8th IEEE International Symposium on Industrial Embedded Systems (SIES 2013)

Mixed-criticality scheduling upon non-monitored varying-speed processors

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

***Abstract*—A *varying-speed* 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 slowly (but no slower than at its degraded speed). A processor that is *self-monitoring* immediately knows if its speed falls below its normal speed during run-time; by contrast, a *non-monitored* processor cannot detect such degradation in performance during run-time. The problem of executing an integrated workload, consisting of some more important components and some less important ones, upon a non-monitored varying-speed 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.**

Keywords*—*Mixed-criticality scheduling; varying-speed processors; non-monitored processors; speedup bounds.

I. Introduction

Conditions during run-time, such as changes to the ambient temperature, the supply voltage, etc., may result in variations in the clock speed of a processor. While at the hardware level, innovations in computer architecture for increasing clock frequency can lead to varying-speed clocks during run time. Existing worst-case execution time (WCET) tools must make the most pessimistic assumptions regarding clock speed, which may result in a a significant under-utilization of the CPU's computing capacity since the lowest possible clock speed is highly unlikely to be reached in practice.

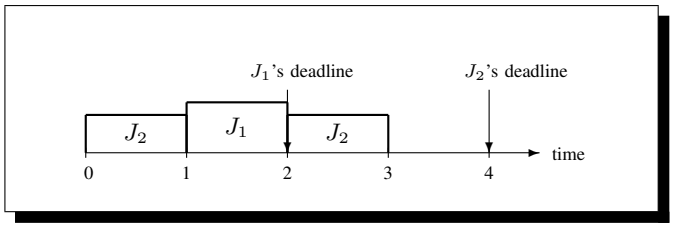
In this paper we consider a mixed-criticality (MC) real-time workload comprised of multiple independent jobs under such varying-speed processor. Each job is characterized by a release date, a worst-case execution time (WCET), and a deadline; each job is further designated as being either HI-criticality (more important) or LO-criticality (less important). It is desired to execute this workload upon a single shared preemptive processor. Let the function $s : \mathbb{R} \rightarrow \mathbb{R}$ denote the *speed* or *computing capacity* of this processor as a function of time, in the sense that the amount of execution completed by executing a job over the time-interval $[a, b)$ is equal to $\int_a^b s(t)\,dt$. So long as $s(t) \geq s_n$ for some (known) constant s_n, the processor is said to be operating in *normal* mode. If $s(t)$ falls below s_n but remains above s_d for another known constant $s_d < s_n$, the processor is said to be operating in *degraded* mode. Note that here $s(t)$ can be any real value at any time, we are only defining these two thresholds for separating functionality of the processor. If $s(t)$ falls below s_d, the processor is said to be *non-functional*. Moreover, the function $s(t)$ is not known beforehand: it is revealed during run-time as the processor executes. We seek a scheduling strategy that guarantees to complete all jobs by their deadlines if the processor remains in normal mode (i.e., $s(t) \geq s_n$ for all $t \geq 0$), while simultaneously guaranteeing to complete all HI-criticality jobs so long as the processor does not become non-functional (i.e., $s(t) \geq s_d$ for all $t \geq 0$).

978-1-4799-0658-1/13/$31.00 © 2013 IEEE

We have recently [4] considered the scheduling of such mixed-criticality workloads under the assumption that the platform upon which the workload is being executed is *self-monitoring* during run-time, in the sense that it immediately knows if it transits from normal to degraded mode (i.e., if its speed falls from $\geq s_n$ to below s_n).

In this paper, we remove this assumption, and consider platforms that lack the ability to self-monitor.

A natural question arises: does the lack of such an ability even matter? We construct a simple example mixed-criticality instance below that shows that it does. This example instance consists of one LO-criticality job J_1 and one HI-criticality job J_2, that are to be preemptively scheduled on a processor with normal speed $s_n = 1$ and degraded speed $s_d = \frac{1}{2}$. Both jobs arrive at time-instant zero; J_1's WCET is one and its deadline is at time-instant two, while J_2's WCET is two and its deadline is at time-instant four. Upon a self-monitoring processor, we could start out scheduling the system according to the following scheduling table:



If at any instant during this execution the processor determines that its execution speed has degraded below s_n, 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 such degradation occurs, 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 ≥ 2).

