Mixed-criticality scheduling upon unreliable processors

Sanjoy Baruah Zhishan Guo
The University of North Carolina at Chapel Hill

***Abstract*— An *unreliable* processor is characterized by two execution speeds: a normal speed and a degraded speed. Under normal circumstances it will execute at its normal speed; unexpected conditions may occur during run-time that cause it to execute more slowly (but no slower than at its degraded speed).**

The problem of executing an integrated workload, consisting of some more important components and some less important ones, upon such an unreliable processor is considered. It is desired that all components execute correctly under normal circumstances, whereas the more important components should execute correctly (although the less important components need not) if the processor runs at any speed no slower than its specified degraded speed.

I. Introduction

In *mixed-criticality (MC)* systems, functionalities of different degrees of importance (or *criticalities*) are implemented upon a common platform. Such MC implementations are becoming increasingly common in embedded systems – consider, for example, Integrated Modular Avionics (IMA) in aviation [17] and the AUTOSAR initiative (`www.autosar.org`) for automotive systems. As a consequence the real-time systems research community has recently devoted much attention to better understanding the challenges that arise in implementing such MC systems.

Much prior work on MC scheduling (see, e.g., [19], [5], [2], [8], [3], [20], [16], [6], [18], [12] – this list is not meant to be exhaustive) has taken the approach of validating the correctness of highly critical functionalities under *more pessimistic assumptions* than the assumptions used in validating the correctness of less critical functionalities. (For example, a piece of code may be characterized by a larger worst-case execution time (WCET) [19] in the more pessimistic analysis, or recurrent code that is triggered by some external recurrent event may be characterized by a higher frequency [1].) All functionalities are expected to be demonstrated correct under the less pessimistic analysis, whereas the analysis under the more pessimistic assumptions need only demonstrate the correctness of the more critical functionalities.

In this paper we take a somewhat different perspective on mixed-criticality scheduling: the system is analyzed only once, under a single set of assumptions. The mixed-criticality nature of the system arises in the fact that while we would like all functionalities to execute correctly under normal circumstances, it is essential that the more critical functionalities execute correctly even when circumstances are *not* normal. To express this formally, we model the workload of a MC system as being comprised of a collection of real-time jobs — these jobs may be independent, or they may be generated by recurrent tasks. Each job is characterized by a release date, a worst-case execution time (WCET), and a deadline; each job is further designated as being HI-criticality (more important) or LO-criticality (less important). We desire to schedule the system upon a single processor. This processor is *unreliable* in the following sense: while under normal circumstances it completes one unit of execution during each time unit (equivalently, it executes as a speed-1 processor), it may at any instant lapse into a degraded mode during which it can only complete as few as s units of execution during each time unit, for some (known) constant $s < 1$. It is not *a priori* known when, or whether, such degradation will occur[1]. We seek a scheduling strategy that *guarantees to complete all jobs by their deadlines if the performance of the processor does not degrade during run-time, while simultaneously guaranteeing to complete all* HI*-criticality jobs if the processor does suffer a degradation in performance*.

Example 1: Consider the following collection of two jobs, to be scheduled on a preemptive processor with normal speed 1 and degraded speed $s = \frac{1}{2}$:

Job	Criticality	Release date	WCET	Deadline
J_1	LO	0	3	5
J_2	HI	1	4	10

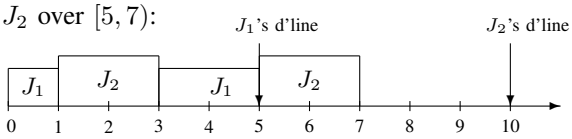
An Earliest Deadline First (EDF) [13] schedule for this system prioritizes J_1 over J_2. This is fine if the processor does not degrade: J_1 executes over the interval $[0, 3)$ and J_2 over $[3, 7)$, thereby resulting in both deadlines being met.

Now suppose that the processor were to degrade at some instant within the time-interval $[0, 10]$: a correct scheduling strategy should execute the HI-criticality job J_2 to complete by its deadline (although it may fail to execute J_1 correctly). But consider the scenario where the processor degrades starting at time-instant 3: in the EDF schedule J_2 would obtain merely $(10 - 3) \times \frac{1}{2} = 3\frac{1}{2}$ units of execution prior to its deadline at time-instant 10. Since J_2's WCET is 4, we conclude that EDF does not schedule this system correctly.

An alternative scheduling strategy could instead execute jobs as follows on a normal (non-faulty) processor: J_1 over

[1]We do however assume that the system is capable of self-monitoring: it immediately knows if and when such degradation occurs. We leave the analysis of systems in which such self-monitoring does not occur to future work – see Section VI.

the interval $[0, 1)$; J_2 over $[1, 3)$; J_1 again, over $[3, 5)$; and finally J_2 over $[5, 7)$:



If the processor degrades at any instant during this execution then J_1 is immediately discarded and the processor executes J_2 exclusively.

It may be verified, by exhaustive consideration of all possible instants at which the processor may degrade, that this scheduling strategy will result in J_2 completing by its deadline regardless of when (if at all) the processor degrades, and in both deadlines being met if the processor remains normal (or degrades at any instant ≥ 5). ■

Contributions and Organization. The research described in this paper aims to define a formal framework for the scheduling-based analysis of mixed-criticality (MC) systems of the kind described above, that execute upon unreliable processors. To this end, in Section II we introduce a very simple model for MC systems, that allows for the representation of systems consisting of a finite number of independent jobs. In Section III we present, and analyze, algorithms for the preemptive scheduling of MC systems that can be represented using this model; in Section IV, we consider the problem when preemption is forbidden. In Section V, we consider a more general model for MC systems: one that allows for the modeling of systems comprised of recurrent tasks. We conclude in Section VI by placing this work within the larger context of mixed-criticality scheduling, and briefly enumerate some important and interesting directions for further research.

A note. Although we have chosen to model the problem in terms of real-time jobs executing on unreliable processors, the model (and our results) are equally applicable to the transmission of time-sensitive messages on potentially faulty communication media. Specifically, they are particularly relevant to data-communication problems in which time-sensitive messages and message streams must be transmitted over potentially faulty communications media which can provide a high bandwidth under most circumstances but can only *guarantee* a lower bandwidth: the high bandwidth would correspond to the normal processor speed, and the lower bandwidth to the degraded speed. We therefore believe that this work is relevant to problems of factory communication, communication within automobiles or aircraft, wireless sensor networks, etc., in addition to processor scheduling of mixed-criticality workloads.

