Technical Report

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

François Santy
Geoffrey Nelissen
Pratyush Kumar
Eduardo Tovar

Gurulingesh Raravi
Vincent Nelis
Joël Goossens

CISTER-TR-131004
Version:
Date: 10-09-2013
Two Protocols to Reduce the Criticality Level of Multiprocessor Mixed-Criticality Systems

François Santy, Gurulingesh Raravi, Geoffrey Nelissen, Vincent Nelis, Pratyush Kumar, Joël Goossens, Eduardo Tovar

CISTER Research Unit
Polytechnic Institute of Porto (ISEP-IPP)
Rua Dr. António Bernardino de Almeida, 431
4200-072 Porto
Portugal
Tel.: +351.22.8340509, Fax: +351.22.8340509
E-mail:
http://www.cister.isep.ipp.pt

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.
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 or 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.

1. INTRODUCTION
Many industrial domains such as avionics, automobiles, 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 time (WCET) 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 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).
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 modelling of real-time tasks is particularly suited to functionallities being subject to mandatory certification, as certification processes often impose a specific maximum probability of failure per criticality level [12, 22].

2 - 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 $\tau_i$ is characterized by a 4-tuple $\langle L_i, C_i, T_i, D_i \rangle$ where:

- $L_i \in L$ denotes the criticality level of task $\tau_i$;
- $C_i \in N_0^L$ 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 $\tau_i$ at criticality level $\ell \in L$;
- $T_i \in N_0$ denotes the minimum inter-arrival time of task $\tau_i$;
- $D_i \in N_0$ denotes the deadline of task $\tau_i$, with $D_i \leq T_i$.

We assume $C_i(\ell)$ is monotonically increasing for increasing values of $\ell$. More precisely, for task $\tau_i$ 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 $r^{k}_{j,\ell}$, with $k \in L$, to denote the set of tasks having criticality level greater than or equal to $k$.

Each task $\tau_i$ 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 $\tau_i$, an instance of $\tau_i$. The $k^{th}$ job $J_{i,k}$ released by the mixed-criticality task $\tau_i$ is characterized by a 3-tuple of parameters $\langle r_{i,k}, d_{i,k}, c_{i,k} \rangle$ where:

- $r_{i,k} \in N$ denotes the release time of job $J_{i,k}$;
- $d_{i,k} \in N_0$ denotes the absolute deadline of job $J_{i,k}$ and is given by $d_{i,k} \overset{\text{def}}{=} r_{i,k} + D_i$;
- $c_{i,k} \in N_0$ denotes the actual execution time of job $J_{i,k}$.

From the specifications of $\tau_i$, 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 $s_{i,k} \in N$ and $f_{i,k} \in N_0$, 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)** 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 (WCRT))** The worst-case response time (WCRT) $R_i(\ell)$ of task $\tau_i$ at criticality level $\ell$ is the maximum amount of time that elapses between the release of any job $J_{i,k}$ of $\tau_i$ at time $r_{i,k}$, and its completion at time $f_{i,k}$, in any scenario of criticality level $\ell$.

### 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 hp($\tau_i$) the set of tasks having a priority higher than $\tau_i$, and by lp($\tau_i$) the set of tasks having a priority lower than $\tau_i$.

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 $\ell$ 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. ($S$-Schedulable)** Let $S$ be a scheduling policy, and $\tau$ a mixed-criticality task set. We say $\tau$ is $S$-Schedulable if, for any scenario of criticality $\ell \in L$, $S$ 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 $S$-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 $\pi$ 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”.

