Implementing Mixed-criticality Systems Upon a Preemptive Varying-speed Processor

Zhishan Guo and Sanjoy K. Baruah

University of North Carolina, Chapel Hill, NC, USA, `{zsguo,baruah}@cs.unc.edu`

Abstract

A *mixed criticality (MC)* workload consists of components of varying degrees of importance (or "criticalities"); the more critical components typically need to have their correctness validated to greater levels of assurance than the less critical ones. The problem of executing such a MC workload upon a preemptive processor whose effective speed may vary during run-time, in a manner that is not completely known prior to run-time, is considered.

Such a processor is modeled as being characterized by several execution speeds: a normal speed and several levels of degraded speed. Under normal circumstances it will execute at or above its normal speed; conditions during run-time may cause it to execute slower. It is desired that all components of the MC workload execute correctly under normal circumstances. If the processor speed degrades, it should nevertheless remain the case that the more critical components execute correctly (although the less critical ones need not do so).

In this work, we derive an optimal algorithm for scheduling MC workloads upon such platforms; achieving optimality does not require that the processor be able to monitor its own run-time speed. For the sub-case of the general problem where there are only two criticality levels defined, we additionally provide an implementation that is asymptotically optimal in terms of run-time efficiency.

2012 ACM Subject Classification Real-Time Schedulability

Keywords and phrases Mixed criticalities, varying-speed processor, preemptive uniprocessor scheduling

Digital Object Identifier 10.4230/LITES-v001-i002-a003

Received 2014-04-23 **Accepted** 2014-08-26 **Published** 2014-11-17

Editor Neil Audsley

1 Introduction

As stated in the title, this paper is concerned with the implementation of *mixed-criticality systems* upon *varying-speed processors*. We start out by explaining these terms.

Varying-speed CPUs. Due to cost and related considerations, there is an increasing trend in embedded computing towards implementing safety-critical systems upon commercially available general-purpose processors (commonly known as *commercial off-the-shelf* or *COTS* processors). The special-purpose processors previously used in implementing safety-critical systems were designed to be highly predictable in the sense that tight bounds on the run-time behavior of a system could be *a priori* determined during system design time itself. However, such design-time predictability is difficult to achieve with COTS processors that are typically engineered to provide good average-case performance rather than worst-case guarantees. Such design-time predictability is nevertheless essential for safety-critical functionalities whose correctness must be validated to very high levels of assurance prior to system deployment. In this paper, we focus upon one aspect of guaranteeing real-time performance upon COTS processors despite their inherent unpredictability: *worst-case execution time (WCET)*.

The WCET abstraction plays a central role in the analysis of real-time systems. For a specific piece of code and a particular platform upon which this code is to execute, the WCET of the code denotes (an upper bound on) the amount of time the code takes to execute upon the platform.

© Zhishan Guo and Sanjoy K. Baruah;
licensed under Creative Commons Attribution 3.0 Germany (CC BY 3.0 DE)

Leibniz Transactions on Embedded Systems
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Determining the exact WCET of an arbitrary piece of code is provably an undecidable problem. Devising analytical techniques for obtaining tight upper bounds on WCET is currently a very active area of research, and sophisticated tools incorporating the latest results of such research have been developed (see [16] for an excellent survey). WCET tools require that some assumptions be made about the run-time behavior of the processor upon which the code is to execute; for example, the *clock speed* of the processor during run-time must be known in order to be able to determine the rate at which instructions will execute. However, conditions during run-time, such as changes to the ambient temperature, the supply voltage, etc., may result in variations in the clock speed. For instance, a system may be designed to have its CPU clock speed(s) reduced into a certain value temporarily whenever it is detected that core temperature gets higher than a threshold. Such variation is likely to be further exacerbated in the future, with the increasing trend in computer architecture towards *Globally Asynchronous Locally Synchronous*, or GALS, circuit designs. In order to be able to guarantee that the values they compute are correct under all run-time conditions, a WCET tool must make the most pessimistic assumptions regarding clock speed: that during run-time *the clock speed takes on the lowest possible value.* If this lowest possible value is highly unlikely to be reached in practice during actual runs, then a significant under-utilization of the CPU's computing capacity will be observed during run-time.

Mixed-criticality Systems. In safety-critical hard-real-time systems, there is little that can be done about such under-utilization of platform resources. But as stated above, another increasing trend in embedded computing is the move towards mixed-criticality (MC) systems, in which functionalities of different degrees of importance or *criticalities* are implemented upon a common platform. 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 (see [5] for a review of some of this work). The typical approach has been to validate the correctness of highly critical functionalities under *more pessimistic assumptions* than the assumptions used in validating the correctness of less critical functionalities. For instance, a piece of code may be characterized by a larger WCET in the more pessimistic analysis and a smaller WCET in the "normal" (less pessimistic) analysis [15]. All the functionalities are expected to be demonstrated correct under the normal analysis, whereas the analysis under the more pessimistic assumptions need only demonstrate the correctness of the more critical functionalities.

The results reported in this paper fall within the same framework as this prior work. However, rather than considering variations in estimating WCET, we assume that each piece of code is characterized by a single WCET, and focus instead on the variations in run-time speed of the processing platform. As in earlier work, the mixed-criticality nature of the system that is considered in this paper is reflected 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 under pathological conditions which, while extremely unlikely to occur in practice, cannot be entirely ruled out. 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 (single) WCET, a deadline, and a criticality level $\in \{1, 2, \ldots, m\}$ expressing its degree of importance, with larger values denoting greater importance. We desire to schedule the system upon a single preemptive processor. This processor is a varying-speed one that is characterized by a sequence of m speeds $1 = s_1 > s_2 > \ldots > s_m$. The run-time behavior of this processor is as follows: while under normal circumstances it completes at least one unit of execution during each time unit (equivalently, it executes as a speed-1, or faster, processor), its speed may degrade to lower values during run-time. The precise manner in which the speed will vary during run-time is not *a priori*

known. We seek a scheduling strategy that for all $l, 1 \leq l \leq m$ *guarantees to correctly execute all those jobs that have criticality* $\geq l$*, provided the processor speed never falls below* s_l *during run-time.*

The following example illustrates this model.