II. Model

We start out considering a workload model consisting of *independent jobs*; a model for representing *recurrent tasks* is considered in Section V.

In our model, a mixed-criticality real-time workload is comprised of basic units of work known as mixed-criticality jobs. Each mixed-criticality (MC) job J_i is characterized by a 4-tuple of parameters: a release date a_i, a WCET c_i, a deadline d_i, and a criticality level $\chi_i \in \{\text{LO}, \text{HI}\}$. A mixed-criticality *instance* I is specified by specifying

1) a finite collection of MC jobs $\{J_1, J_2, \ldots, J_n\}$, and
2) an unreliable processor that is characterized by both a normal speed (without loss of generality, assumed to be equal to one) and a specified *degraded processor speed* $s < 1$.

The interpretation is that the jobs $\{J_1, J_2, \ldots, J_n\}$ are to execute on a single shared processor that has two modes: a *normal* mode and a *degraded* or *faulty* mode. In normal mode, the processor executes as a unit-speed processor and hence completes one unit of execution per unit time, whereas in degraded mode it completes less than one, but at least s, units of execution per unit time.

The processor starts out executing at its normal speed. It is not *a priori* known when, if at all, the processor will degrade and begin executing at its degraded speed: this information only becomes revealed during run-time when the processor actually begins executing at the slower speed. We seek to determine a ***correct scheduling strategy***:

Definition 1 (correct scheduling strategy): A scheduling strategy for MC instances is *correct* if it possesses the property that upon scheduling any MC instance $I = (\{J_1, J_2, \ldots, J_n\}, s)$,

- if the processor remains in normal mode throughout the interval $[\min_i\{a_i\}, \max_i\{d_i\})$, then all jobs complete by their deadlines; and
- if the processor operates at or above its degraded speed of s throughout the interval $[\min_i\{a_i\}, \max_i\{d_i\})$, then all jobs J_i with $\chi_i = \text{HI}$ complete by their deadlines.

■

That is, a correct scheduling strategy ensures that HI-criticality jobs execute correctly regardless of whether the processor executes in normal or degraded mode; LO-criticality jobs are required to execute correctly only if the processor executes throughout in normal mode.

Related work. A lot of research has recently been done on various aspects of mixed-criticality scheduling. As stated in Section I above, though, most of this prior work draws inspiration from the seminal work of Vestal [19], which asked how a *single* MC system could be subject to multiple different analyses, each under different assumptions and with different requirements on the system's behavior. We are not aware of other work in real-time mixed-criticality scheduling theory that addresses our model: all jobs should complete under normal circumstances and HI-criticality jobs should complete (although LO-criticality jobs may not) under degraded conditions. To the best of our knowledge, current practice in implementation of such mixed-criticality systems to assign greater scheduling priority to HI-criticality jobs, but this approach can easily be seen to perform arbitrarily poorly even in scheduling under non-degraded conditions.

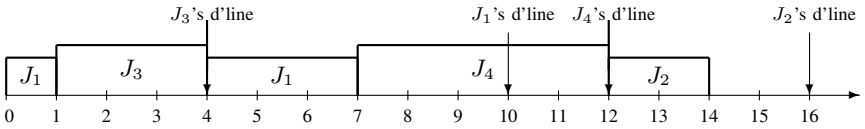


Fig. 1. Scheduling table $S(I)$ for the MC instance I of Example 2

III. Preemptive scheduling

In this section we present efficient strategies for scheduling preemptable mixed-criticality instances. We start out with a general **overview** of our strategy. Given an instance I, our scheduling strategy is as follows. Prior to run-time we will construct a scheduling table $S(I)$, for use while the processor is in normal (i.e., not faulty) mode. This scheduling table should possess the property that each job J_i receives c_i units of execution over the interval $[a_i, d_i)$. During run-time scheduling decisions are initially made according to scheduling table $S(I)$. If at any instant it is detected that the processor has transited to faulty mode, scheduling table $S(I)$ is no longer used; instead, we immediately discard all LO-criticality jobs and henceforth execute the (remaining) HI-criticality ones according to EDF.

In the remainder of this section we present, and prove the correctness of, simple efficient algorithms for constructing these scheduling tables $S(I)$ optimally. By *optimal*, we mean that if there is a correct scheduling strategy –Definition 1– for an instance I, then the scheduling strategy described above is also a correct scheduling strategy with the tables we will construct. For pedantic reasons, we start out in Section III-A by considering MC instances in which all jobs have a common release date – this section should help develop the intuition and insights that are subsequently used to solve the general problem in Section III-B. We have implemented the algorithm that solves the general problem, and have performed some simulation experiments that seek to determine the tightness of some lower bounds derived in Section III-B; we report on these simulation experiments in Section III-C.

A. Synchronous instances

Definition 2 (Synchronous instance): A mixed-criticality instance I is said to be ***synchronous*** if all jobs J_i in the instance I have the same release date a_i; without loss of generality, we may assume that this common release date is zero. ■

We now describe a procedure for generating the scheduling table $S(I)$ for any synchronous instance I; this procedure is then illustrated via a simple example (Example 2):

1) First, schedule each LO-criticality job to execute *as late as possible*.
   This can be accomplished by considering the LO-criticality jobs in non-increasing order of deadlines (i.e., with the latest-deadline job considered first); in considering a job, we schedule it as close to its deadline as possible[2].
2) Next construct an EDF schedule for the HI-criticality jobs in the remaining processor capacity.
3) If this fails then *report failure* and exit: there does not exist a schedule that can guarantee to meet all deadlines upon the normal (non-degraded) processor.
4) Else, **validate** whether this schedule will perform satisfactorily during degraded mode. This is done as follows.
   a) At the start of *each* continuous interval during which HI-criticality jobs are scheduled, determine whether all remaining HI-criticality deadlines will be met under an EDF schedule of only the remaining HI-criticality jobs on a degraded processor.
   b) If this fails then *report failure* and exit.
5) *Report success*

Example 2: Consider an instance I comprised of four jobs with parameters as shown below, to be implemented upon an unreliable processor with normal speed one and degraded speed $s = 1/2$.