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 for the following reasons: (i) there are very few time instants at which all the processors are idle; (ii) 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, \( L = \{LO, HI\} \). Let the task set \( \tau = \{\tau_1, \tau_2, \tau_3\} \) consists of three implicit-deadline sporadic tasks with the following parameters:

\[
\begin{align*}
\tau_1 &= (HI, [6, 9], 10, 10) \\
\tau_2 &= (HL, [6, 9], 10, 10) \\
\tau_3 &= (LO, [3, 3], 10, 10)
\end{align*}
\]

Consider an identical multiprocessor platform \( \pi = \{\pi_1, \pi_2\} \) with two processors. Let us consider a FTP scheduler where task \( \tau_1 \) has the highest priority, task \( \tau_2 \) the lowest priority, and task \( \tau_3 \) an intermediate priority. Let us assume that the first job of \( \tau_1 \) is released at time \( t = 0 \) and its subsequent jobs are released periodically after every 10 time units. Similarly, let the first job of \( \tau_2 \) be released at time \( t = 5 \) and all its subsequent jobs be released exactly 10 time units apart. Finally, let the first job of \( \tau_3 \) be released at time \( t = 0 \) and all its subsequent jobs be released periodically after every 10 time units. The scenario illustrated by Figure 1 shows that a time instant at which both \( \tau_1 \) and \( \tau_2 \) are idle never occurs even though all the jobs released after time \( t = 9 \) respect their WCET at the lowest criticality level. Therefore, task \( \tau_3 \) 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 \( \ell \).
- We wish to decrease CL from level \( \ell \) to level \( \ell < \ell \).

### 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 \( \tau_i \) is described. This method consists in considering the execution of a job \( J_{i,k} \) released by \( \tau_i \) in a time window starting at \( J_{i,k} \)’s release time \( r_{i,k} \), and finishing \( \Delta \) 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, and the interference (as explained in the following definitions) suffered by task \( \tau_i \) in any scenario of criticality level \( \ell \in L \) during that interval. The WCRT of \( \tau_i \) at criticality level \( \ell \) is then deduced from the previously computed values.

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

A task \( \tau_j \in \text{hp}(\tau_i) \) is considered as a carry-in task if \( \tau_j \) released a job before the start of the window, and that job executes (partly or fully) within the window. Otherwise, \( \tau_j \) 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 interferring workload \( I_{\text{carry}}(\tau_i) \) (resp. non carry-in interfering workload \( I_{\text{non-carry}}(\tau_i) \)) of \( \tau_i \) on task \( \tau_j \) in any scenario of criticality level \( \ell \), is the cumulative length of time interval within the time window of size \( \Delta \) during which a job released by a carry-in task \( \tau_j \) (resp. a non-carry-in task \( \tau_j \)) 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 \( \tau_i \in \text{hp}(\tau_j) \) will be denoted by:

\[
I_{\text{carry}}(\tau_i, \tau_j) - I_{\text{non-carry}}(\tau_j, \tau_j) = I_{\text{carry}}(\tau_i, \ell) - I_{\text{non-carry}}(\tau_j, \ell)
\]

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 \( I_i(\Delta, \ell) \) of tasks \( \tau_j \in \text{hp}(\tau_i) \) on task \( \tau_i \) in any scenario of criticality level \( \ell \), over any time window of size \( \Delta \), is the sum of interfering workload of all the higher priority tasks within that window. The total interfering workload suffered by task \( \tau_i \) is computed as follows:

\[
I_i(\Delta, \ell) = \sum_{\tau_j \in \text{hp}(\tau_i)} I_{\text{carry}}(\tau_j, \tau_i) + I_{\text{non-carry}}(\tau_j, \tau_i)
\]

**Definition 12. (Interference \[19\])** The interference suffered by a task \( \tau_i \) in any scenario of criticality level \( \ell \), and
during a time interval of length $\Delta$, is the cumulative length of time intervals during which the $m$ processors are busy executing tasks belonging to $P(t)$. An upper-bound on the interference suffered by $\tau_1$ over an interval of length is given by $\left[ \frac{1}{m} \int_\Delta \tilde{\tau}(\Delta) \right]$. Finally, and from the above definitions, since in any scenario of criticality level $\ell$, $J_{x,k}$ is allowed to execute for at most $C_1(\ell)$ time units, the WCRT $R(\ell)$ of task $\tau_1$ at criticality level $\ell$ is obtained by determining the least fixed point of the following function:

$$R(\ell) = C_1(\ell) + \frac{1}{m} \int R(\ell) \, \tilde{\tau}(\Delta)$$

Note that since the task set is $S$-schedulable (as Section 2), it must be the case that $R(0) \leq D, \forall \ell \in L$. In the remainder of this section, we will denote by $T(\Delta, \ell) \leq \tilde{\tau}(\Delta, \ell)$ the total interfering workload suffered by $\tau_1$ over an interval of length $\Delta$. Moreover, we assume that the tasks in $\tau$ are prioritized in the order of their indices. That is, if $i < j$ then $\tau_i$ has a higher priority than $\tau_j$.