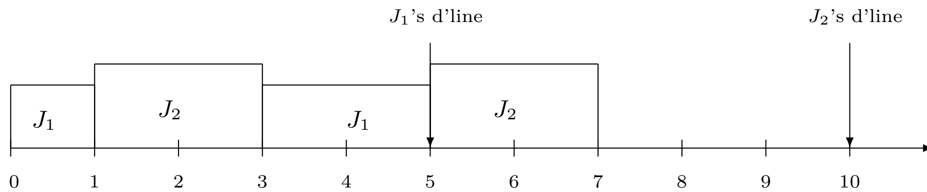
▶ **Example 1.** Consider the following collection of two jobs, to be scheduled on a preemptive processor with specified speeds $s_1 = 1$ and $s_2 = \frac{1}{2}$:

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

An Earliest Deadline First (EDF) [12] 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 to some speed $s' < \frac{4}{7}$, or ≈ 0.55) starting at time-instant 3: in the EDF schedule J_2 would obtain merely $(10 - 3) \times s' < 4$ units of execution prior to its deadline at time-instant 10. We therefore conclude that EDF does not schedule this system correctly.

An alternative scheduling strategy could instead execute jobs as follows on a normal (speed-1) processor: J_1 over 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 to a speed < 1 at any instant during this execution then the processor immediately switches to executing J_2 until it completes.

It may be verified that this scheduling strategy will result in J_2 completing by its deadline regardless of when (if at all) the processor degrades to any speed $\geq \frac{1}{2}$, and in both deadlines being met if the processor remains normal (or degrades at any instant ≥ 5).

Contributions and Organization. As mixed-criticality (MC) systems increasingly come to be implemented upon commodity processors, we believe it imperative that real-time scheduling theory provide an understanding of how to implement these systems to meet the twin goals of providing *correctness guarantees at high levels of assurance* to the more critical functionalities while simultaneously making *efficient use of platform resources*. As discussed above, commodity processors tend to execute at varying speeds as ambient conditions change; in order to make correctness guarantees at very high levels of assurance upon such varying-speed processors, it may be necessary to consider the possibility that the processor is executing at a very low speed. In this paper, we seek to define a formal framework for the scheduling-based analysis of MC systems that execute upon CPUs which may be modeled as varying-speed processors. To this end, in Section 2 we describe a very simple model for representing MC systems. In Section 3 we propose, analyze, and evaluate an algorithm for the preemptive uniprocessor scheduling of MC systems that can be represented using this model. In Section 4 we consider the special case where there are only two criticality levels (such MC systems have been called *dual-criticality*

systems in the literature), and provide a more efficient algorithm for this restricted case. In Section 5, we discuss the computational complexity of the problem when preemption is forbidden. In Section 6, recurrent tasks are considered and a scheduling strategy is provided when unbounded preemption is permitted. We conclude in Section 7 by placing this work within the larger context of mixed-criticality scheduling, and briefly enumerate some important and interesting directions for further research.

Relationship to Prior Work. The years since Vestal's seminal paper in 2007 [15] have seen a large amount of research in mixed-criticality scheduling. Much of this research considers a model in which each job is characterized by multiple WCETs. The results from this prior research can be directly applied to our problem, in the following manner. Consider a job in our setting that has WCET c and is being scheduled on a varying-speed processor with normal speed $s_1 = 1$ and degraded speeds $s_2, \ldots, s_m$. This job may be represented in the multiple-WCET model as a job with a normal WCET of c and more pessimistic WCETs of $c/s_2, \ldots, c/s_m$; if all jobs execute for no more than their normal WCETs then all jobs should execute correctly, while if some jobs execute beyond their normal WCETs then only some of the jobs (those with criticality levels exceeding a particular value) are required to execute correctly. It is not difficult to show that the algorithms proposed in prior work for scheduling MC systems with multiple WCET specifications can be used to schedule this transformed system, and that the resulting scheduling strategy correctly schedules our (original) system upon the varying-speed processor. Hence, all the problems considered in this paper could in principle be solved by simply transforming to the earlier, multiple-WCET, model, and applying the previously-proposed solution techniques.

However, in [2] we showed that one can sometimes do better than such an approach. This was observed to be because the problem we are considering here, of MC scheduling on varying-speed processors, is *simpler* (from a computational complexity perspective) than the previously-considered problem of MC scheduling with multiple-WCETs specified. For instance, whereas determining preemptive uniprocessor feasibility for a collection of independent MC jobs specified according to the multiple-WCET model is known [3] to be NP-hard in the strong sense, in Section 3 we will present an optimal polynomial-time algorithm for solving the same problem in our model. For the case of dual-criticality systems of implicit-deadline sporadic tasks on preemptive uniprocessors, a speedup lower bound of 4/3 had been established [4] for the multiple-WCETs model, whereas [2] had provided an optimal (speedup-1) algorithm.

This paper extends our recent work [2] in several significant directions. First (as stated above), the results in [2] were only shown to hold for mixed-criticality systems that are implemented upon varying-speed processors for which just two speeds are specified; this paper extends these results to be applicable to mixed-criticality systems implemented upon varying-speed processors with an arbitrary number of speeds specified. Second, [2] had derived a linear-programming (LP) approach to solving the problem in the two-level case (thereby establishing that the problem could be solved in polynomial time). In this paper, we derive an altogether different algorithm for solving the two-speed case, that has a worst-case run-time of $\mathcal{O}(n \log n)$ where n is the number of jobs in the instance; this is more efficient than the earlier LP-based approach. And finally, the concept of *self-monitoring* by processors was introduced [8] as a means of distinguishing between processors that do or do not "know" at each instant during run-time, what their precise speeds are. While the algorithms derived in [2] assume that the processor possesses the self-monitoring property, the algorithms we derive here do not require this property to hold.