J_i	a_i	c_i	d_i	χ_i
J_1	0	4	10	HI
J_2	0	2	16	HI
J_3	0	3	4	LO
J_4	0	5	12	LO

First, our algorithm *generates* the scheduling table $S(I)$ as follows (see Figure 1):

- First, schedule the LO-criticality jobs to execute as late as possible. This results in J_4 executing over $[7, 12)$, and J_3 executing over $[1, 4)$.
- Next, schedule the HI-criticality jobs in the remaining capacity, using EDF. Job J_1 therefore executes over $[0, 1)$ and $[4, 7)$, while J_2 executes over $[12, 14)$.

Next, this schedule must be *validated* to determine whether it would meet all HI-criticality deadlines in the event of a degradation in processor speed. For this validation step, there are three continuous intervals of HI-criticality execution to consider: $[0, 1)$, $[4, 7)$, and $[12, 14)$.

- If the processor degrades to speed $1/2$ at time-instant 0 and only J_1 and J_2 need to complete, then an EDF schedule on these two jobs would have J_1 execute over $[0, 8)$ and J_2 over $[8, 12)$. Both jobs thus complete by their deadlines.
- Suppose next that the processor degrades to speed $1/2$ at time-instant 4 and only J_1 and J_2 need to complete. J_1

[2]Observe that each job executes non-preemptively in such a schedule; therefore, the number of distinct contiguous blocks of execution in this partial schedule is bounded by the number of LO-criticality jobs.

has 3 units of remaining execution while J_2 has 2 units of remaining execution. An EDF schedule would have J_1 execute over $[4, 10)$ and J_2 over $[10, 14)$. Both jobs thus complete by their deadlines.

- Suppose finally that the processor degrades to speed $1/2$ at time-instant 12 and only J_1 and J_2 need to complete. J_1 has already completed execution; J_2 has 2 units of remaining execution. An EDF schedule would execute J_2 over $[12, 16)$, thus complete it by its deadline.

The schedule is therefore validated. ■

Theorem 1: Our algorithm is ***correct***: if it reports success then during run-time the dispatch policy guarantees to both (i) meet all deadlines if the processor executes at normal speed throughout; and (ii) meet all HI-criticality deadlines if the processor begins executing at degraded speed after some (*a priori* unknown) point in time.

Proof: Suppose that the algorithm generates a scheduling table and reports success.

- If the processor executes throughout at normal speed, it is evident that the schedule will complete each job by its deadline.
- Suppose now that the processor begins executing at some speed s' at time t, where $s \leq s' < 1$. We consider two possibilities, depending on whether some HI-criticality job is executing or not at time-instant t
  1) ***Some* HI*-criticality job is executing at time-instant t.*** Let $[a, b)$ denote the contiguous block of HI-criticality execution such that $t \in [a, b)$. We had determined (during the validation step) that the HI-criticality jobs remaining at time-instant a would be EDF-scheduled to meet all deadlines on a speed s processor. It follows from the speed-sustainability of preemptive uniprocessor EDF (speed of the processor, at one, is greater over $[a, t)$; $s' \geq s$) that our dispatch strategy will also meet all HI-criticality deadlines provided the speed of the processor remains above s.
  2) ***Some* HI*-criticality job is not executing at time-instant t.*** Let a denote the earliest start time of a contiguous block of HI-criticality execution such that $t \leq a$. We had determined during step 2 that the HI-criticality jobs remaining at time-instant a would be EDF-scheduled to meet all deadlines on a speed s processor. It once again follows from the sustainability of preemptive uniprocessor EDF that our dispatch strategy will also meet all HI-criticality deadlines provided the speed of the processor remains above s.

■

Theorem 2: Our algorithm is ***optimal***: if it reports failure, then no non-clairvoyant scheduling strategy can guarantee to both (i) meet all deadlines if the processor executes at normal speed throughout; and (ii) meet all HI-criticality deadlines if the processor begins executing at degraded speed after some (*a priori* unknown) point in time.

Proof: If our algorithm reports failure while generating the scheduling table, it is straightforward to show that the instance is infeasible on a speed-1 processor. We therefore consider the case when our algorithm reports failure during the validation phase. Specifically, suppose that our algorithm, upon considering a contiguous block $[a, b)$ of high-criticality execution in the scheduling table, determines that all HI-criticality execution remaining at time-instant a cannot be guaranteed to complete on time when they are EDF-scheduled upon a speed-s processor.

The crucial observation here is that the scheduling table that we generate executes HI-criticality work as soon as possible (since it schedules all LO-criticality jobs for execution as late as possible). Hence it must be the case that at time-instant a no other schedule could have completed more HI-criticality work than our schedule does. Furthermore, within this HI-criticality execution our schedule executes the HI-criticality jobs in EDF order. Since our schedule cannot complete all the remaining HI-criticality work on time upon a speed-s processor, it follows that no other algorithm will be able to, either. Hence, any other algorithm will also report failure.

■

B. General (not necessarily synchronous) instances

We start out identifying the following (obvious) necessary conditions for MC-schedulability:

Lemma 1: In order that a correct scheduling strategy exist for MC instance $I = (\{J_1, J_2, \ldots, J_n\}, s)$, it is *necessary* that (i) EDF correctly schedule all the jobs in I on a speed-1 processor, and (ii) EDF correctly schedule all the HI-criticality jobs in I on a speed-s processor. ■

Given any instance I, it can be efficiently determined whether I satisfies the necessary conditions of Lemma 1: simply simulate the EDF scheduling of all the jobs in I upon a unit-speed processor, and of the HI-criticality jobs in I upon a speed-s processor. In the remainder of this section, let us therefore assume that any instance under consideration satisfies these necessary conditions. (In other words, an instance that fails these necessary conditions can obviously not have a correct scheduling strategy, and is therefore flagged as being unschedulable.)

Given a (not necessarily synchronous) MC instance $I = (\{J_1, J_2, \ldots, J_n\}, s)$ that satisfies the conditions of Lemma 1, we now describe how to construct a *linear program* of size polynomial in the size of the instance I, such that a feasible solution for this linear program can be used to construct the scheduling table $S(I)$. It is known that a linear program can be solved in polynomial time by the ellipsoid algorithm [10] or the interior point algorithm [9]. (In addition, the exponential-time simplex algorithm [4] has been shown to perform extremely well "in practice," and is often the algorithm of choice despite its exponential worst-case behavior.)