### 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 $t_{overrun}$ and that no job exceeds its WCET at criticality level $\ell$ after $t_{overrun}$. For every task $\tau_1$ in a decreasing order of priority, the protocol identifies a time instant $f'_{\tau_1}$ satisfying Condition 1.

**Condition 1.** Time $f'_{\tau_1}$ is such that:

1. $f'_{\tau_1} \geq f'_{\tau_{i-1}}$ (with $f'_0 = t_{overrun}$);
2. $\tau_1$ has no active job at time $f'_{\tau_1}$.

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

More precisely, starting at time $t_{overrun}$, the protocol (Algorithm 1) identifies the earliest time instant $f'_{\tau_1}$ satisfying Condition 1. That is, the protocol checks whether task $\tau_1$ (the highest priority task belonging to $\tau$) has an active job $J_{x,k}$ at time $t_{overrun}$. If it is the case, then the protocol waits for that job to complete its execution at time $f'_1 = f_{x,k}$. Otherwise, $f'_1 = t_{overrun}$. The protocol then looks for the earliest instant after $f'_1$ where $\tau_2$ has no active job (i.e., the earliest instant $f'_2 \geq f'_1$ satisfying Condition 1). Again, if at time $f'_2$, $\tau_2$ has no active job, then $f'_2 = f'_1$. Otherwise, the protocol simply waits for the active job of $\tau_2$ 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 $\tau_i$ only when it has previously identified such a time instant for task $\tau_{i-1}$). The re-enablement of the previously suspended tasks having a criticality at least equal to $\ell$ can then take place when an instant $f'_{\tau_i}$ satisfying Condition 1 has been found.

Note that our reasoning is based on the assumption that an overrun is unusual. In particular, we make the assumption that if the protocol identifies a time instant $f'_{\tau_i}$ satisfying Condition 1 for every task $\tau_1,...,\tau_n$, then these tasks will not overrun their WCET at level $\ell$ until the protocol was able to decrease CL from level $h$ to level $\ell$. 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 $f'_{\tau_i}$ satisfying Condition 1 for task $\tau_i$, 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, $\mathbb{L} = \{ LO, HI \}$. Let the task set $\tau = \{ \tau_1, \tau_2, \tau_3, \tau_4 \}$ consists of four implicit-deadline sporadic tasks with the following parameters:

$\tau_1 = (HI, [4, 5], 5, 5)$
$\tau_2 = (HI, [3, 4], 9, 9)$
$\tau_3 = (HI, [4, 9], 11, 11)$
$\tau_4 = (LO, [1, 1], 5, 5)$

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 $\tau_1$ has the highest priority and task $\tau_4$ the lowest) and consider the scenario illustrated by Figure 2 where tasks $\tau_2, \tau_3$ and $\tau_4$ release a job at time $t = 0$, and task $\tau_1$ releases a job at time $t = 2$. At time $t = 4$, $\tau_3$ does not signal its completion, which results in CL being increased from level LO to level HI. Task $\tau_4$ is therefore suspended. At time $t = 4$, the protocol is launched and checks whether task $\tau_1$ has an active job. Since the first job of task $\tau_1$ is still executing, the protocol must wait for that job to finish at time $t = 5$. Since at time $t = 5$, $\tau_2$ has no active job, the protocol can skip $\tau_2$ and consider $\tau_3$ instead. Since at time $t = 5$, $\tau_3$ has an active job, the protocol must wait for that job to finish at time $t = 9$. Because $\tau_3$ is the lowest priority task among the tasks in $\tau_{HI}$, it follows that no other task in $\tau$ will have an active job (the lower criticality tasks have been suspended.

