_i has an arrival time r_i and an execution requirement $c'_i \leq c_i$ (i.e., job $J'_i \in J'$ has the same arrival time as $J_i \in J$ and an execution requirement no larger than J_i's). Let s'_i and f'_i denote the time at which job J'_i starts and completes its execution (respectively) when J' is scheduled by $\mathcal{S}$. Algorithm $\mathcal{S}$ is said to be predictable if and only if for any set of jobs J and for any such J' obtained from J, it is the case that $s'_i \leq s_i$ and $f'_i \leq f_i$ $\forall i$.*

We also make use of the following result from [9–11].

Lemma 2. (From Ha and Liu [9,11] and [10]) *On identical multiprocessors, any global, preemptive, FJP and work-conserving scheduler is predictable.*

The concept of predictability is important in real-time scheduling theory: using a predictable scheduler, the schedulability of a given traditional task set τ can be deduced from the schedulability of *all* its worst-case scenarios. According to Definition 16, if all these worst-case scenarios are schedulable on the target platform then any other scenario of τ in which jobs execute for less than their WCETs are also schedulable on this platform. This property enables the system designers to verify only the schedulability of all these worst-case scenarios to deduce the schedulability of the whole task set τ under every (other) possible execution scenario. Hence, most of the schedulability analysis techniques base their computations on the parameter C_i (i.e., the worst-case execution time of each task).

Definition 17. (Actual worst-case remaining execution time) *At each time instant t, $\texttt{act-rem}_i(t)$ denotes the actual worst-case remaining execution time of the active job of task τ_i.*

Definition 18. (Reference worst-case remaining execution time) *At each time instant t, $\texttt{ref-rem}_i(t)$ denotes the reference worst-case remaining execution time of the active job of task τ_i, assuming that all the jobs released from the beginning (from time 0) have executed for their WCET.*

Lemma 3. *Let τ be a traditional task set scheduled by a predictable scheduler $\mathcal{S}$ on a platform π. At run-time, suppose that all the deadlines are met. It holds for all τ_i and time-instant t that $\texttt{act-rem}_i(t) \leq \texttt{ref-rem}_i(t)$.*

Proof. The proof is obtained by contradiction. Let t denote the first time-instant such that there exists a task τ_i for which $\texttt{act-rem}_i(t) > \texttt{ref-rem}_i(t)$. Let J denote the collection of jobs that τ has released (at run-time) from time 0 to t. Let $S_{\mathcal{S}}(J)$ denote the schedule of J by $\mathcal{S}$ on the given platform π, i.e. $S_{\mathcal{S}}(J)$ denotes the schedule that has been generated at run-time from time 0 to t.

Let J^{wc} denotes a worst-case scenario corresponding to J such that every job in J^{wc} has the same release time and deadline as the corresponding job in J but has an execution time equal to the WCET of the task that released it. Let $S_{\mathcal{S}}(J^{\text{wc}})$ denote the schedule of the scenario J^{wc} by $\mathcal{S}$ on π.

Let $J_{i,k}$ denote the last job released by task τ_i in $[0, t)$, i.e., $J_{i,k}$ is the job of τ_i for which $\texttt{act-rem}_i(t) > \texttt{ref-rem}_i(t)$. By definition of $J_{i,k}$, it holds that $J_{i,k} \in J$ and $J_{i,k} \in J^{\text{wc}}$ and by definition of J^{wc}, the actual execution time $c_{i,k}$ of $J_{i,k}$ in J is no greater than its corresponding actual execution time in J^{wc}.