Suppose, however, that the processor *cannot* self-monitor: it does not know what its speed is at each instant during run-time. The schedule above is no longer acceptable: it is possible that the processor had degraded at the very beginning and was already operating at a reduced speed of $1/2$ throughout the interval $[0, 4)$, in which case neither job J_1 nor job J_2 would complete on time. This remains true even if J_2 were allocated execution over $[3, 4)$ upon it being discovered that it had not completed execution at time-instant 3. Indeed, there will not be any scheduling strategy for this example instance that meets all our requirements upon a non-monitoring processor, since the only scheduling strategy that can ensure that J_2 completes on a degraded processor would first execute J_2 to completion, but such a schedule would necessarily miss J_1's deadline even when the processor does not degrade.

Generally speaking, a self-monitoring processor knows its degradation as soon as it occurs, and can make the best choice such as drop LO-criticality jobs to save enough capacity for HI-criticality jobs. However if the processor cannot self-monitor, it won't realize such degradation until a job received enough execution time and hasn't got finished - LO-criticality jobs will continue to receive execution even when the processor is running at a degraded speed.

Contributions. Our contribution in this paper is twofold. First in Section III, we present strategies for scheduling MC workloads on non-monitored processors. Second, the example above illustrates that there are indeed systems that can be scheduled correctly by a self-monitoring processor but not by an unmonitored one. However, the ability to self-monitor comes at a price: processors with this capability are likely to be more complex (particularly since the self-monitoring facility needs to be accurate in order to be useful). In Section IV we seek to quantify the benefit of such self-monitoring.

II. Model

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 $\mathcal{J}$ of MC jobs: $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$, and
2) an unreliable processor that is characterized by both a normal speed s_n and a degraded speed $s_d < s_n$.

The interpretation is that the jobs in $\mathcal{J}$ 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 speed-s_n (or faster) processor and hence completes s_n (or more) units of execution per unit time, whereas in degraded mode it completes less than s_n, but at least s_d, units of execution per unit time.

For each $t \in \mathbb{R}_{\geq 0}$, let $s(t)$ denote the speed of the processor at time-instant t. The processor starts out executing at or above its normal speed: $s(0) \geq s_n$. It is not *a priori* known when, if at all, the processor will degrade. We seek to determine a ***correct scheduling strategy***, which is defined as follows:

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

- if $s(t) \geq s_n$ for all $t \in [\min_i\{a_i\}, \max_i\{d_i\})$, then all jobs complete by their deadlines; and
- if $s(t) \geq s_d$ for all $t \in [\min_i\{a_i\}, \max_i\{d_i\})$, 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.

A. Related work

A lot of research has recently been done on various aspects of mixed-criticality scheduling. Although most of prior work draws inspiration from the seminal work of Vestal [8], 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. Apart from our own recent efforts (reported in [4]), 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 assigns 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.

III. A scheduling algorithm

The high-level description of our algorithm is as follows. Given an MC instance $I = (\mathcal{J}, s_n, s_d)$, we aim to derive offline (i.e., prior to run-time) a total priority ordering of the jobs of $\mathcal{J}$ such that scheduling the jobs according to this priority ordering constitutes a correct scheduling strategy, where *scheduling according to priority* means that at each moment in time the highest-priority available job is executed.

The priority list is constructed recursively using the approach commonly referred to in the scheduling literature as "Lawler's algorithm" [7] or the "Audsley approach" [1], [2]. We first determine (as described below) some job that may be assigned lowest priority, and assign it the lowest priority. Then the procedure is repeated upon the set of jobs excluding the lowest priority job, until all jobs are ordered, or at some iteration a lowest priority job cannot be found.

J_i	a_i	c_i	d_i	χ_i
J_1	0	2	5	LO
J_2	0	3	10	HI
J_3	3	1	5	HI
J_4	2	4	10	LO

Fig. 1. Mixed-criticality instance considered in Example 1.

Determining a lowest-priority job. It can be shown, using techniques very similar to those used in, e.g. [5], that if any LO-criticality job may be assigned lowest priority then so may the LO-criticality job with the latest deadline, and that if any HI-criticality job may be assigned lowest priority then so may the HI-criticality job with the latest deadline. Hence we only need to determine whether one of the two jobs, the latest-deadline LO-criticality job or the latest-deadline HI-criticality job, may be assigned lowest priority.