![Figure 2: An example to illustrate the protocol described in Section 4.2.](image-url)
at time $t = 4$). Thus, at time $t = 9$, the criticality of the system can be decreased to level LO, thereby reactivating the task $\tau_4$.

### 4.4 Proof of Correctness

**Lemma 1.** Let $r_{\text{overrun}}$ be the last time at which a job overrun its WCET at criticality level $\ell$. Let us assume that the protocol identifies an instant $f'_{\ell - 1} \geq r_{\text{overrun}}$ respecting Condition 1 for task $\tau_{\ell - 1}$ and no job with an execution time greater than its WCET at criticality level $\ell$ is released after $r_{\text{overrun}}$. From time $f'_{\ell - 1}$ onwards, the actual total interfering workload $I(\Delta, \ell)$ suffered by task $\tau_{\ell}$ over any window of size $\Delta$ is less than or equal to $I(\Delta, \ell)$.

**Proof.** From Definition 11, we know that $I_{\ell}(\Delta, \ell)$ is an upper-bound on the total interfering workload suffered by task $\tau_{\ell}$ in any scenario of criticality level $\ell$. More precisely, we have $I_{\ell}(\Delta, \ell) \leq I(\Delta, \ell)$ if no job released by a task in higher priority order is available within the interval $\Delta$ for an execution time greater than its WCET at criticality level $\ell$. We now prove that this is the case from time $f'_{\ell - 1}$ onwards.

The proof is by induction on the task priorities. In particular, assuming that Lemma 1 holds for task $\tau_{\ell - 1}$, we prove that it also holds for task $\tau_{\ell}$.

**Base case.** Let us consider that $\tau_{\ell - 1}$ is the highest priority task (i.e., $\tau_{\ell - 1} = \tau_1$), and that $\tau_2 = \tau_{\ell - 1}$. It follows that the jobs released by task $\tau_1$ are the only jobs interfering with the execution of task $\tau_2$ from time $f'_{\ell - 1}$ onwards. Since task $\tau_2$ has no active job at time $f'_{\ell - 1}$ and because we assumed that all the jobs released by $\tau_2$ after $f'_{\ell - 1}$ have an execution time that is less than or equal to their WCET at level $\ell$, it follows that all the jobs interfering with the execution of $\tau_2$ from time $f'_{\ell - 1}$ onwards have an execution time that is less than or equal to their WCET at level $\ell$. Hence, $I_{2}(\Delta, \ell) \leq I(\Delta, \ell)$ for any window of size $\Delta$ starting at time $f'_{\ell - 1}$.

**Inductive step.** The jobs released by the task in $\text{hp}(\tau_i)$ are the only ones interfering with the execution of task $\tau_i$ from time $f'_{i - 1}$ onwards. Because Algorithm 1 considers the tasks in their priority order, we know from Condition 1 that for every task $\tau_i \in \text{hp}(\tau_j)$, $f'_{i} \leq f'_{i - 1}$. Furthermore, by the induction hypothesis, it holds that no job with an execution time greater than its WCET at criticality level $\ell$ interferes with the execution of $\tau_j$ after $f'_{j}$, and thus after $f'_{j - 1}$. Furthermore, because we made the assumption that no job released by a task after $f'_{j - 1}$ have an execution time not greater than their WCET at level $\ell$, it results that all the jobs interfering with the execution of $\tau_j$ after $f'_{j - 1}$ have an execution time not greater than their WCET at level $\ell$. Hence, $I_{\ell}(\Delta, \ell) \leq I(\Delta, \ell)$ for any window of size $\Delta$ starting at time $f'_{\ell - 1}$. $\square$

**Corollary 1.** Let $r_{\text{overrun}}$ be the last time at which a job overrun its WCET at criticality level $\ell$. When the protocol identifies an instant $f'_{\ell - 1} \geq r_{\text{overrun}}$ satisfying Condition 1 for task $\tau_{\ell}$, then from time $f'_{\ell - 1}$ onwards, the actual total interfering workload suffered by any task $\tau_i \in \tau$ is less than or equal to $I(\Delta, \ell)$.