Now, let us modify J and J^{wc} by setting $c_{i,k}^{new} = C_i - \texttt{ref-rem}_i(t)$ in both scenarios J and J^{wc}. With this reduced $c_{i,k}^{new}$, it holds that J_ℓ now finishes at time t in $S_{\mathcal{S}}(J^{\text{wc}})$ whereas it finishes after time t in $S_{\mathcal{S}}(J)$ (since it is assumed that $\texttt{act-rem}_i(t) > \texttt{ref-rem}_i(t)$). This clearly contradicts the predictability of $\mathcal{S}$ as the scenario J can be obtained from J^{wc} by applying the same transformation as the one explained in Definition 16, which implies that all the jobs in J should start and finish not later than their corresponding job in J^{wc} when scheduled by $\mathcal{S}$. □

Lemma 4. (Memoryless property) *Let J be an infinite collection of jobs and suppose that J is guaranteed to meet all the deadlines when scheduled by a predictable algorithm $\mathcal{S}$ on a platform π. Similarly, let J' be another infinite collection of jobs and suppose that J' is guaranteed to meet all the deadlines when scheduled by another predictable algorithm $\mathcal{S}'$ on the same platform π. In the schedule of J by $\mathcal{S}$ (depicted as schedule (a) in Figure 5), let the functions $\texttt{completed}(J,t)$, $\texttt{active}(J,t)$, and $\texttt{future}(J,t)$ denote the subset of jobs of J that are completed at time t, active at time t, and not yet released at time t, respectively. The functions $\texttt{completed}(J',t)$, $\texttt{active}(J',t)$, and $\texttt{future}(J',t)$ are defined analogously for the schedule of J' by $\mathcal{S}'$ (see schedule (b) in Figure 5).*

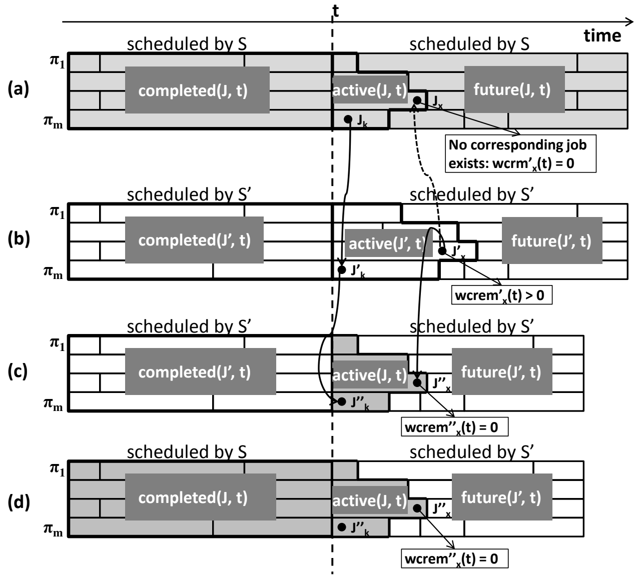

Figure 5: Illustration of the different schedules used in the proof of Lemma 4.

Now, suppose that there exists a time t during the schedules of J and J' (by $\mathcal{S}$ and $\mathcal{S}'$, respectively) at which every job $J_k \in \texttt{active}(J,t)$ can be one-to-one mapped to a job $J'_k \in \texttt{active}(J',t)$ such that J_k and J'_k have the same release time, deadline, and execution requirement, but the remaining execution time $\texttt{rem}_k(t)$ of J_k is less than or equal to that of the corresponding job J'_k, i.e.,

$$\forall J_k \in \texttt{active}(J,t) : \texttt{rem}_k(t) \leq \texttt{rem}'_k(t) \tag{7}$$

Under these assumptions, all the job deadlines will be met in a schedule (see schedule (d) in Figure 5) where:

1. *J is scheduled by $\mathcal{S}$ from time 0 to time t,*
2. *at time t, the scheduler $\mathcal{S}$ is substituted for $\mathcal{S}'$, and*
3. *from time t onward, the next jobs to arrive are those from $\texttt{future}(J',t)$ only.*

Proof. Let us construct the following (infinite) collection of jobs J'' from J' as follows. J'' is composed of:

1. the subsets of jobs $\texttt{completed}(J',t)$ and $\texttt{future}(J',t)$ from J', and
2. a subset of jobs containing one job J''_k for each $J'_k \in \texttt{active}(J',t)$, with same release time and deadline as J'_k but with an execution requirement c''_k equal to $c'_k - (\texttt{rem}'_k(t) - \texttt{rem}_k(t))$. That is, $c''_k \leq c'_k = c_k$. Note that $\texttt{rem}_k(t)$ is assumed to be zero if there is no corresponding job J_k in $\texttt{active}(J,t)$ for the job J'_k (as $\texttt{active}(J',t)$ can contain more jobs than $\texttt{active}(J,t)$).