Without loss of generality, assume that the HI-criticality jobs in I are indexed $1, 2, \ldots, n_h$ and the LO-criticality jobs are indexed $n_{h+1}, \ldots, n$.

Let $t_1, t_2, \ldots, t_{k+1}$ denote the at most $2n$ distinct values for the release date and deadline parameters of the n jobs, in

Given the MC instance $(\{J_1, J_2, \ldots, J_n\}, s)$, with the job release-dates and deadlines partitioning the time-line over $[\min_i\{a_i\}, \max_i\{d_i\})$ into the k intervals $I_1, I_2, \ldots, I_k$

Determine values for the $\Big(n \times k\Big)$ variables

$$\Big\{x_{i,j}\Big\}_{i=1,\ldots,n, j=1,\ldots,k}$$

satisfying the following **constraints:**

- For each i, $1 \leq i \leq n$,
$$\Big(\sum_{(j|t_j \geq a_i \ \wedge \ d_i \geq t_{j+1}} x_{i,j}\Big) \geq c_i \qquad (1)$$
- For each j, $1 \leq j \leq k$,
$$\Big(\sum_{i=1}^{n} x_{i,j}\Big) \leq t_{j+1} - t_j \qquad (2)$$
- For each ℓ, $1 \leq \ell \leq k$, for each m, $\ell < m \leq (k+1)$
$$\Big(\sum_{i:(\chi_i=\text{HI}) \wedge (d_i \leq t_m)} \Big(\sum_{j=\ell}^{m-1} x_{i,j}\Big)\Big) \leq s(t_m - t_\ell) \qquad (3)$$

Fig. 2. Linear program for constructing scheduling table $S(I)$ for a given MC instance $(\{J_1, J_2, \ldots, J_n\}, s)$

increasing order ($t_j < t_{j+1}$ for all j). These release dates and deadlines partition the time-interval $\big[\min_i\{a_i\}, \max_i\{d_i\}\big)$ into k intervals, which we will denote as $I_1, I_2, \ldots, I_k$; $I_j := [t_j, t_{j+1})$

To construct our linear program we define $n \times k$ variables $x_{i,j}$, $1 \leq i \leq n; 1 \leq j \leq k$. The variable $x_{i,j}$ denotes the amount of execution we will assign to job J_i in the interval I_j, in the scheduling table that we are seeking to build.

We use the following n constraints to specify that each job receives adequate execution in the normal schedule:

$$\Big(\sum_{(j|t_j \geq a_i \ \wedge \ d_i \geq t_{j+1}} x_{i,j}\Big) \geq c_i, \text{ for each } i, 1 \leq i \leq n \qquad (1)$$

and the following k constraints to specify the capacity constraints of the intervals:

$$\big(\sum_{i=1}^{n} x_{i,j}\big) \leq t_{j+1} - t_j, \text{ for each } j, 1 \leq j \leq k \qquad (2)$$

Within each interval, we will execute all the HI-criticality jobs assigned execution within that interval first, followed by all the LO-criticality jobs assigned execution within that interval. That is, the interval I_j will have a block of HI-criticality execution of duration $\sum_{i=1}^{n_h} x_{i,j}$, followed by a block of LO-criticality execution of duration $\sum_{i=n_h+1}^{n} x_{i,j}$.

It should be evident that any scheduling table generated in this manner from $x_{i,j}$ values satisfying the above $(n+k)$ constraints will execute all jobs to completion upon a normal (non-degraded) processor. It now remains to write constraints for specifying the requirements that the HI-criticality jobs complete execution even in the event of the processor degrading into faulty mode. As was the case with scheduling tables for synchronous instances, it is not hard to show that the worst-case scenarios occur when the processor transits to degraded mode at the very *beginning* of a contiguous block of HI-criticality execution in the scheduling table. For each ℓ, $1 \leq \ell \leq k$, we represent the possibility that this transition occurs at the start of the interval I_ℓ in the following manner:

(i) Suppose that the fault occurs at time-instant t_ℓ; i.e., the start of the interval I_ℓ. Henceforth, only HI-criticality jobs will be executed; furthermore, these will be executed according to preemptive EDF.

(ii) Hence for each $t_m \in \{t_{\ell+1}, t_{\ell+2}, \cdots, t_{k+1}\}$, constraints must be introduced to ensure that the cumulative remaining execution requirement of all HI-criticality jobs with deadline at or prior to t_m can complete execution by t_m on a speed-s processor.

(iii) This is ensured by writing a constraint
$$\Big(\sum_{i:(\chi_i=\text{HI}) \wedge (d_i \leq t_m)} \big(\sum_{j=\ell}^{m-1} x_{i,j}\big)\Big) \leq s(t_m - t_\ell) \qquad (3)$$
To see why this represents the requirement stated in (ii) above, note that for any job J_i with $d_i \leq t_m$, $\big(\sum_{j=\ell}^{m-1} x_{i,j}\big)$ represents the remaining execution requirement of job J_i at time-instant t_ℓ. The outer summation on the LHS is simply summing this remaining execution requirement over all the HI-criticality jobs that have deadlines at or prior to t_m.

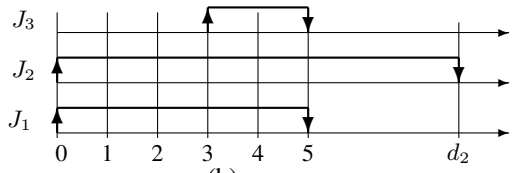
(iv) A moment's thought should convince the reader that rather than considering all t_m's in $\{t_{\ell+1}, t_{\ell+2}, \cdots, t_{k+1}\}$ as stated in (ii) above, it suffices to only consider those that are deadlines for some HI-criticality job.

(v) The Constraints (3) above only prevent missed deadlines after t_ℓ when the (degraded) processor is continually busy over the interval between t_ℓ and the missed deadline; what about deadline misses when the processor is not continually busy over this interval (and the RHS of the inequality of Constraints (3) therefore does not reflect the actual amount of execution received)? We point out that for such a deadline miss to occur, it must be the case that there is a subset of HI-criticality jobs – those with release dates and deadlines between the last idle instant prior to the deadline miss and the deadline miss itself – that miss their deadlines on a speed-s processor. But this would contradict our assumption that the instance passes the necessary conditions of Lemma 1, i.e., all the HI-criticality jobs together (and therefore, every subset of these jobs) execute successfully on a speed-s processor.