- We assign lowest priority to the latest-deadline LO-criticality job if it would complete by its deadline on a speed-s_n processor if every other job were assigned higher priority.
- Else, we assign lowest priority to the latest-deadline HI-criticality job if it would complete by its deadline *on a speed-s_d processor* if every other job were assigned higher priority. Here no LO-criticality job is allowed to execute more than the duration of its WCET on a speed-s_n processor [1].
- Else, we declare failure

(Note that at this point in time we do not check to determine whether the jobs assigned higher priority would meet their own deadlines or not – we are simply assuming that they each execute to completion in a work-conserving manner.)

We illustrate the priority-assignment process by means of a simple example.

Example 1: Consider the instance consisting of the four jobs J_1–J_4 shown in tabular form in Figure 1, to be implemented upon a processor of normal speed $s_n = 1$ and a degraded speed $s_d = 0.75$.

- It may be verified that J_4 would meet its deadline on a unit-speed processor if it were assigned lowest priority. We therefore assign J_4 the lowest priority.
- Next, we must determine which of the remaining three jobs may be assigned lowest priority amongst them.
  - If J_1 were assigned lower priority than both J_2 and J_3, then upon a unit-speed processor J_2 would execute over $[0, 2)$, and J_2 and J_3 together would execute over $[2, 4)$. That leaves J_1 just one unit of execution by its deadline, which is not enough to allow it to meet its deadline.
  - If J_2 were assigned lower priority than both J_1 and J_3, then on a speed-0.75 processor J_3 would execute for $1/0.75$ time units, and J_1 would still execute for 2 time units as a LO-criticality job (since it should be dropped when not finished after receiving this much execution time). Thus J_1 and J_3 will execute 3.33 time units over the interval $[0, 3.33)$, which would allow J_2 to execute over the interval $[3.33, 7.33)$ and consequently receive the required units of execution ($3/0.75 = 4$). We therefore assign J_2 the second-lowest priority from amongst the four jobs.
- That leaves us with J_1 and J_3. Suppose J_1 is assigned lower priority than J_3. Then on a unit-speed processor J_1 would execute over $[0, 2)$, and complete by its deadline. It may therefore be assigned the third-lowest priority.
- The remaining job J_3 is therefore assigned lowest priority.

The final priority ordering is thus as follows (letting $J_i \rhd J_j$ denote that J_i has greater priority than J_j):

$$J_3 \rhd J_1 \rhd J_2 \rhd J_4$$

■

It is evident that this algorithm for assigning priorities is very efficient – it has a run-time that is a low-order polynomial in the number of jobs – and it is guaranteed to find a total priority ordering of the jobs, if one exists, such that scheduling according to this priority ordering is a correct online scheduling strategy.

Lemma 1: Priority-based scheduling according to the priorities derived as described above constitutes a correct scheduling strategy.

Proof: Suppose for a contradiction that our priority-assignment procedure was successful in assigning priorities to all the jobs in instance $I = (\mathcal{J}, s_n, s_d)$, but that job $J_i \in \mathcal{J}$ misses its deadline during run-time.

- Suppose first that J_i is a LO-criticality job ($\chi_i = \text{LO}$). It follows from the manner in which priorities were assigned and the *sustainability* [3] of preemptive fixed-priority scheduling with respect to processor speed, that J_i would have met its deadline despite the interference of jobs assigned greater priority, if the processor had executed throughout at a speed of s_n or greater. For the deadline miss to occur, hence, the processor must have executed at some speed strictly less than s_n at some instant prior to J_i's deadline. By the definition of correct scheduling strategy (Definition 1), J_i does not need to meet its deadline.

[1] The intuition behind this is that although the processor is not self-monitoring, we still know its degradation when a job has been executed for enough time and is still not finished; i.e., a LO-criticality job should be dropped if is not finished after executing enough time units – its WCET divided by s_n.