If J'' is scheduled by $\mathcal{S}'$ on π (see schedule (c) in Figure 5), it holds by construction that at time t, the remaining execution time $\texttt{rem}''_k(t)$ of each job $J''_k \in \texttt{active}(J'',t)$ is equal to $\texttt{rem}_k(t) \leq \texttt{rem}'_k(t)$ (or equal to zero if $\nexists J_k \in \texttt{active}(J,t)$ corresponding to J''_k as explained above). Thus, it holds by Definition 16, from the schedulability of J' and the predictability of $\mathcal{S}'$ that the schedule of J'' by $\mathcal{S}'$ meets all the job deadlines as well.

Now, let us compare the schedule of J'' by $\mathcal{S}'$ ((c) in Figure 5) with the schedule assumed in the claim ((d) in Figure 5). At time t, the sets of active jobs in both schedules ((c) and (d) in Figure 5) are identical (i.e. with equal job remaining execution times) and since from time t onward the two schedules will be in all points identical (same arriving jobs and same scheduling decisions), the claim holds true. □

5.1.3 Alpha-queue

The alpha-queue is a data structure that stores the value of $\texttt{ref-rem}_i(t)$ for each task $\tau_i \in \tau$ at any time t during the scheduling of a traditional task set τ. Remember that $\texttt{ref-rem}_i(t)$ denotes the worst-case remaining execution time of task τ_i in the schedule of all the jobs released by τ where each job executes for its WCET. To do so, the alpha-queue has to be updated at run-time to keep track of the reference schedule.

From an implementation point of view, an alpha-queue is simply a dynamic list containing one element for each task that has an active job. The element for task τ_i records the value of $\texttt{ref-rem}_i(t)$ for the current time t. The alpha-queue for a traditional task set is updated as follows:

R1. Upon the arrival of a job of task τ_i at time t, the alpha-queue creates an element for task τ_i and sets this entry to C_i , i.e. $\texttt{ref-rem}_i(t) = C_i$.

R2. At any time t, the alpha-queue is sorted by decreasing order of job priorities, with the m highest priority jobs (elements) at the head of the queue.

R3. As time elapses, the m elements $\texttt{ref-rem}_i(t)$ (if any) at the head of the alpha-queue are decremented. Whenever one element reaches zero, the element is removed from the alpha-queue and the update continues, with the new m highest priority elements (if any). No update is performed if the alpha-queue is empty.

For the same reasons as described in [2], the following observation holds: At any time t, the alpha-queue updated according to the rules R1–R3 contains only those jobs that have not yet finished execution (*unfinished jobs*) at time t. Moreover, $\texttt{ref-rem}_i(t)$ fields contain the worst-case remaining execution times of every unfinished job at time t in that reference schedule. It also holds, as explained in [2], that the dynamic update of the $\texttt{ref-rem}_i(t)$ fields do not need to be performed at each and every time unit. Instead, for efficiency, the update can be performed only *on-demand*, i.e. only at time instants that are of interest (by taking into account the time elapsed since the last update).