A Note. Although we have chosen to model the problem in terms of real-time jobs executing on varying-speed processors, the model (and our results) are also applicable to the transmission of

time-sensitive data on potentially bandwidth-varying communication media. Specifically, they are particularly relevant to data-communication problems in which time-sensitive data and data-streams must be transmitted over communications media which can provide a high bandwidth under most circumstances but can only *guarantee* some lower bandwidths: the high bandwidth would correspond to the normal processor speed, and the lower bandwidths to the degraded speeds. 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.

2 Model

We start out considering mixed-criticality systems that can be modeled as collections of *independent jobs*; a model for *recurrent tasks* is considered in Section 6. In our model, a mixed-criticality real-time workload consists 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 $\chi_i \in \{1, 2, \ldots, m\}$. Note that this WCET c_i is measured based upon some constant unit-speed processor – a job with WCET of c_i may require a period of length c_i/s when executing on a speed-s processor.

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 increasing order (i. e., $t_j < t_{j+1}$ 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 $I_1, I_2, \ldots, I_k$, with I_j denoting the interval $[t_j, t_{j+1})$.

A mixed-criticality *instance* I is specified by specifying

- a finite collection of MC jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$, and
- a varying-speed processor that is characterized by a normal speed s_1 (without loss of generality, assumed to be 1) and some specified *degraded processor speeds* $s_2, \ldots, s_m$ in strictly decreasing order; i. e., $s_m < s_{m-1} < \ldots < s_2 < 1$.

The interpretation is that the jobs in $\mathcal{J}$ are to execute on a single shared processor that has m modes: a *normal* mode and $(m-1)$ *degraded* modes. In the normal mode, the processor executes as a unit-speed processor and hence completes one unit of execution per unit time, whereas in degraded mode l it completes fewer than s_{l-1}, but at least s_l, units of execution per unit time, for $l = 2, \ldots, m$.

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

▶ **Definition 2** (Correct Scheduling Strategy)**.** A scheduling strategy for MC instances is *correct* if it possesses the property that upon scheduling any MC instance $I = (\mathcal{J} = \{J_1, J_2, \ldots, J_n\}, s_1, \ldots, s_m)$, each job J_i completes by its deadline if the processor executes at speeds $\geq s_{\chi_i}$ throughout its scheduling window $[a_i, d_i)$.

3 A Scheduling Algorithm

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, prior to run-time we will construct a scheduling table $S(I)$ which prescribes the amounts of execution to be received by each job during each interval. During run-time, scheduling decisions are made according to this scheduling table. Amounts within each interval are executed in the decreasing order of criticality

levels (greater criticality first). A job is dropped at its deadline if it has not completed execution by then. Note that we do not discard a job with criticality level lower than ℓ even when the processing speed is detected to have fallen to some value in the range $(s_{\ell+1}, s_\ell])$ – such a mechanism improves the likelihood of lower criticality jobs meeting their deadlines despite processor degradation[1]

In the remainder of this section we present, and prove the correctness of, a simple linear-programming based algorithm for constructing the scheduling table $S(I)$ optimally. By *optimal*, we mean that if there is a correct scheduling strategy (Definition 2 above) for an instance I, then the scheduling strategy described above is a correct scheduling strategy with the scheduling table we will construct. We start out identifying the following (obvious) necessary condition for MC-schedulability:

▶ **Lemma 3.** *In order that a correct scheduling strategy exists for MC instance $I = (\mathcal{J}, s_1, \ldots, s_m)$, it is* necessary *that for each criticality level $l = 1, \ldots, m$, EDF correctly schedules all the jobs in I with criticality level $\geq l$ upon a speed-s_l uniprocessor.*

Given any instance I, it can be efficiently determined whether I satisfies the necessary conditions of Lemma 3: for each l, simply simulate the EDF scheduling of all the jobs in I with criticality-level $\geq l$ upon a speed-s_l processor. In the remainder of this section, let us therefore assume that any instance under consideration satisfies these necessary conditions. (I.e., any instance that fails these conditions can obviously not have a correct scheduling strategy, and is therefore flagged as being unschedulable.)

Given an MC instance $I = (\{J_1, J_2, \ldots, J_n\}, s_1, \ldots, s_m)$ that satisfies the conditions of Lemma 3, we now describe how to construct a linear program (LP) such that a feasible solution for this linear program can be used to construct scheduling table $S(I)$.

To construct our linear program we define $n \times k$ variables $x_{i,j}$, $1 \leq i \leq n; 1 \leq j \leq k$. 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.

The following n constraints specify that each job receives adequate execution in the normal schedule:

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

while the following k constraints specify the capacity constraints of the intervals:

$$\left(\sum_{i=1}^{n} x_{i,j} \right) \leq s_1(t_{j+1} - t_j), \text{ for each } j, 1 \leq j \leq k. \tag{2}$$

Within each interval, jobs will be executed in the priority order of their criticality levels; i. e., amounts from higher criticality level jobs get executed first. (That is, the interval I_j will have a block of level-m criticality execution of duration $\sum_{i:\chi_i=m}^{n} x_{i,j}$, followed by blocks of l-criticality execution of duration $\sum_{i:\chi_i=l} x_{i,j}$ with l from $m-1$ down to 1, in order.) 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 with respect to degraded conditions – that the higher-criticality jobs complete execution even in the event of the processor degrading into corresponding modes.

[1] An example of such benefit will be shown in the execution analysis (Item 2) of Example 4, where J_2 with criticality level of 2 may meet its deadline despite the processor speed falling to below s_2 during $[a_2, d_2)$.

Since within each interval, amounts are executed in decreasing order of criticality level, we observe that the worst-case scenarios occur when the processing speed drops at the very *beginning* of a time interval, since that would leave the minimum computing capacity. For each $\{p, l\}$, $1 \le p \le k, 2 \le l \le m$, we represent the possibility that the processor degrades into speed-s_l mode at the start of the interval I_p in the following manner:

(i) Suppose that the processor degrades into speed-s_l mode at time-instant t_p; i. e., the start of the interval I_p. Henceforth, only jobs of criticality $\ge l$ must be fully executed in order to meet their deadlines.