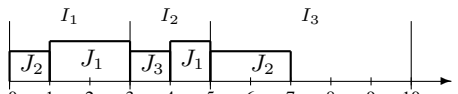
Given a solution to this linear program, we construct a scheduling table that assigns job J_i an amount $x_{i,j}$ of execution during the interval I_ℓ, for each pair (i, ℓ); in I_ℓ, HI-criticality execution is performed before LO-criticality execution – the jobs may be executed in any order within each criticality level. During run-time, scheduling decisions are initially made according to this scheduling table. If a processor

J_i	a_i	c_i	d_i	χ_i
J_1	0	3	5	LO
J_2	0	c_2	d_2	HI
J_3	3	1	5	HI

(a)



(b)



(c)

Fig. 3. Illustrating Example 3. The jobs are listed in (a), and depicted graphically in (b). The scheduling table that is constructed is depicted in (c).

failure is detected, the table is no longer used; instead, all LO-criticality jobs are discarded and the remaining HI-criticality jobs are executed acording to EDF.

The entire linear program is listed in Figure 2; we now illustrate the construction of such a linear program by means of a simple example.

Example 3: We will consider a MC instance I consisting of three jobs, with parameters as depicted in Figure 3(a), with c_2's value left unspecified for now, and d_2 assumed to be larger than 5. The release dates and deadlines of these three jobs define three intervals: $I_1 = [0, 3)$; $I_2 = [3, 5)$; $I_3 = [5, d_2)$, as illustrated in Figure 3(b).

Since there are three jobs in I ($n = 3$), Constraints 1 of the LP will be instantiated to the following three inequalities, specifying that all three jobs receive adequate execution in the scheduling table $S(I)$ to execute correctly on a normal (non-degraded) processor:

$$\begin{aligned} x_{11} + x_{12} &\geq 3 \\ x_{21} + x_{22} + x_{23} &\geq c_2 \\ x_{32} &\geq 1 \end{aligned}$$

There are also three intervals I_1, I_2, and I_3. Constraints 2 of the LP will therefore yield the following three inequalities, specifying that the capacity constraints of the intervals are met:

$$\begin{aligned} x_{11} + x_{21} + x_{31} &\leq 3 \\ x_{12} + x_{22} + x_{32} &\leq 2 \\ x_{13} + x_{23} + x_{33} &\leq d_2 - 5 \end{aligned}$$

It remains to instantiate the Constraints 3, that were introduced to ensure correct behavior in the event of processor degradation. These must be separately instantiated to model the possibility of the processor degrading at the start of each of the three intervals I_1, I_2 and I_3. We consider these separately:

- **Fault at the start of I_1.** In this case, Constraints 3 is instantiated twice: once each for $t_m = 5$ and $t_m = d_2$:

$$\begin{aligned} x_{31} + x_{32} &\leq (5 - 0)\, s \\ \big(x_{21} + x_{22} + x_{23}\big) + \big(x_{31} + x_{32} + x_{33}\big) &\leq (d_2 - 0)\, s \end{aligned}$$

- **Fault at the start of I_2.** In this case, too, Constraints 3 is instantiated once each for $t_m = 5$ and $t_m = d_2$:

$$\begin{aligned} x_{32} &\leq (5 - 3)\, s \\ \big(x_{22} + x_{23}\big) + \big(x_{32} + x_{33}\big) &\leq (d_2 - 3)\, s \end{aligned}$$

- **Fault at the start of I_3.** In this case, Constraints 3 is instantiated just once, for $t_m = d_2$:

$$x_{33} \leq (d_2 - 5)\, s$$

(We note that there are nine variables and eleven constraints in this particular example.)

Continuing this example, suppose that c_2 and d_2 were 3 and 10 respectively, and s was equal to $1/2$. A possible solution to the LP would assign the x_{ij} variables the following values:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & 2 \\ 0 & 1 & 0 \end{bmatrix}$$

As a consequence, the scheduling table would be as depicted in Figure 3(c). We can easily see that this scheduling table yields a correct scheduling strategy: observe that there are three contiguous blocks of HI-criticality execution: $[0, 1)$, $[3, 4)$, and $[5, 7)$, and consider the possibility of the processor degrading at the start of each:

- If the processor failed during $[0, 1)$, then J_2 can execute over $[0, 3)$ and $[5, 8)$, while J_3 can execute over $[3, 5)$. Both HI-criticality jobs would meet thus their deadlines on the speed-0.5 processor.
- If the processor failed during $[3, 4)$, then J_3 would execute over $[3, 5)$. J_2 will have completed one unit of execution prior to the processor failing, and therefore need two additional units of execution. This it will obtain by executing over $[5, 9)$ on the speed-0.5 processor. If the processor failed during $[5, 7)$, then J_2 will have completed one unit of execution prior to the processor failing. It needs two more units, which it will obtain by executing over $[5, 9)$ on the speed-0.5 processor.

We thus see that the solution of the LP does indeed yield a feasible scheduling strategy. ■

Bounding the size of this LP. It is not difficult to show that the LP of Figure 2 is of size polynomial in the number of jobs n in MC instance I:

- The number of intervals k is at most $2n - 1$. Hence the number of $x_{i,j}$ variables is $O(n^2)$.
- There are n constraints of the form (1), and k constraints of the form (2). The number of constraints of the form (3) can be bounded from above by $(k \times n_h)$, since for each $\ell \in \{1, \ldots, k\}$, there can be no more than n_h t_m's corresponding to deadlines of HI-criticality jobs. Since $n_h \leq n$ and $k \leq (2n - 1)$, it follows that the number of constraints is $O(n) + O(n) + O(n^2)$, which is $O(n^2)$.