5.2 Description of the Protocol

The protocol for FJP schedulers, whose correctness is proved in Section 5.3, works as follows: During system execution, the protocol updates for every task τ_i a variable $\texttt{Q}_i(t)$ that depends on the current time t. The variable $\texttt{Q}_i(t)$ records the duration of time for which the last released job of task τ_i has been executed up to time t. Let the current criticality level be denoted by $curr$. At every time instant t, if there exists a task τ_j such that $\texttt{Q}_j(t) > C_j(curr)$ (the task is overrunning at the current level), then the system switches to the next (higher) criticality level $h = curr + 1$. Otherwise, the system computes the worst-case remaining execution time at level ℓ (where ℓ can be $curr - 1$ or any lower level) as $\texttt{act-rem}_i(t, \ell) = C_i(\ell) - \texttt{Q}_i(t)$ of every task τ_i with $L_i \geq curr$. The system switches to the lower criticality level ℓ if the following two sets of conditions are satisfied:

COND1. $\forall \tau_i$ with $L_i \geq curr$: $\texttt{act-rem}_i(t, \ell) \geq 0$ (the task is not overrunning at level ℓ) and

COND2. $\forall \tau_i$ with $L_i \geq curr$: $\texttt{act-rem}_i(t, \ell) \leq \texttt{ref-rem}_i(t, \ell)$

If any of the above conditions is not satisfied then the system continues in its current criticality level, $curr$.

The quantities $\texttt{ref-rem}_i(t, k)$, where $k \in \{1, 2, \ldots, L\}$, are obtained from L different alpha-queues (one for each level of criticality) updated at run-time. The alpha-queue corresponding to level k contains one element for each active task defined at level k. The element for task τ_i at level k records the value of $\texttt{ref-rem}_i(t, k)$ for the current time t. These L alpha-queues for mixed-criticality task sets are updated as follows (*mixed-criticality variants* of rules R1–R3):

MCR1. Upon the arrival of a job of τ_i at time t, each alpha-queue k with $k \leq L_i$ creates an element for task τ_i and sets this element to $C_i(k)$, i.e. $\texttt{ref-rem}_i(t, k) = C_i(k)$.

MCR2. At any time t, the L alpha-queues are sorted by decreasing order of the job priorities, with the m highest priority jobs at the head of the queue.

MCR3. As time elapses, the m elements $\texttt{ref-rem}_i(t, k)$ (if any) at the head of each alpha-queue k are decremented. Whenever one element reaches zero, the element is removed from the alpha-queue and the update continues, still with the m highest priority elements (if any). Obviously, no update is performed on an alpha-queue that is empty.

5.3 Proof of Correctness

Theorem 3. *Consider a task set τ which is deemed $\mathcal{A}$-schedulable on a platform π by a predictable algorithm $\mathcal{A}$. At any time t during the scheduling of τ by $\mathcal{A}$ at the criticality level $curr$, it is safe to switch back to the lower criticality level $\ell < curr$ if, for all task $\tau_i \in \tau$ with $L_i \geq curr$:*

$$\texttt{act-rem}_i(t, \ell) \geq 0 \qquad \textit{and} \tag{8}$$

$$\texttt{act-rem}_i(t, \ell) \leq \texttt{ref-rem}_i(t, \ell) \tag{9}$$

Proof. The safety of this approach follows from the memory-less property proven in Lemma 4. Expression 8 asserts that, at time t, no task running at criticality level $curr$ has executed for more than its worst-case execution time at level ℓ (otherwise the task would be overrunning at level ℓ and the system must continue at the current level). Then, it can be seen that in Lemma 4, by defining (i) J as the set of jobs of tasks with criticality level $curr$, (ii) J' as the set of jobs of tasks with criticality level ℓ and (iii) both $\mathcal{S}$ and $\mathcal{S}'$ as algorithm $\mathcal{A}$, Lemma 4 is applicable to the situation described in the claim (since Expression 9 is equivalent to Expression 7). Therefore, switching from criticality level $curr$ to the lower criticality level ℓ at time t when Expressions 8 and 9 are satisfied does not jeopardize the schedulability of the system. Hence the proof. □

6. DISCUSSION AND CONCLUSIONS

We studied the problem of deciding when to lower the criticality level of a multiprocessor mixed-criticality system, so that all the suspended tasks from that (lower) criticality level can resume execution without jeopardizing the schedulability of the system. We proposed two protocols, one of which could be applied to any fixed-task priority scheduler, and one of which could be applied to any fixed-job priority scheduler. For the first protocol, we also provided an upper-bound on the suspension delay suffered by the lower criticality tasks. Both protocols are independent of the number of criticality levels and the number of processors. To the best of our knowledge, this work presents the first solutions to the problem of safe criticality reduction on multiprocessor platforms.