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

(iii) This is ensured by writing a constraint

$$\left(\sum_{i | (\chi_i \ge l) \wedge (d_i \le t_q)} \left(\sum_{j=p}^{q-1} x_{i,j} \right) \right) \le s_l (t_q - t_p). \tag{3}$$

Note that for any job J_i with $d_i \le t_q$, $\left(\sum_{j=p}^{q-1} x_{i,j}\right)$ represents the remaining execution requirement of job J_i at time-instant t_p. The outer summation on the left-hand side is simply summing this remaining execution requirement over all the jobs of criticality $\ge l$ that have deadlines at or prior to t_q.

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

(v) The Constraints (3) above only prevent missing deadlines after t_p when the (degraded) processor is continually busy over the interval between t_p and the missed deadline; what about deadline misses when the processor is not continually busy over this interval (and the right-hand side 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 jobs of criticality $\ge l$ – 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_l processor. But this would contradict our assumption that the instance passes the necessary conditions of Lemma 3, i. e., all the jobs of criticality $\ge l$ together (and therefore, every subset of these jobs) execute successfully on a speed-s_l processor.

The entire linear program is listed in Figure 1, and the steps of our LP-based table-driven mixed-criticality scheduling approach, titled Algorithm TDMC-LP, is described in Figure 2.

It is evident that during run-time Algorithm TDMC-LP is performing a typical interval-by-interval execution – unless idleness is detected, no amount of execution that is assigned in later intervals can be "promoted" (executed in an earlier interval).

Note that due to processor degradation, it is possible that some amounts of execution that were assigned to an interval may not have completed by the end of the interval. In such a case, we do not simply drop these execution amounts, but pass them over into the subsequent interval. The reason for this additional modification during run time is that Constraints (3) only provide guarantees as to the total amount of execution provided for each job *until its deadline*. This can be done by adding the unfinished part of the amounts into the corresponding rows in the column of the scheduling table at the end of each interval (as described in Step 2b)[2]. The rationale behind such maintenance during run-time will also be shown in Example 4.

[2] Note that here $Ex(i, j)$ does not denote the total execution *time* of job J_i within Interval I_j – the processing speed during run-time needs to be considered as well.

Given: MC instance $(\{J_1, J_2, \ldots, J_n\}, s_1, \ldots, s_m)$, with 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 x_{ij} variables, $i = 1, \ldots, n, j = 1, \ldots, k$ satisfying the following **constraints:**

- For each i, $1 \leq i \leq n$,

$$\left(\sum_{j \mid t_j \geq a_i \wedge d_i \geq t_{j+1}} x_{i,j} \right) \geq c_i. \tag{1}$$

- For each j, $1 \leq j \leq k$,

$$\left(\sum_{i=1}^{n} x_{i,j} \right) \leq s_1(t_{j+1} - t_j). \tag{2}$$

- For each p, $1 \leq p \leq k$, for each l, $2 \leq l \leq m$, and for each q, $p < q \leq (k+1)$

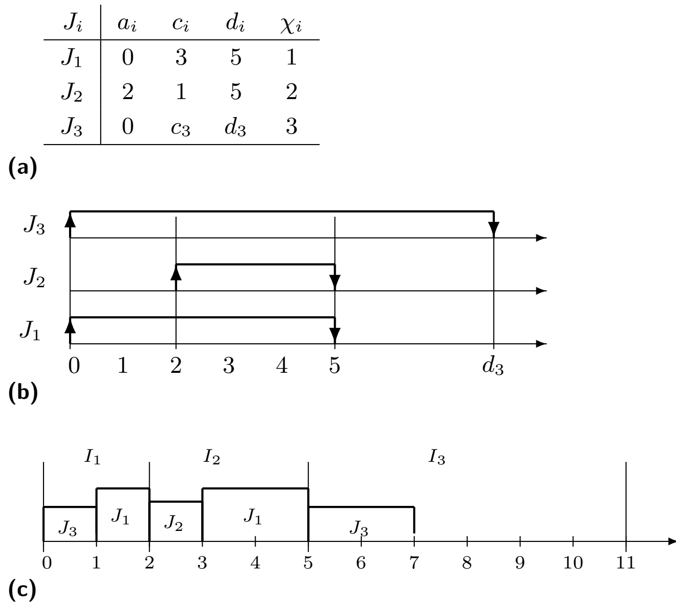
$$\left(\sum_{i \mid (\chi_i \geq l) \wedge (d_i \leq t_q)} \left(\sum_{j=p}^{q-1} x_{i,j} \right) \right) \leq s_l(t_q - t_p). \tag{3}$$

Figure 1 Linear program for constructing the scheduling table.

Given: $J = \cup_{i=1}^{n}\{J_i\}$ to be scheduled on a varying-speed processor with speed thresholds $s_1, \ldots, s_m$.

- Construct the scheduling table S according to Figure 1, with $x_{i,j}$ denoting the amount of execution assigned to job J_i during the interval I_j, for each pair (i, j).
- **For** each interval $I_j, j = 1$ up to k:
  1. Higher-criticality execution is performed before lower-criticality ones within each interval, while amounts with the same criticality level may be executed in any order.
  2. At the end of the interval; i. e., at time $t = t_j$
     a. **If** t_j is some unfinished job's deadline, **then** the job is dropped; this is indicated by setting $x_{i,j} \leftarrow -1$ $\forall i$ for which $d_i = t_j$.
     b. Other unfinished executions (if any) need to be carried over into the next interval; i. e., $\forall i$ such that $d_i > t_j$, $x_{i,j+1} \leftarrow x_{i,j+1} + x_{i,j} - Ex(i, j)$, where $Ex(i, j)$ denotes the *amount* of execution that job J_i received within Interval I_j.
  3. Whenever an idleness is detected, we may execute the (released) jobs with amounts assigned to later interval(s) in the same priority order described in Step 1.