**Proof.** Since, by definition of the highest priority task, the highest priority task $\tau_1$ never suffers from any interference from lower priority tasks, this corollary is obviously true for $\tau_1$. Furthermore, because according to Algorithm 1, we have $f'_{\ell - 1} \leq f'_{\ell} \leq f'_{\ell - 1}$ for $1 \leq i \leq n$, this corollary directly follows from Lemma 1 for any task $\tau_i$ such that $1 < i \leq n$. $\square$

![Figure 3: Upper-bound on the re-enablement of suspended tasks.](image)

From Corollary 1, we will now show that every suspended task $\tau_k \in \tau^{(b)}$ can be re-enabled at time $f'_{n}$.

**Theorem 1.** Upon an overrun at time $r_{\text{overrun}}$, when the protocol identifies an instant $f'_{\ell - 1} \geq r_{\text{overrun}}$ satisfying Condition 1 for the lowest priority task $\tau_{\ell} \in \tau$, every suspended task $\tau_k \in \tau^{(b)}$ can be re-enabled.

**Proof.** From Corollary 1, we know that from time $f'_{\ell - 1}$ onwards, every job $J_{i,k}$ released by a task $\tau_i \in \tau$ will suffer a total interfering workload that is not greater than $I(\Delta, \ell)$ in any window of size $\Delta$. Hence, from Expression 5, every job will complete its execution after no more that $R_{i}(\ell)$ time units. Furthermore, since the system was deemed $S$-schedulable, we have that $R_{i}(\ell) \leq D_{i} \forall i$. Consequently, every task $\tau_i \in \tau^{(b)}$ can be safely re-enabled. $\square$

### 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 $\ell$, the system will eventually be able to recover from the generated overload.

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

$$s\text{usp}^{h} = \sum_{\tau_i \in \tau^{(b)}} R_{i}(h)$$

**Proof.** Since all the tasks that have a criticality smaller than $h$ are suspended at time $r_{\text{overrun}}$, it follows that the only tasks that may have an active job within the time interval $[r_{\text{overrun}}, r_{\text{overrun}} + s\text{usp}^{h})$ are the ones in $\tau^{(b)}$. Therefore, from line 6 of Algorithm 1, we have:

$$\forall \tau_i \in \tau \setminus \tau^{(b)}, \quad f'_{\ell - 1} = f'_{\ell - 1}$$

Furthermore, in the worst-case, each task $\tau_i \in \tau^{(b)}$ released a job right at time $f'_{\ell - 1}$. 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:

\[ \forall \tau_i \in \mathcal{S}(h), f_i' = f_i'_{\tau_i} + R_i(h) \]  

(6)

Because all the suspended tasks in \( \mathcal{S}(h) \) can be re-enabled when an instant \( f_i'_{\tau} \) is found for task \( \tau_i \), \( \text{susp}_h^b \) is an upper-bound on \( f_i'_{\tau} \). Using Equations 5 and 6, we get that \( f_i'' \leq \sum_{\tau_i \in \mathcal{S}(h)} R_i(h) \), thereby leading to:

\[ \text{susp}_h^b = \sum_{\tau_i \in \mathcal{S}(h)} R_i(h) \]

\( \Box \)

An illustration of the computation of the upper-bound \( \text{susp}_h^b \) 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 \( \ell \in L \), 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 \( \ell \)) The reference schedule at criticality level \( \ell \) for a task set \( \tau \) considering a scheduling algorithm \( \mathcal{S} \), is the schedule generated by \( \mathcal{S} \) for the jobs released by \( \tau \) under the assumption that all these jobs execute exactly for their WCET at level \( \ell \).

Let us consider that an algorithm \( \mathcal{S} \) is used to schedule a mixed-criticality task set \( \tau \). At run-time, if the current criticality level of the module is \( \ell \), then \( \mathcal{S} \) will schedule every task \( \tau_i \) with \( L_i \geq \ell \) under the assumption that none of these tasks will exceed its WCET at criticality level \( \ell \). Yet, at some point in time, a task \( \tau_i \) with \( L_i > \ell \) might overrun, thus causing the system to increase the criticality level from \( \ell \) 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 \( \ell \).