- Suppose now that J_i is a HI-criticality job ($\chi_i = \text{HI}$). It once again follows from the manner in which priorities are assigned, and the sustainability property, that J_i would have met its deadline despite the interference of jobs assigned greater priority, if the processor had executed throughout at a speed of s_d or greater. For the deadline miss to occur, the processor must therefore have executed at some speed strictly less than s_d at some instant prior to J_i's deadline. By the definition of correct scheduling strategy (Definition 1), J_i does not need to meet its deadline.

We thus see that deadlines are missed only when doing so does not violate the requirements of correct scheduling. ■

A. Optimization versions of the problem

Given an MC instance $I = (\mathcal{J}, s_n, s_d)$, we derived above an algorithm for determining a correct scheduling strategy for instance I. Two *optimization* versions of the MC scheduling problem can also be defined:

1) Given a collection of MC jobs $\mathcal{J}$ and a normal processor speed s_n, what is the smallest degraded processor speed s_d such that we can determine a correct scheduling strategy for the MC instance $I = (\mathcal{J}, s_n, s_d)$?
2) Given a collection of MC jobs $\mathcal{J}$ and a degraded processor speed s_d, what is the smallest normal processor speed s_n such that we can determine a correct scheduling strategy for the MC instance $I = (\mathcal{J}, s_n, s_d)$?

It is evident that both these optimization problems can be approximately solved to any desired degree of accuracy by applying the technique of "binary search" in conjunction with the algorithm for determining a correct scheduling strategy for a given instance (in which all three parameters – $\mathcal{J}$, s_n, and s_d – are specified). Consider, for example, the first optimization problem listed above, in which $\mathcal{J}$ and s_n are specified and the objective is to determine the smallest s_d. An upper bound on the value of s_d is s_n; a lower bound is zero. We could therefore repeatedly guess a value for s_d within this interval, seeking the smallest value for which we are able to construct a correct scheduling strategy for $I = (\mathcal{J}, s_n, s_d)$.

However, it turns out that we can in fact solve the problem directly, without needing to do binary search. For the first optimization problem listed above, the pseudo-code for doing so is given in Figure 2. We start out "guessing" that the value of s_d is zero (line 2 of the pseudo-code), and repeatedly seeking to determine whether some job can be assigned lowest priority for this value of s_d. If so, we continue; if not, we increase the guessed value of s_d to the smallest value needed to be able to assign some job the lowest priority and then continue. (If follows from the sustainability property of fixed-priority scheduling with respect to processor speed that if lower-priority jobs met their deadlines with the smaller values of s_d, they will continue to do so when s_d's value is increased.)

An analogous strategy can be obtained for the second optimization problem, with the value of s_d fixed and a value guessed for s_n. We omit the details.

```
OPTI-1(J, s_n)
 1  J' ← J
 2  s_d ← 0
 3  repeat
 4     Let J_L be the latest-deadline LO-criticality job in J'
 5     Let J_H be the latest-deadline HI-criticality job in J'
 6     if J_L meets its deadline as the lowest-priority job on
    a speed-s_n processor
 7      then J_L gets lowest priority
 8           J' ← J' \ {J_L}
 9      else
          Determine s', the smallest speed such that J_H meets
          its deadline as the lowest-priority job on a speed-
          s' processor, where LO-criticality job in J' receives
          execution time of at most its WCET divided by s_n
10        s_d ← max{s_d, s'}
11        J_H gets lowest priority
12        J' ← J' \ {J_H}
13        s_d ← max{s_d, s'}
14     until J' is empty
```

Fig. 2. Determining the smallest degraded processor speed.

IV. Quantifying the benefits of self-monitoring

In Section I, we had identified two sets of contributions of this paper. The first set concerned the design of algorithms for obtaining correct scheduling strategies for MC systems: these were presented in Section III above. We now turn to the second set of contributions: a quantitative evaluation of the benefits of providing self-monitoring facilities to processors.

If a MC instance $I = (\mathcal{J}, s_n, s_d)$ can be scheduled by a correct scheduling strategy upon a self-monitoring processor, then it is evident that the jobs in $\mathcal{J}$ can be scheduled by a correct scheduling strategy upon an unmonitored processor in which the normal and the degraded speeds are *both* equal to s_n (equivalently, the processor does not have a non-trivial degraded mode). The following lemma shows that this is the best general result we can come up with:

Lemma 2: There are MC instances $I = (\mathcal{J}, s_n, s_d)$ that can be scheduled by a correct scheduling strategy upon a self-monitoring processor, but for which $(\mathcal{J}, s_n, s')$ cannot be scheduled by a correct scheduling strategy upon an unmonitored processor for all $s' < s_n$.