Figure 2 Basic steps of the proposed scheduling algorithm TDMC-LP.



J_i	a_i	c_i	d_i	χ_i
J_1	0	3	5	1
J_2	2	1	5	2
J_3	0	c_3	d_3	3

Figure 3 Illustrating Example 4. The jobs are listed in (a), and depicted graphically in (b). The scheduling table that is constructed is depicted in (c).

(We also point out that the execution order when an idleness is detected, as described in Step 3, represents an optimization in run-time behavior that has nothing to do with correctness – the proof of Theorem 5 will go through even if the processor is left idled until the end of such an interval.)

Before proving its correctness and optimality, we first illustrate the operation of Algorithm TDMC-LP by means of a simple example.

▶ **Example 4.** We will consider a MC instance I consisting of three jobs with parameters as depicted in Figure 3(a), with c_3's value left unspecified for now, and d_3 assumed to be larger than 5.

The release dates and deadlines of these three jobs define three intervals: $I_1 = [0, 3)$; $I_2 = [3, 5)$; $I_3 = [5, d_3)$, 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:

$$\begin{aligned} x_{11} + x_{12} &\geq 3; \\ x_{22} &\geq 1; \\ x_{31} + x_{32} + x_{33} &\geq c_3. \end{aligned}$$

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:

$$\begin{aligned} x_{11} + x_{21} + x_{31} &\leq 2; \\ x_{12} + x_{22} + x_{32} &\leq 3; \\ x_{13} + x_{23} + x_{33} &\leq d_3 - 5. \end{aligned}$$

It remains to instantiate the Constraints (3), that were introduced to ensure correct behavior in the event of processor degradation. In this example there are three criticality levels, and thus a need to consider degradation cases of both speed-s_2 and speed-s_3. 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:

- **Degradation at the start of I_1.** In this case, Constraints (3) is instantiated three times: speed-s_2 for $t_m = 5$, and both speed-s_2 and speed-s_3 for $t_m = d_3$:

$$\begin{aligned} x_{21} + x_{22} &\leq (5-0)\, s_2; \\ \big(x_{21} + x_{22} + x_{23}\big) + \big(x_{31} + x_{32} + x_{33}\big) &\leq (d_3 - 0)\, s_2; \\ x_{31} + x_{32} + x_{33} &\leq (d_3 - 0)\, s_3. \end{aligned}$$

- **Degradation at the start of I_2.** This case is similar as the above one that Constraints (3) is instantiated once for $t_m = 5$ and twice for $t_m = d_3$:

$$\begin{aligned} x_{22} &\leq (5-2)\, s_2; \\ \big(x_{22} + x_{23}\big) + \big(x_{32} + x_{33}\big) &\leq (d_3 - 2)\, s_2; \\ x_{32} + x_{33} &\leq (d_3 - 2)\, s_3. \end{aligned}$$

- **Degradation at the start of I_3.** In this case, Constraints (3) is instantiated twice, for $t_m = d_3$ with speeds s_2 and s_3:

$$\begin{aligned} x_{33} &\leq (d_3 - 5)\, s_2; \\ x_{33} &\leq (d_3 - 5)\, s_3. \end{aligned}$$

  (Note that there are nine variables and fourteen constraints in this particular example.)

Continuing with this example, suppose that c_3 and d_3 are 3 and 11 respectively, with degraded speeds $s_2 = 1/2$ and $s_3 = 1/3$. A possible solution to the LP would assign the x_{ij} variables the following values:

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

As a consequence, the scheduling table would be as depicted in Figure 3(c).

We can see that this scheduling table yields a correct scheduling strategy: observe that there are three contiguous blocks of execution of criticality-level 2 or greater: $[0, 1)$, $[2, 3)$, and $[5, 7)$, and consider the possibility of the processor degrading during each:

- If the processor degrades to speed-s_2 during $[0, 2)$, then J_3 will execute over $[0, 2)$ and $[5, 9)$, while J_2 can execute over $[2, 4)$. Both jobs of criticality ≥ 2 would thus meet their deadlines on the speed-1/2 processor. J_1 is executed over $[4, 5)$ and dropped at $t = 5$.
- If the processor degrades to speed-s_3 during $[0, 2)$, then for the first interval $[0, 2)$, J_3 will be executed. However the assigned amount $x_{31} = 1$ may not be finished in case the processor degrades early, say at $t = 0$. As a result, the scheduling table needs to be updated at time $t = 2$ according to Step 2b in Figure 2: $x_{32} \leftarrow (0 + 1 - 2/3)$, or 1/3. J_3 will therefore get to execute over $[2, 3)$ and $[5, 11)$, and meet its deadline, on the speed-1/3 processor. Time interval $[3, 5)$ will be used to execute J_2, and both J_1 and J_2 will be dropped at time $t = 5$ in the worst case, leaving x_{12} and x_{22} the value of -1 for reference. In case the processor degrades to speed-s_3 late, say at $t = 0.5$ (while remaining at unit-speed beforehand), the assigned amount $x_{31} = 1$ can be finished upon $t = 2$, and thus although under a slowest speed condition, J_2 may finish on time be executing over $[2, 5)$.

- If the processor degrades to a speed of either s_2 or s_3 during $[2, 5)$, then J_2 would execute prior to J_1 within this interval and gets finished on time. Job J_3 will not continue its execution until $t = 5$ since $x_{32} = 0$ – it only needs two additional units of execution which will be obtained by executing over the third interval $[5, 9)$.
- If the processor degrades to speed-s_2 (or s_3) during $[5, 7)$, J_3 will still meet its deadline since it has completed one unit of execution prior to the processor degradation – it needs two more units, which will be obtained by executing over $[5, 9)$ (or $[5, 11)$) on the speed-1/2 (or 1/3) processor.