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.

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 \( \tau \) 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 \( \tau_i \);
- \( T_i \in \mathbb{N}_0 \) denotes the minimum inter-arrival time of task \( \tau_i \);
- \( D_i \in \mathbb{N}_0 \) denotes the deadline of task \( \tau_i \).

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 an 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 \( J'_i \) has the same arrival time as \( J_i \) 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 \( f_i' \leq f_i \).

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 \( \tau \) 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 \( \tau \) 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 $\tau$ under every (other) possible execution scenario. Hence, most of the schedulability analysis techniques base their computations on the parameter $C_t$ (i.e., the worst-case execution time of each task).

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

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

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

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

Let $J''$ denote a worst-case scenario corresponding to $J$ such that every job in $J''$ 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 $S_G(J'')$ denote the schedule of the scenario $J''$ by $S$ on $\pi$.

Let $J_{i,k}$ denote the last job released by task $\tau_i$ in $[0, t)$, i.e., $J_{i,k}$ is the job of $\tau_i$ for which $\text{act-rem}(t) > \text{ref-rem}(t)$. By definition of $J_{i,k}$, it holds that $J_{i,k} \in J$ and $J_{i,k} \in J''$ and by definition of $J''$, the actual execution time $c_{i,k}$ of $J_{i,k}$ in $J''$ is no greater than its corresponding actual execution time $c_{i,k}$ in $J''$.

Now, let us modify $J$ and $J''$ by setting $c_{i,k}^{\text{new}} = c_{i,k} - \text{ref-rem}(t)$ in both scenarios $J$ and $J''$. With this reduced $c_{i,k}^{\text{new}}$, it holds that $J_t$ now finishes at time $t$ in $S_G(J'')$ whereas it finishes after time $t$ in $S_G(J)$ (since it is assumed that $\text{act-rem}(t) > \text{ref-rem}(t)$). This clearly contradicts the predictability of $S$ as the scenario $J$ can be obtained from $J''$ 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 $J''$ when scheduled by $S$. \hfill $\square$

**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 $S$ on a platform $\pi$. 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 $S'$ on the same platform $\pi$. In the schedule of $J$ by $S$ (depicted as schedule (a) in Figure 5), let the functions $\text{completed}(J,t)$, $\text{active}(J,t)$, and $\text{future}(J,t)$ 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 $\text{completed}(J',t)$, $\text{active}(J',t)$, and $\text{future}(J',t)$ are defined analogously for the schedule of $J'$ by $S'$ (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 $S$ and $S'$, respectively) at which every job $J_k \in \text{active}(J,t)$ can be one-to-one mapped to a job $J''_k \in \text{active}(J',t)$ such that $J_k$ and $J''_k$ have the same release time, deadline, and execution requirement, but the remaining execution time $\text{rem}_k(t)$ of $J_k$ is less than or equal to that of the corresponding job $J''_k$, i.e.,

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

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 $S$ from time 0 to time $t$,
2. at time $t$, the scheduler $S$ is substituted for $S'$, and
3. from time $t$ onward, the next jobs to arrive are those from $\text{future}(J',t)$ 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 $\text{completed}(J',t)$ and $\text{future}(J',t)$ from $J'$, and
2. a subset of jobs containing one job $J''_k$ for each $J''_k \in \text{active}(J',t)$, with same release time and deadline as $J'_k$ but with an execution requirement $c''_k$ equal to $c''_k = (\text{rem}'_k(t) - \text{rem}_k(t))$. That is, $c''_k \leq c_k = c_k$. Note that $\text{rem}_k(t)$ is assumed to be zero if there is no corresponding job $J_k$ in $\text{active}(J,t)$ for the job $J''_k$ (as $\text{active}(J,t)$ can contain more jobs than $\text{active}(J,t)$).