In other words, such instances can only be scheduled upon an unmonitored processor if the processor does not have a non-trivial degraded mode.

Proof: We prove this lemma by demonstrating the existence of such an instance I. Let $s_n \leftarrow 1$, and let s_d be any constant less than one. Let k denote some large positive constant. Consider the collection of MC jobs $\mathcal{J} = \{J_{1,}, J_2\}$ as listed below:

J_i	a_i	c_i	d_i	χ_i
J_1	0	$(k+1)$	$(k+1)/s_d$	HI
J_2	0	$k(1-s_d)/s_d$	k/s_d	LO

For instance if s_d were $1/2$ and k is chosen equal to 9, J_1 would have a WCET of 10 and a deadline at 20, while J_2

would have a WCET of 9 and a deadline at 18.

Upon a self-monitoring processor we could construct a scheduling table that executes the HI-criticality job J_1 over $[0, k)$, J_2 over $[k, k/s_d)$, and J_1 again over $[k/s_d, k/s_d + 1)$. For the example parameters of $s_d = 1/2$ and $k = 9$, this would correspond to scheduling J_1 over $[0, 9)$ and $[18, 20)$, and J_2 over $[9, 18)$.

It is evident that a self-monitoring processor would complete both jobs on a processor that executes throughout in normal mode. If the speed of the processor falls to below s_n (which, for our example, is 1) at any instant, the LO-criticality job J_2 is immediately discarded and J_1 executed – it may be validated that this strategy results in J_1 always meeting its deadline as long as the processor speed remains at least s_d (for our example,$1/2$).

Upon a non-monitored processor with normal speed also equal to 1 and degraded speed s', we must execute J_2 for its WCET prior to its deadline (since we cannot determine, prior to J_2's deadline, whether the processor is in normal or degraded mode). Since J_1's deadline is after J_2's, the duration for which J_1 will execute is hence bounded by

$$\begin{aligned}(d_1 - a_1) - c_2 \\ = \quad & \frac{k+1}{s_d} - \frac{k(1-s_d)}{s_d} \\ = \quad & \frac{1+ks_d}{s_d}\end{aligned}$$

Suppose that the processor were to be degraded mode throughout; i.e., starting at time-instant zero. For J_1 to execute to completion by its deadline, we need that

$$\begin{aligned}& s' \times \left(\frac{1+ks_d}{s_d}\right) \geq c_1 \\ \Leftrightarrow \quad & s' \times \left(\frac{1+ks_d}{s_d}\right) \geq (1+k) \\ \Leftrightarrow \quad & s' \geq \frac{s_d + ks_d}{1+ks_d}\end{aligned}$$

from which it follows that s' approaches one as $k \to \infty$. The lemma is thus proved. ■

A. The speedup cost of not monitoring

As discussed in Section II-A, much previous work on MC scheduling has focused upon a model in which the processor speed is assumed to remain constant throughout run-time but each job is characterized by two different WCET values: a LO-criticality value and a larger HI-criticality value. An algorithm titled OCBP for *Own Criticality-Based Priorities* was proposed in [5], [6] for scheduling such MC systems, and the following *speedup bound* proved (as, e.g., [5, Lemma 5]): If an MC instance is schedulable on a given processor, then it is OCBP-schedulable on a processor that is $(1+\sqrt{5})/2$ (or approximately 1.618) times as fast.

Consider a single job with WCET of c upon unit speed processor ($s_n = 1$). Upon an unreliable processor where $s(t)$ varies from s_d to 1, it may need up to c/s_d units of time to finish execution. Under the assumptions of our model, a slower non-monitored processor can be modeled as a longer WCET, and thus we can re-formulate the MC model that is described in Section II above into the 2-WCET model assumed by the OCBP algorithm, in the following manner. Given a MC instance $I = (\mathcal{J}, s_n, s_d)$ in the model described in Section II, each job $J_i = (a_i, c_i, d_i, \chi_i)$ in $\mathcal{J}$ is modeled as a job J_i' with the same criticality, release date, and deadline, and with LO-criticality WCET equal to c_i/s_n and HI-criticality WCET equal to c_i/s_d. Hence for instance if s_n were equal to one and s_d one-half, J_i''s LO-criticality WCET would equal c_i and its HI-criticality WCET would be equal to $2c_i$.