We thus see that the solution of the LP does indeed yield a feasible scheduling strategy according to the proposed run time strategies in TDMC-LP. ◀

Observe that Algorithm TDMC-LP is performing "best-effort execution" – it only discards a job if it has not completed by its deadline, and not merely because a processor degradation is detected. We now formally show that it is guaranteed that the assigned execution amounts with criticality level no lower than ℓ will nevertheless get executed so long as processing speed remains at least as large as s_ℓ (as required under the correctness definition).

▶ **Theorem 5.** *Algorithm* TDMC-LP *is correct.*

Proof. The proof is by contradiction. Assume that some job J_i with criticality level χ_i has not completed by its deadline $d_i = t_q$ (at the end of Interval I_{q-1}), while the processor remains at (or above) a speed of s_{χ_i} over the interval $[a_i, d_i)$.

From constraints (3), we know that total assigned amounts of execution with criticality level no lower than χ_i for intervals that lie within $[a_i, d_i)$ cannot exceed $s_{\chi_i} \times (d_i - a_i)$. Given the fact that no amount with lower criticality level(s) can be executed within the interval $[a_i, d_i)$ (since else J_i would have been assigned and executed during the execution of lower criticality amounts), there must be some "carry-in" amounts of execution with criticality level no lower than χ_i due to Step 2b. Let t_p denote the end of the last interval (before a_i) with either idleness or some execution of amounts with criticality level lower than χ_i (so that no amount assigned before t_p with criticality level $\geq \chi_i$ can be "carried-in"). It is now evident that Constraints (3) must be violated for Interval $[t_p, t_q)$ under speed s_{χ_i}. ◀

▶ **Theorem 6.** *Algorithm* TDMC-LP *is* optimal – *whenever it fails to maintain correctness, no other algorithm can.*

Proof. From Theorem 5, Algorithm TDMC-LP fails only when there is no feasible solution to the LP described in Figure 1. Since the three set of constraints are all necessary ones according to Lemma 3, violations of any of them indicates that the given instance is *not schedulable* under some circumstances (e. g., speed performances during run-time). Thus no other algorithm can maintain correctness as well. ◀

Bounding the Size of This LP. It is not difficult to show that the LP of Figure 1 is of size polynomial in the number of jobs n in MC instance I as well as the number of criticality levels m:

- 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 (nkm), since for each $p \in \{1, \ldots, k\}$, there can be no more than n t_q's corresponding to deadlines of jobs. Since $k \leq (2n - 1)$, it follows that the number of constraints is $O(n) + O(n) + O(n^2 m)$, which is $O(n^2 m)$.

Since it is known [10, 9] that a linear program can be solved in time polynomial in its representation, it follows that our algorithm for generating the scheduling tables for a given MC instance I takes time polynomial in the representation of I.

4 The Two-criticality-level Case

In this section, we revisit the same restricted version of the problem that was addressed in [2], and derive a more efficient algorithm for solving it. That is, we consider *dual*-criticality systems executing on a variable-speed processor characterized by just two speeds: a normal speed (assumed as 1) and a degraded speed (designated as s, with $s < 1$). We use the standard designations of LO and HI to denote the lower and higher criticality levels respectively. We propose an alternative method to the Linear Programming approach presented in [2] (and extended for > 2 levels in Section 3 above) for constructing the scheduling table, and show that this new method is computationally very efficient.

At a high level, our algorithm is organized in a manner similar to the one described in Section 3: Given a dual-criticality MC instance I, we will first construct a *scheduling table* $S(I)$, and then make run-time job-dispatch decisions in a manner that is compliant with this scheduling table.

To construct the scheduling table, we first identify (Step 1 below) the latest time intervals during which the HI-criticality jobs must execute if they are to complete execution on a degraded processor; having identified these intervals, we construct (in Step 2) an EDF schedule for the HI-criticality jobs in these intervals.

Step 1. *Considering only the* HI*-criticality jobs in the instance, determine the intervals during which the jobs would execute upon a speed-s processor, if*

1. *each job executes for its* HI*-criticality WCET,*
2. *execution occurs as late as possible.*

It is evident that these intervals may be determined by considering the jobs in non-increasing order of their deadlines (i. e., latest deadline first), and taking the cumulative execution requirements of these jobs. These intervals may therefore be determined in $\mathcal{O}(n_{\text{HI}} \log n_{\text{HI}})$ time (which comes from the time complexity of EDF), where n_{HI} denotes the number of HI-criticality jobs.

Step 2. *Construct an EDF schedule for the* HI*-criticality jobs upon a preemptive processor that has speed s during the intervals determined in Step 1 above, and speed zero elsewhere.*

It follows from the optimality property[3] of EDF that if this step fails to ensure that each HI-criticality job receives adequate execution prior to its deadline, then no scheduling algorithm can guarantee correctness (see Definition 2) for this instance. We would therefore ***report failure***: this MC instance is not feasible. The remainder of this section assumes that Step 2 above was successful in completing each HI-criticality job prior to its deadline.

We now describe how to use this EDF schedule to construct the scheduling table – recall that this scheduling table is used for job dispatch decisions upon both the normal and degraded processor, and is therefore constructed assuming a normal-speed (i. e., speed-1) processor.

Step 3. *To construct the scheduling table, partition the time-line over $[\min_i\{a_i\}, \max_i\{d_i\}]$ into the k intervals $I_1, I_2, \ldots, I_k$. (Recall, from Section 2, that these are the intervals defined by the release dates and deadlines of all the jobs –* LO*-criticality and* HI*-criticality.)*

3.1 *For each* HI*-criticality job J_i and each interval I_ℓ in which it is scheduled in the EDF schedule constructed in Step 2 above, execute J_i within this interval for an amount x_{i_ℓ} which equals*

[3] Although the optimality proof of EDF in [12], which is based on a swapping argument, assumes that the processor speed remains constant, it is trivial to extend the proof to apply to processors that are only available during limited intervals, or indeed to arbitrary varying-speed processors.