C. An optimization problem

Lemma 1 gives us a lower bound on the degraded speed s such that the MC instance $(\{J_1, J_2, \ldots, J_n\}, s)$ can be scheduled in a correct manner: s can be no smaller than the speed of the slowest processor upon which the HI-criticality jobs in the collection would be correctly scheduled by EDF. But is this lower bound tight? The following example illustrates that it is not:

Example 4: Consider the following three MC jobs:

J_i	a_i	c_i	d_i	χ_i
J_1	0	2	2	LO
J_2	0	1	4	HI
J_3	2	1	4	HI

It is evident that

- all three jobs are schedulable on a unit-speed processor (execute J_1 over $[0, 2)$, J_2 over $[2, 3)$, and J_3 over $[3, 4)$), and
- J_2 and J_3 are schedulable on a speed-$\frac{1}{2}$ processor (execute J_2 over $[0, 2)$, and J_3 over $[2, 4)$).

Hence MC instance $(\{J_1, J_2, J_3\}, \frac{1}{2})$ satisfies the necessary conditions identified at the beginning of Section III-B. However, there is no (non-clairvoyant) scheduling strategy that can execute this instance correctly: consider the run-time behavior in which the processor operates in normal mode over $[0, 2)$.

- If J_1 did not execute exclusively over the interval $[0, 2)$, then it misses its deadline at time-instant 2. The processor remains in normal mode.
- If J_1 did execute exclusively over the interval $[0, 2)$, then the processor enters degraded mode at time-instant 2.

In either case, the instance was not correctly scheduled despite satisfying the necessary conditions of Lemma 1. ■

It turns out that a slight modification to the linear program of Figure 2 can be used to answer the following optimization version of the schedulability problem we have considered thus far:

Given a collection of MC jobs $\{J_1, J_2, \ldots, J_n\}$, determine the ***smallest*** value of s such that the MC instance $(\{J_1, J_2, \ldots, J_n\}, s)$ can be scheduled in a correct manner. ■

To compute an answer to this question, we add the objective function

$$minimize\ s$$

to our linear program of Figure 2. That is, our modified linear program computes those values of the $x_{i,j}$ parameters that yield a scheduling strategy guaranteeing to meet all deadlines on a unit-speed processor, and HI-criticality jobs' deadlines when the degraded speed is the smallest possible; this smallest speed is the desired solution to the optimization version of our MC scheduling problem.

We implemented this modified LP and executed it on multiple randomly-generated MC instances, using standard workload-generation techniques that are widely used in real-time systems. Our observations are depicted in graphical form in Figure 4. The x-axis denotes the loading factor[3] of the HI-criticality jobs in the MC instance under consideration; the y-axis, the degraded speed s of the instance. By Lemma 1 the loading factor of the HI-criticality jobs is a lower bound on the degraded speed for which a correct scheduling strategy may exist — this lower bound is depicted as a dotted line in Figure 4. A multiplicity of instances were considered for each value of u_{HI}; the solid line depicts the average of the smallest degraded speeds (for which a correct scheduling strategy actually exists) as computed by our linear program. The dashed line depicts the average degraded speed plus one standard deviation. Although we do not claim that our simulations are extensive or comprehensive enough to draw conclusions with absolute certainty, the evidence presented in this graph does indicate that the actual minimum speed (as computed by our linear program) for which the typical randomly-generated MC instance is correctly schedulable is very close to the lower bound implied by Lemma 1.

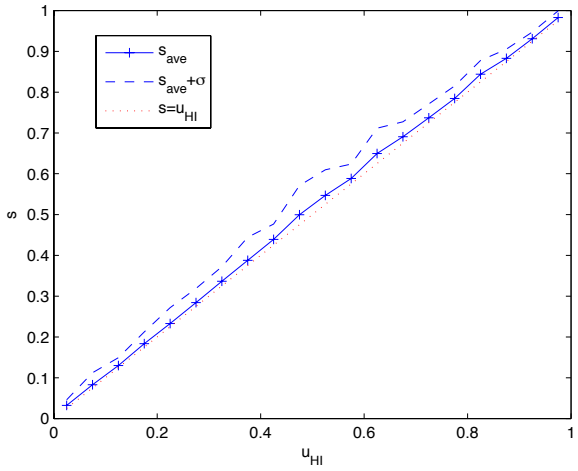


Fig. 4. Experimental evaluation: comparing lower bound on degraded speed (dotted line) with average computed value (solid line) and average plus standard deviation (dashed line).

IV. Non-preemptive scheduling

Recall that the scheduling strategy we adopted in Section III above is as follows. Given an instance I, we construct a scheduling table $S(I)$ for use while the processor is in normal mode. During run-time scheduling decisions are initially made according to this table. If at any instant it is detected that the processor has transited to faulty mode, the scheduling strategy is *immediately* switched: henceforth, only HI-criticality jobs are executed, and these are executed according to EDF. Such a scheduling strategy requires that the job that is executing at the instant of transition can be preempted, and is hence not applicable for *non-preemptive* systems. In this section, we

[3] See, e.g. [14, p. 81] for the definition of the *loading factor* of a collection of jobs; it is known that the loading factor is equal to the speed of the smallest processor upon which such a collection can be scheduled using preemptive EDF.

consider the problem of scheduling non-preemptive mixed-criticality instances.

Non-preemptivity mandates that each job receive its execution during one contiguous interval of time. Let us suppose that a LO-criticality job is executing when the processor experiences a degradation in speed. We can specify two different kinds of non-preemptivity requirements:

1) This LO-criticality job does not need to complete – it may immediately be dropped.
2) This LO-criticality job cannot be preempted and discarded – it must complete execution despite that fact that the processor has degraded and this job's completion is not required for correctness.

Although the first requirement – that the LO-criticality job may be dropped – may at first glance seem to be the more reasonable one, implementation considerations may favor the second requirement. For instance, it is possible that the LO-criticality job had been accessing some shared resource within a critical section, and preempting and discarding it would leave the shared resource in an unsafe state. We will consider both these forms of mixed-critical non-preemptivity.

It has long been known [11] that the problem of scheduling a given collection of independent jobs on a single non-preemptive processor (that does not have a faulty mode) is already NP-hard in the strong sense [11]. Since our mixed-criticality problem, under either interpretation of the non-preemptivity requirements, is easily seen to be a generalization, it is also NP-hard. In fact, although determining whether a synchronous instance of jobs can be scheduled on a non-faulty processor is easily solved in polynomial time by EDF, we show below that even this restricted problem is NP-hard for MC scheduling.

Theorem 3: It is NP-hard to determine whether there is a correct scheduling strategy for scheduling non-preemptive synchronous mixed-criticality instances.