Upon such translation, the algorithm that we described in Section III behaves in essentially the same manner as OCBP, and as a consequence similar speedup bounds can be derived: If an instance can be scheduled on a self-monitoring processor, then it can be scheduled on a non-monitoring processor that is $(1+\sqrt{5})/2$ times as fast in both the normal and the degraded mode.

Furthermore, in our speed-varying processor model, an lower bound s_d for the processor speed is given a priori; which, under the translation, provides an additional upper bound to the ratio between two WCET's of a HI-criticality job in previous 2-WCET model. As a result, we are able to prove a tighter speed up factor bound – that is related to the given s_d – in the following theorem.

Theorem 1: Let $I = (\mathcal{J}, 1, s)$ denote a MC instance that can be correctly scheduled by an optimal scheduling strategy upon a self-monitoring processor. If the instance $I' = (\mathcal{J}, \phi, \phi \times s)$ is not correctly scheduled by the algorithm described in Section III, then $\phi < \min\{2 - s, \sqrt{s} + 1\}$.

Proof: For a given s, let $I = (\mathcal{J}, 1, s)$ denote some minimal instance that can be scheduled correctly by an optimal algorithm on a self-monitoring processor, but $I' = (\mathcal{J}, \phi, \phi\, s)$ is not correctly scheduled on a non-monitoring processor using the algorithm of Section III.

Let d_{LO} denote the latest deadline of any LO-criticality job, and d_{HI} the latest deadline of any HI-criticality job; let c_{LO} and c_{HI} denote the cumulative WCET's of the LO- and HI-criticality jobs respectively:

$$\begin{aligned}d_{\text{LO}} &= \max_{j|\chi_j=\text{LO}} d_j, \\ d_{\text{HI}} &= \max_{j|\chi_j=\text{HI}} d_j, \\ c_{\text{LO}} &= \sum_{j|\chi_j=\text{LO}} c_j, \\ c_{\text{HI}} &= \sum_{j|\chi_j=\text{HI}} c_j.\end{aligned}$$

Consider now any work-conserving schedule of $\mathcal{J}$ upon a speed-ϕ processor, when each job J_i requests exactly c_i units of execution[2]. Let $\Lambda_1, \Lambda_2, \ldots$ denote the intervals, of

[2]We are not attempting to meet deadlines in this schedule, simply keeping the processor active whenever there are jobs remaining that have arrived but not completed execution, regardless of whether their deadlines are met or not.

cumulative length λ, during which the processor is idle in this schedule.

Observation 1: No LO-criticality job has a scheduling window that overlaps with Λ_ℓ, for any ℓ.

Proof: Suppose that some LO-criticality job J_i were to overlap withΛ_ℓ for some ℓ. This means that all the jobs which arrive prior to Λ_ℓ complete by the beginning of Λ_ℓ. Hence, J_i would complete by its deadline upon a speed-ϕ processor, if it were assigned lowest priority. But this contradicts the assumption that $I' = (\mathcal{J}, \phi, \phi\, s)$ is a *minimal* instance that is not correctly scheduled on a non-monitoring processor using the algorithm of Section III. ■

Since $I = (\mathcal{J}, 1, s)$ is assumed to be schedulable on a self-monitoring processor, all LO-criticality jobs would complete by d_{LO}, the latest deadline of any LO-criticality job, on a speed-1 processor. It therefore follows from Observation 1 that the cumulative WCET's of all LO-criticality jobs cannot exceed $(d_{\text{LO}} - \lambda)$:

$$c_{\text{LO}} \quad \leq \quad d_{\text{LO}} - \lambda \tag{1}$$

Since we are assuming that the instance $I' = (\mathcal{J}, \phi, \phi\, s)$ is not correctly scheduled by the algorithm described in Section III, it must be the case that the LO-criticality job with the latest deadline cannot be the lowest-priority job on a speed-ϕ processor. Hence, it is necessary that

$$c_{\text{LO}} + c_{\text{HI}} > (d_{\text{LO}} - \lambda)\, \phi \tag{2}$$

We now argue from the schedulability of $I = (\mathcal{J}, 1, s)$ on a self-monitoring processor that