If $J''$ is scheduled by $S'$ on $\pi$ (see schedule (c) in Figure 5), it holds by construction that at time $t$, the remaining execution time $\text{rem}_k(t)$ of each job $J''_k \in \text{active}(J'',t)$ is equal to $\text{rem}_k(t) \leq \text{rem}'_k(t)$ (or equal to zero if $\exists J_k \in \text{active}(J,t)$ corresponding to $J''_k$ as explained above). Thus, it holds by Definition 16, from the schedulability of $J'$ and the predictability of $S'$ that the schedule of $J''$ by $S'$ meets all the job deadlines as well.
Now, let us compare the schedule of $J''$ by $S'$ (in Figure 5) with the schedule assumed in the claim (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 $\text{ref-rem}(t)$ for each task $\tau_i \in \tau$ at any time $t$ during the scheduling of a traditional task set $\tau$. Remember that $\text{ref-rem}(t)$ denotes the worst-case remaining execution time of task $\tau_i$ in the schedule of all the jobs released by $\tau$ 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 $\tau_i$ records the value of $\text{ref-rem}(t)$ 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 $\tau_i$ at time $t$, the alpha-queue creates an element for task $\tau_i$ and sets this entry to $C_i$, i.e. $\text{ref-rem}(t) = C_i$.

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 $\text{ref-rem}(t)$ (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, $\text{ref-rem}(t)$ 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 $\text{ref-rem}(t)$ 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 $\tau_i$ a variable $Q_i(t)$ that depends on the current time $t$. The variable $Q_i(t)$ records the duration of time for which the last released job of task $\tau_i$ has been executed up to time $t$. Let the current criticality level be denoted by $c_i$. At every time instant $t$, if there exists a task $\tau_j$ such that $Q_j(t) > C_j(c_i)$ (the task is overrunning at the current level), then the system switches to the next (higher) criticality level $h = c_i + 1$. Otherwise, the system computes the worst-case remaining execution time at level $\ell$ (where $\ell$ can be $c_i - 1$ or any lower level) as $\text{act-rem},(t, \ell) = C_i(\ell) - Q_i(t)$ of every task $\tau_i$, with $L_i \geq \text{curr}$. The system switches to the lower criticality level $\ell$ if the following two sets of conditions are satisfied:

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

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

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

The quantities $\text{ref-rem}(t, k)$, 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 $\tau_i$ at level $k$ records the value of $\text{ref-rem}(t, k)$ 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 $\tau_i$ at time $t$, each alpha-queue $k$ with $k \leq L_i$, creates an element for task $\tau_i$ and sets this element to $C_i(k)$, i.e. $\text{ref-rem}(t, k) = C_i(k)$.

**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 $\text{ref-rem}(t, k)$ (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 $\tau$ which is deemed $A$-schedulable on a platform $\pi$ by a predictable algorithm $A$. At any time $t$ during the scheduling of $\tau$ by $A$ at the criticality level $\text{curr}$, it is safe to switch back to the lower criticality level $\ell < \text{curr}$ if, for all task $\tau_i \in \tau$ with $L_i \geq \text{curr}$:

$$\text{act-rem},(t, \ell) \geq 0 \quad \text{and} \quad \text{act-rem},(t, \ell) \leq \text{ref-rem}(t, \ell)$$

**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 $\text{curr}$ has executed for more than its worst-case execution time at level $\ell$ (otherwise the task would be overrunning at level $\ell$ 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 $\text{curr}$, (ii) $J'$ as the set of jobs of tasks with criticality level $\ell$ and (iii) both $S$ and $S'$ as algorithm $A$, Lemma 4 is applicable to the situation described in the claim (since Expression 9 is equivalent to Expression 7). Therefore, switching from criticality level $\text{curr}$ to the lower criticality level $\ell$ 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 $f$ 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 FEDER-022701 (CISTER) and grant nr. FCOMP-01-0124-FEDER-Competitiveness’, within projects ref. FCOMP-01-0124-COMPETE (Operational Programme ‘Thematic Factors of Competitiveness’) and grant nr. 295371 (CRAFTERS); by the EU ARTEMIS JU funding, within grant nr. 333053 (REGAIN); by National Funds through FCT and by FEDER-022701 (CISTER) and ref. FCOMP-01-0124-FEDER-Competitiveness’, within projects ref. FCOMP-01-0124-COMPETE (Operational Programme ‘Thematic Factors of Competitiveness’) 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