Proof Sketch: We prove this first for the second interpretation of non-preemptivity requirements (LO-criticality jobs that have begun execution must be executed to completion), and indicate how to modify the proof for the first interpretation.

This proof consists of a reduction of the partitioning problem [7], which is known to be NP-complete, to the problem of determining whether a given non-preemptive synchronous mixed-criticality instance I can be scheduled correctly. The partitioning problem is defined as follows. *Given* a set S of n positive integers $y_1, y_2, \ldots, y_n$ summing to $2B$, *determine* whether there is a subset of S with elements summing to exactly B.

Given an instance S of the partitioning problem, we construct an instance of the synchronous mixed-criticality scheduling problem I comprised of $(n+1)$ jobs $J_1, J_2, \ldots, J_{n+1}$. The parameters of the jobs are

$$J_i = \begin{cases} (0, y_i, 5B, \text{HI}), & 1 \leq i \leq n \\ (0, B, 2B, \text{LO}), & i = n+1 \end{cases}$$

The normal processor speed is one; the degraded processor speed s is assigned a value equal to half: $s \leftarrow 1/2$.

We will show that there is a partitioning for instance S if and only if there is a correct scheduling strategy for I.

There is a partitioning for S. Let $S' \subseteq S$ denote the subset summing to exactly B. We construct our scheduling table as follows. Jobs corresponding to the elements in S' are scheduled over the interval $[0, B)$, after which J_{n+1} is scheduled over $[B, 2B)$, followed by the scheduling of the jobs corresponding to the elements in $(S \setminus S')$ over $[2B, 3B)$.

- If the processor enters faulty mode prior to time-instant B, then only the HI-criticality jobs need to complete execution; it may be verified that they will do so by their common deadline.
- If the processor enters faulty mode over $[B, 2B)$, then J_{n+1} may execute for no more than the interval $[B, 3B)$. That still leaves adequate capacity for the jobs corresponding to elements in $(S \setminus S')$ to complete execution by their deadline at $5B$, on the speed-0.5 processor.
- Otherwise, J_{n+1} completes by time-instant $2B$. That leaves adequate capacity for the jobs corresponding to elements in $(S \setminus S')$ to complete execution by their deadline at $5B$, regardless of whether the processor enters faulty mode or not.

There is *no* partitioning for S. In this case, consider the time-instant t_o at which the LO-criticality job J_{n+1} begins execution. We consider three possibilities:

- If $t_o > B$, the processor remains in normal mode but J_{n+1} misses its deadline at time-instant $2B$.
- If $t_o = B$, then the processor must have been idled for some time during $[0, B)$. If the processor were to now enter faulty mode at this time-instant t_o, job J_{n+1} will execute over $[B, 3B)$, after which the strictly more than B units of remaining HI-criticality execution would execute — this cannot complete by the deadline of $5B$ on the speed-1/2 processor.
- Now suppose that that $t_o < B$, and the processor enters faulty mode at this time-instant t_o. It must be the case that $\leq t_o$ units of execution of the HI-criticality jobs has occurred prior to time-instant t_o. Job J_{n+1} will execute over $[t_o, t_o + 2B)$, after which the at least $(2B - t_o)$ remaining units of HI-criticality work must complete. But on the speed-1/2 processor this would not happen prior to the time-instant

$$\begin{aligned} &\geq \quad t_o + 2B + 2(2B - t_o) \\ &= \quad 6B - t_o \\ &> \quad 5B \end{aligned}$$

  which means that some HI-criticality job misses its deadline.

We have thus shown that there is a correct scheduling strategy for the non-preemptive synchronous mixed-criticality instance I if and only if S can be partitioned into two equal subsets.

The proof above assumed the second interpretation of non-preemptivity requirements, in which LO-criticality jobs that begin execution need to complete even if the processor degrades.

For the first interpretation of non-preemptivity requirements (LO-criticality jobs that begin execution do not need to complete if the processor degrades while they are executing), we would modify the proof by assigning the jobs $J_1, J_2, \ldots, J_n$ a deadline of $4B$ (rather than $5B$ as above). It may be verified that this modified MC instance can be scheduled correctly if and only if the S can be partitioned into two equal subsets. ■

The intractability result of Theorem 3 above implies that in contrast to the preemptive case, we are unlikely to be able to obtain efficient (polynomial-time) optimal scheduling strategies for non-preemptive MC scheduling of even synchronous instances. We are currently working on devising, and evaluating, polynomial-time approximation algorithms for the non-preemptive scheduling of synchronous mixed-criticality systems.

V. Recurrent tasks

In Sections III and IV above, we have considered mixed-criticality (MC) systems that can be modeled as finite collections of jobs. However, many real-time systems are better modeled as collections of *recurrent processes* that are specified using, e.g., the sporadic tasks model [13], [15]. In this section, we briefly consider this more difficult problem of scheduling mixed-criticality systems modeled as collections of sporadic tasks. As with traditional (i.e., non MC) real-time systems, we will model a MC real-time system τ as being comprised of a finite specified collection of MC recurrent tasks, each of which will generate a potentially infinite sequence of MC jobs. We restrict our attention here to *implicit-deadline MC sporadic tasks*. Each task is characterized by a 3-tuple of parameters: $\tau_i = (C_i, T_i, \chi_i)$, with the following interpretation. Task τ_i generates a potentially infinite sequence of jobs, with successive jobs being released at least T_i time units apart. Each such job has a criticality χ_i, a WCET C_i, and a deadline that is T_i time units after its release. The quantity $U_i = C_i/T_i$ is referred to as the *utilization* of τ_i. An *implicit-deadline MC sporadic task system* is specified by specifying a finite number $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of such sporadic tasks, and the degraded processor speed $s < 1$ (as with MC instances of independent jobs, it is assumed that the normal processor speed is one). As with traditional (non-MC) systems, such a MC sporadic task system can potentially generate infinitely many different MC instances (collections of jobs), each instance being obtained by taking the union of one sequence of jobs generated by each sporadic task.

If unbounded ***preemption*** is permitted, then the scheduling problem for implicit-deadline MC sporadic task systems on uniprocessors is easily and efficiently solved in an optimal manner. We first derive (Theorem 4) a necessary condition for the existence of a correct scheduling strategy. We then present a scheduling strategy, *Algorithm preemptive-MC*, and prove (Theorem 5) that it is optimal.