- All the jobs would complete by d_{HI}, the latest deadline of any job, upon a speed-1 processor. Inequality 3 below, immediately follows.

$$c_{\text{LO}} + c_{\text{HI}} \quad \leq \quad d_{\text{HI}} \tag{3}$$

- All HI-criticality jobs would complete by d_{HI} upon a speed-s processor. Inequality 4 follows:

$$\frac{c_{\text{HI}}}{s} \quad \leq \quad d_{\text{HI}} \tag{4}$$

Observation 2: Consider now any work-conserving schedule of $\mathcal{J}$ upon a speed-ϕ processor, when each LO-criticality job J_i executes for exactly c_i time-units, and each HI-criticality job J_i executes for exactly (c_i/s) time-units[3]. There are no idle intervals in this schedule.

Proof: If there were an idle interval, any job whose scheduling window spans the idle interval would meet its deadline upon the speed-ϕ processor if it were assigned lowest priority. But this contradicts the assumption that $I' = (\mathcal{J}, \phi, \phi\, s)$ is a minimal instance that is not correctly scheduled on a non-monitoring processor using the algorithm of Section III. ■

Since we are assuming that $I' = (\mathcal{J}, \phi, \phi\, s)$ is not correctly scheduled on a non-monitoring processor using the algorithm of Section III it must be the case that the latest-deadline HI-criticality job will not meet its deadline if it were assigned the lowest-priority. Given Observation 2 above, it must then be the case that

$$c_{\text{LO}} + \frac{c_{\text{HI}}}{s} > d_{\text{HI}}\ \phi \tag{5}$$

Suppose that the value of s is known, by multiplying both sides of Inequality (1) by a factor ϕ and combining with Inequality (2), we have

$$c_{\text{LO}} + c_{\text{HI}} > c_{\text{LO}}\phi. \tag{6}$$

By chaining Inequalities (5) and (3), we get

$$c_{\text{LO}} + \frac{c_{\text{HI}}}{s} > (c_{\text{LO}} + c_{\text{HI}})\, \phi \tag{7}$$

while by chaining Inequalities (5) and (4), we get

$$c_{\text{LO}} + \frac{c_{\text{HI}}}{s} > \frac{c_{\text{HI}}}{s}\, \phi \tag{8}$$

Let y denote the ratio of cumulative WCET length of different criticality jobs; i.e., $y := c_{\text{HI}}/c_{\text{LO}}$. From Inequalities (6)-(8), we conclude that

$$\phi < 1 + \min\{y, (1-s)/(y+s), s/y\}. \tag{9}$$

It is evident that $(1-s)/(y+s)$ and s/y decreases, with increasing $y \in \mathbb{R}^+$ and any fixed $s \in (0,1)$. Let $(1-s)/(y+s) = s/y$, and we have $y = s^2/(1-2s)$ which helps break Inequality (9) above into the following two inequalities:

$$\phi < 1 + \min\{y, (1-s)/(y+s)\}, \qquad \text{if } x < \frac{s^2}{1-2s} \tag{10}$$

$$\phi < 1 + \min\{y, \phi/s\}, \qquad \text{otherwise} \tag{11}$$

By solving the two equations $y = (1-s)/(y+s)$ and $y = s/y$, noticing that $y > 0$, we get solutions $y_1^* = (1-s)$ and $y_2^* = \sqrt{s}$. As a result, by substituting the min functions over y in Inequalities (10) and (11) and combining them together, we obtain the following relationship between ϕ and s:

$$\phi < 1 + \min\{1-s, \sqrt{s}\} = \min\{2-s, \sqrt{s}+1\}. \tag{12}$$

■

Figure 3 shows the bound on the speedup factor ϕ as a function of the ratio u of degraded processor speed to normal processor speed: $u := (s_d/s_n)$.