J_i	a_i	c_i	d_i	χ_i
J_1	1	2	10	HI
J_2	5	1	8	HI
J_3	6	2	15	HI
J_4	0	4	6	LO
J_5	1	2	10	LO
J_6	10	3	13	LO

Figure 4 All jobs considered in Example 7, where a_i, c_i, and d_i stands for release date, WCET, and deadline respectively.

to the amount of execution that J_i is allocated during Interval I_ℓ in the EDF scheduled constructed in Step 2 above.

3.2 *Assign* LO*-criticality jobs by simulating the EDF-scheduling of the* LO*-criticality jobs in the remaining capacity of the scheduling table – i. e., in the durations that are not already allocated to the* HI*-criticality jobs during Step 3.1 above.*

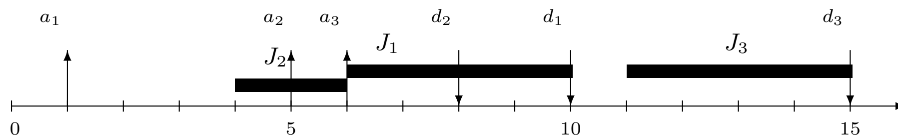
3.3 *If during this EDF simulation there is any capacity left over within an interval (because the supply of currently-active* LO*-criticality jobs has been exhausted), then move over* HI*-criticality jobs, that had been assigned to later intervals in the scheduling table during Step 3.1 above, into the current interval. In so doing favor earlier-deadline jobs over later-deadline ones.*

Note that Step 3.3 is not necessary for correctness; rather, it is an optimization.

We illustrate this table construction process by means of the following example.

▶ **Example 7.** Consider the instance consisting of the six jobs J_1–J_6 shown in tabular form in Figure 4, to be implemented upon a processor of minimum degraded speed $s = 1/2$.

In **Step 1**, we determine the intervals upon which the HI-criticality jobs J_1–J_3 would need to execute if they were to complete as late as possible, upon a degraded processor (one of speed-1/2); this is represented in the following diagram:



In **Step 2**, we construct an EDF schedule of thel HI-criticality jobs J_1–J_3 upon a speed-1/2 processor. Lettting $x_{i,j}$ denote the amount of execution accorded to job J_i in interval I_j, the scheduling table $S(I)$ looks like this:

I_j	$I_1 = [0, 1)$	$I_2 = [1, 5)$	$I_3 = [5, 6)$	$I_4 = [6, 8)$	$I_5 = [8, 10)$	$I_6 = [10, 13)$	$I_7 = [13, 15)$
J_1	0	0.5	0	0.5	1	0	0
J_2	0	0	0.5	0.5	0	0	0
J_3	0	0	0	0	0	1	1

In **Step 3**, we now try to fill in this scheduling table with LO-criticality jobs, interval by interval.

- Interval I_1 will be filled with the job J_4.
- Both J_4 and J_5 are in Interval I_2; J_4 has the earlier deadline. As a result, J_4 receives 3 time units and J_5 takes the remaining 0.5 unit. Here we check that J_4 has received enough execution and meets its deadline.

- Interval I_3 has 0.5 units of execution remaining for job J_5.
- The remaining one time unit capacity in I_4 will be used by J_5. Until now the scheduling table for HI-criticality jobs has remained unchanged from the one constructed in Step 2 (and shown in the above table).
- For the Interval I_5, there is no active LO-criticality job, and the pre-allocated HI-criticality amount $x_{1,5} = 1$ can not fill this up. In this case, we try to move later-assigned HI-criticality amounts into this interval. Specifically, we consider the next interval I_6, where x_{36} should be "promoted" as x_{35}; i. e., the one time unit that originally belongs to Interval $[10, 13)$ will be executed now. Note that after this step, the scheduling table for HI-criticality jobs is changed into the following one (with bold numbers highlighting changes).
- Interval I_6 is now empty and can be fully assigned to job J_6. Here we check that J_6 has received enough execution and meets its deadline.
- Nothing happens to Interval $[13, 15)$.

At the end of Step 3, the scheduling table for all jobs looks like this:

I_j	$[0, 1)$	$[1, 5)$	$[5, 6)$	$[6, 8)$	$[8, 10)$	$[10, 13)$	$[13, 15)$
J_1	0	0.5	0	0.5	1	0	0
J_2	0	0	0.5	0.5	0	0	0
J_3	0	0	0	0	***1***	***0***	1
J_4	1	3	0	0	0	0	0
J_5	0	0.5	0.5	1	0	0	0
J_6	0	0	0	0	0	3	0

◀

Computational Complexity. Although an individual job in an EDF schedule for an instance of n jobs may be preempted as many as $(n - 1)$ times, it is known (see, e. g., [6]) that the *total* number of preemptions in any EDF schedule for an n-job instance cannot exceed $(n - 1)$. In each column of the scheduling table, there should be at least one non-zero element unless all released jobs are finished beforehand. Each more non-zero element denotes that either a job is preempted, or a job finishes its execution within the corresponding interval. Since the number total finishing points is fixed as $n_{\text{HI}} + n_{\text{LO}}$, the total preemption number cannot exceed $(n_{\text{HI}} + n_{\text{LO}} - 1)$, and number of total intervals is no greater than $(2n_{\text{HI}} + 2n_{\text{LO}})$, we know that the total number of non-zero entries in the table of Step 3 cannot exceed $(4n_{\text{HI}} + 4n_{\text{LO}} - 1)$, where n_{HI} (n_{LO}, respectively) denotes the number of HI-criticality (LO-criticality, resp.) jobs in the instance.

We note that standard techniques (see, e. g., [14]) for implementing EDF are known, that allow an EDF schedule for n jobs to be constructed in $\mathcal{O}(n \log n)$ time. Consequently, we conclude that the EDF-schedule of Step 2 can be constructed in $\mathcal{O}(n_{\text{HI}} \log n_{\text{HI}})$ time, and the total scheduler overhead during run-time is also bounded from above by $\mathcal{O}(n \log n)$ where $n = n_{\text{HI}} + n_{\text{LO}}$ denotes the total number of jobs.