Theorem 4: A necessary condition for MC sporadic task system (τ, s) to be schedulable by a non-clarivoyant correct scheduling strategy is that

1) the sum of the utilizations of all the tasks in τ is no larger than 1, and
2) the sum of the utilizations of the HI-criticality tasks in τ is no larger than s.

Proof: It is evident that the first condition is necessary in order that all jobs of all tasks in τ complete execution by their deadlines upon a normal processor, and that the second condition is necessary in order that all jobs of all the HI-criticality tasks in τ complete execution by their deadlines upon a degraded (speed-s) processor. ■

In order to derive a correct scheduling strategy, we first observe that using preemption we can mimic a *processor-sharing* scheduling strategy, in which several jobs are simultaneously assigned fractional amounts of execution with the constraint that the sum of the fractional allocations not exceed the capacity of the processor. (This is done by partitioning the time-line into intervals of length Δ where Δ is an arbitrarily small positive number, and using preemption within each such interval to ensure that each job that is assigned a fraction f of the processor capacity gets executed for a duration $f \times \Delta$ within this interval.)

Consider now the following processor-sharing scheduling strategy:

Algorithm preemptive-MC.

1) Initially (i.e., on the normal –non-faulty– processor), assign a share U_i of the processor to each task τ_i during each instant that is active[4].
2) If the processor transits to degraded mode at any instant during run-time, immediately discard all LO-criticality tasks and execute the HI-criticality tasks according to EDF.

Theorem 5: Algorithm preemptive-MC is an optimal correct scheduling strategy for the preemptive uniprocessor scheduling of MC sporadic task systems.

Proof: Let τ denote a MC implicit-deadline sporadic task system satisfying the necessary conditions for schedulability that have been identified in Theorem 4.

It is evident that Algorithm preemptive-MC meets all deadlines if the processor operates at its normal speed, since the processor-sharing schedule ensures that each job of each task τ_i receives exactly C_i units of execution between its release date and its deadline.

Suppose that the processor degrades at some time-instant t_o. If we were to immediately discard all LO-criticality tasks, the second necessary schedulability condition of Theorem 4 ensures that there is sufficient computing capacity on the degraded processor to continue a processor-sharing schedule in which each HI-criticality task τ_i with an active job receives a share U_i of the processor. The correctness of Algorithm preemptive-MC now follows from the existence of this processor-sharing schedule, and the optimality property of preemptive uniprocessor EDF. ■

[4]A task is defined to be *active* at a time-instant t if it has released a job prior to t and this job has not yet completed execution by time t

If preemption is forbidden, then scheduling of MC sporadic task systems becomes a lot more challenging. As with the collections of independent jobs (Theorem 3), this problem, too, can be shown to be highly intractable. We are currently working on designing efficient approximation algorithms for scheduling such systems.

VI. Conclusions

In this paper we have presented the findings of our initial research into scheduling mixed-criticality systems upon platforms that may suffer degradations in performance during run-time. Upon such platforms, the scheduling objective is to ensure that all jobs complete in a timely manner under normal circumstances, while simultaneously ensuring that more critical jobs complete in a timely manner even under degraded conditions.

The research reported in this paper can be extended in several directions. Much as the initial paper on mixed-criticality scheduling [19] gave rise to a large body of research that explored additional aspects and facets of the problems first described in [19], we are optimistic that other researchers will become enthusiastic about extensions to the research described in this paper, and will work to help solve them. We have already pointed out several interesting open problems throughout the paper; we close by briefly describing a couple of additional extensions that we consider particularly interesting.

More criticality levels. Although we have restricted our attention in this paper to just two criticality levels, it would be useful to extend the model to allow for the specification of more than two criticality levels. Such an extension gives rise to some interesting questions concerning, e.g., tradeoffs: does the processor speed at which a processor is deemed to have degraded one criticality level impact on the processor speed at which it will degrade further criticality levels? If so, what are the factors that the system designer should keep in mind in deciding what the criteria are for deeming a degradation in processor performance?

Systems that do not self-monitor. We have assumed here that a platform "knows" its execution speed at each instant during run-time; specifically, that the scheduling algorithm knows when the processor speed falls below a certain threshold. It would be particularly interesting and important to derive algorithms for scheduling mixed-criticality systems upon platforms that do not have such self-awareness; such scheduling algorithms would need to guarantee that all jobs meet their deadlines upon a normal processor and that all HI-criticality jobs meet their deadlines on a degraded processor, *without* knowing during run-time whether the processor is normal or degraded.

References

[1] S. Baruah. Certification-cognizant scheduling of tasks with pessimistic frequency specification. In *Proceedings of the IEEE Symposium on Industrial Embedded Systems (SIES)*. IEEE Press, June 2012.

[2] S. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, Vienna, Austria, 2011. IEEE Computer Society Press.

[3] S. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, Prague, Czech Republic, July 2008. IEEE Computer Society Press.

[4] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

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

[6] P. Ekberg and W. Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012. IEEE Computer Society Press.

[7] M. Garey and D. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman and company, NY, 1979.

[8] N. Guan, P. Ekberg, M. Stigge, and W. Yi. Effective and efficient scheduling for certifiable mixed criticality sporadic task systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, Vienna, Austria, 2011. IEEE Computer Society Press.

[9] N. Karmakar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[10] L. Khachiyan. A polynomial algorithm in linear programming. *Dokklady Akademiia Nauk SSSR*, 244:1093–1096, 1979.

[11] J. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[12] H. Li and S. Baruah. Global mixed-criticality scheduling on multiprocessors. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012. IEEE Computer Society Press.

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

[14] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, Inc., Upper Saddle River, New Jersey 07458, 2000.

[15] A. Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983. Available as Technical Report No. MIT/LCS/TR-297.

[16] R. Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012. IEEE Computer Society Press.

[17] P. J. Prisaznuk. Integrated modular avionics. In *Proceedings of the IEEE 1992 National Aerospace and Electronics Conference (NAECON 1992)*, volume 1, pages 39–45, May 1992.

[18] F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with FP. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012. IEEE Computer Society Press.

[19] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the Real-Time Systems Symposium*, pages 239–243, Tucson, AZ, December 2007. IEEE Computer Society Press.

[20] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012. IEEE Computer Society Press.