For convenience, throughout the paper, we explained our protocols for global mixed-criticality scheduling algorithms. However, it is trivial to see that the proposed protocols also work for partitioned scheduling, as long as the underlying (mixed-criticality) scheduling algorithm is fixed job priority and work conserving. Both protocols can be applied independently on each processor. Analogously, for both protocols, depending on the outcome, the criticality level is reduced from h to ℓ either (i) on each processor or (ii) only on those processors in which the respective conditions are satisfied for lowering the criticality level. Observe that, for partitioned scheduling, the increase in the run-time overhead is negligible for the first protocol compared to the second protocol. This is because the second protocol now has to maintain L alpha queues on each processor and needs to update all these queues at run-time.

Acknowledgments

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within projects ref. FCOMP-01-0124-FEDER-022701 (CISTER) and ref. FCOMP-01-0124-FEDER-020447 (REGAIN); by National Funds through FCT and by the EU ARTEMIS JU funding, within grant nr. 333053 (CONCERTO) and grant nr. 295371 (CRAFTERS); by ERDF, through ON2 - North Portugal Regional Operational Programme, under the National Strategic Reference Framework (NSRF), within project ref. NORTE-07-0124-FEDER-000063 (BEST-CASE).

7. REFERENCES

[1] Avionics application software standard interface: Part 1 - required services (ARINC specification 653-2). Technical report, Avionics Electronic Engineering Committee (ARINC), March 2006.

[2] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Trans. Comput.*, 53(5):584–600, May 2004.

[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 *ECRTS 2012*, pages 145–154.

[4] 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 *ESA 2011*, pages 555–566.

[5] S. K. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *RTSS 2011*, pages 34–43.

[6] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *RTSS 2009*, pages 291–300, 2009.

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

[8] N. Guan, M. Stigge, W. Yi, and G. Yu. New response time bounds for fixed priority multiprocessor scheduling. In *RTSS 2009*, pages 387–397.

[9] R. Ha. *Validating Timing Constraints in Multiprocessor and Distributed Systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[10] R. Ha and J. W. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1993.

[11] R. Ha and J. W. S. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *ICDCS 1994*.

[12] ISO/TC22. ISO26262: Road Vehicules - Functional Sagety. Technical report, International Organization for Standardization, 2011.

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

[14] H. Li and S. Baruah. Outstanding paper award: Global mixed-criticality scheduling on multiprocessors. In *ECRTS 2012*, pages 166–175.

[15] H. Li and S. K. Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *RTSS 2010*, pages 183–192.

[16] H. Li and S. K. Baruah. Global mixed-criticality scheduling on multiprocessors. In *ECRTS 2012*, pages 166–175.

[17] H. Li and S. K. Baruah. Load-based schedulability analysis of certifiable mixed-criticality systems. In *EMSOFT 2010*, pages 99–108.

[18] V. Nelis and J. Goossens. Mora: An energy-aware slack reclamation scheme for scheduling sporadic real-time tasks upon multiprocessor platforms. In *RTCSA 2009*, pages 210–215.

[19] R. M. Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *ECRTS 2012*, pages 309–320.

[20] F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with FP. In *ECRTS 2012*, pages 155–165.

[21] H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *DATE 2013*, pages 147–152.

[22] F. A. A. United States. DO-178B: Software Considerations in Airborne Systems and Equipment Certification. Technical report, Radio Technical Commission for Aeronautic, 1992.

[23] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS 2007*, pages 239–243.

[24] F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quinones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, T. V. L. Cucu, and F. Cazorla. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *SIES 2013*, pages 241–248.