5 Non-preemptive Scheduling

Recall that the scheduling strategy we adopted in Section 3 above is as follows. Given an instance I, we construct a scheduling table $S(I)$. 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 degraded 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 hence is not applicable for

non-preemptive systems. In this section, we 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.

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 degraded mode) is already NP-hard in the strong sense [11][4]. 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 an instance of (regular, not MC) jobs that all share a common release time can be non-preemptively scheduled on a fixed-speed processor is easily solved in polynomial time by EDF, it turns out that even this restricted problem is NP-hard for MC scheduling.

▶ **Theorem 8.** *It is NP-hard to determine whether there is a correct scheduling strategy for scheduling non-preemptive mixed-criticality instances in which all jobs share a common release date.*

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 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 mixed-criticality scheduling problem I comprised of $(n+1)$ jobs $J_1, J_2, \ldots, J_{n+1}$. The parameters of the jobs are

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

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.

[4] Indeed, it seems that it is difficult to even obtain *approximate* solutions to this problem, to our knowledge, the best polynomial-time algorithm known [1] requires a processor speedup by a factor of 12.

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 degraded 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 degraded 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 degraded 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 degraded 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 degraded 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

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

  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 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 8 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. We are currently working on devising, and evaluating, polynomial-time approximation algorithms for the non-preemptive scheduling of mixed-criticality systems.

6 Recurrent Tasks

In Sections 3-5 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 [12, 13]. 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 *dual-criticality* systems of *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). 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 9) a necessary condition for the existence of a correct scheduling strategy. We then present a scheduling strategy, *Algorithm preemptive-MC*, and prove (Theorem 10) that it is optimal.

▶ **Theorem 9.** *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 should 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-degradation – processor), assign a share U_i of the processor to each task τ_i during each instant that is active.[5]

[5] 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.

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 10.** *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 9.

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 9 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. ◀

If preemption is forbidden, then scheduling of MC sporadic task systems becomes a lot more challenging. As with the collections of independent jobs (Theorem 8), this problem, too, can be shown to be highly intractable.

7 Context and Conclusions

Advanced processors may need to be modeled as *varying-speed* ones: although they are likely to execute at unit speed (or faster) during run-time, we can only *guarantee* that they will execute at lower speeds – the greater the level of assurance at which such a guarantee is sought, the lower the speed that can be guaranteed. Upon such a processor, the scheduling objective is to ensure that all jobs complete in a timely manner if the processor executes at its normal speed, while simultaneously ensuring that more critical jobs complete in a timely manner even if the processor speed falls to below this normal value.

In this paper, we have presented a formal framework for the scheduling-based analysis of MC systems that execute upon CPUs which may be modeled as varying-speed processors. We have defined a very simple model for representing MC systems, and have derived, and proved the correctness of, an optimal algorithm for the preemptive uniprocessor scheduling of MC systems that can be represented using our model. For the special case where there are only two criticality levels (such MC systems have been called *dual-criticality* systems in the literature), we have provided a more efficient scheduling algorithm. We have also cataloged the computational complexity of the problem when preemption is forbidden, and have derived a scheduling strategy for scheduling recurrent mixed-criticality task systems when unbounded preemption is permitted.

Acknowledgements. The research reported here was supported in part by NSF grants CNS 1016954, CNS 1115284, CNS 1218693, and CNS 1409175; and ARO grant W911NF-09-1-0535.

References

1 Nikhil Bansal, Ho-Leung Chan, Rohit Khandekar, Kirk Pruhs, Clifford Stein, and Baruch Schieber. Non-preemptive min-sum scheduling with resource augmentation. In *48th Annual IEEE Symp. on Foundations of Computer Science (FOCS'07), October 20–23, 2007, Providence, RI, USA*, pages 614–624. IEEE Computer Society, 2007. doi:10.1109/FOCS.2007.46.

2 Sanjoy Baruah and Zhishan Guo. Mixed-criticality scheduling upon varying-speed processors. In

IEEE 34th Real-Time Systems Symp. (RTSS'13), Vancouver, BC, Canada, December 3–6, 2013, pages 68–77. IEEE, 2013. doi:10.1109/RTSS.2013.15.

3 Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Trans. Computers*, 61(8):1140–1152, 2012. doi:10.1109/TC.2011.142.

4 Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne van der Ster, and Leen Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *24th Euromicro Conf. on Real-Time Systems (ECRTS'12), Pisa, Italy, July 11–13, 2012*, pages 145–154. IEEE Computer Society, 2012. doi:10.1109/ECRTS.2012.42.

5 Alan Burns and Robert Davis. Mixed-criticality systems: A review. Unpublished manuscript, 4th edition, July 31, 2014. URL: http://www-users.cs.york.ac.uk/~burns/review.pdf.

6 Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer US, 2nd edition, 2005. doi:10.1007/978-1-4614-0676-1.

7 M.R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

8 Zhishan Guo and Sanjoy Baruah. Mixed-criticality scheduling upon non-monitored varying-speed processors. In *8th IEEE Int'l Symp. on Industrial Embedded Systems (SIES'13), Porto, Portugal, June 19–21, 2013*, pages 161–167. IEEE, 2013. doi:10.1109/SIES.2013.6601488.

9 Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–396, 1984. doi:10.1007/BF02579150.

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

11 Jan Karel Lenstra, Alexander H. G. Rinnooy Kan, and Peter Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977. doi:10.1016/S0167-5060(08)70743-X.

12 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

13 Aloysius 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. URL: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-297.pdf.

14 Aloysius Mok. Task management techniques for enforcing ED scheduling on a periodic task set. In *5th IEEE Workshop on Real-Time Software and Operating Systems*, pages 42–46, Washington D. C., May 1988.

15 Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE Real-Time Systems Symp. (RTSS'07), December 3–6, 2007, Tucson, Arizona, USA*, pages 239–243. IEEE Computer Society, 2007. doi:10.1109/RTSS.2007.35.

16 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008. doi:10.1145/1347375.1347389.