By solving the equation $2 - s = \sqrt{s} + 1$, we get $s^* = (3-\sqrt{5})/2$ and $\phi \leftarrow (2-s) = (1+\sqrt{5})/2$. Figure 3 shows the 1.618 speed-up factor upper bound, which matches the results in previous works.

Corollary 1: Let $I = (\mathcal{J}, 1, s)$ denote a MC instance that can be correctly scheduled by an optimal scheduling strategy upon a self-monitoring processor. If the instance $I' = (\mathcal{J}, \phi, \phi \times s)$ is not correctly scheduled by the algorithm described in Section III, then $\phi < (1+\sqrt{5})/2$.

From Figure 3, we can see that compared to results in prior work [5], we can achieve a lower speed-up factor when

[3]As in Observation 1, we are not attempting to meet deadlines in this schedule.

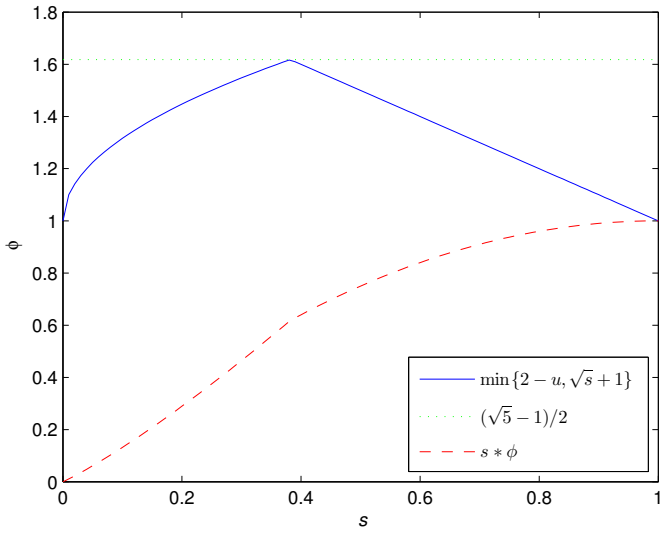


Fig. 3. Lower bound on the speedup factor ϕ as a function of s - the ratio of the degraded to the normal processor speed.

the given s varies according to Theorem 1 for the non-self-monitoring case. The slashed line shows how $s \times \phi$ is related to s. $s \times \phi$ gives the actual minimum degraded speed that our algorithm need, and can be recognized as a measurement of CPU resource usage, which is the lower the better. The case $s \times \phi = 1$ means that we need exactly a processor that runs $1/s$ times faster for a degraded speed s, which is equivalent to having a processor running at minimum speed 1 in degraded mode; and as $s \times \phi < 1$, we are doing better – giving the speed-up tradeoff, the processor no longer needs to run at unit speed in degraded speed while all jobs are still guaranteed to be finished on time regarding their criticality levels.

How *tight* is this relationship between speedup and s? To answer this question, consider the MC instance $I = (\mathcal{J}, 1, s)$; let σ denote $1/(1-s)$, and let $\mathcal{J}$ consist of the following two jobs:

J_i	a_i	c_i	d_i	χ_i
J_1	0	1	1	LO
J_2	0	σ	$(\sigma - 1)$	HI

It has been shown [6, Proposition 2] that by taking $\sigma = (1 + \sqrt{5})/2$, this instance reaches its schedulablity bound. Noticing that for such σ, $s = (\sigma - 1)/\sigma = (3 - \sqrt{5})/2$ takes exactly the value of s^* that is calculated above. This implies that Inequality (12) provides a tight bound for the speedup factor ϕ, and the upper bound of ϕ can be calculated by $\max(\phi) = 2 - s^* = 1 + \sqrt{s^*} = (1 + \sqrt{5})/2$. We can also tell from Figure 3 that for a given $\phi \in (0, 1)$, the upper bound of speedup factor varies from 1 when the ratio of normal to degraded speed is either zero or one, to $(1 + \sqrt{5})/2$ when this ratio is equal to $(3 - \sqrt{5})/2$ (or ≈ 0.382).

V. Conclusions

We have recently [4] begun studying the scheduling of 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. This prior work has assumed that a platform is self-monitoring: it "knows" its execution speed at each instant during run-time.

In this paper, we reported the results of our investigations into the scheduling of mixed-criticality systems upon processors that do not have such self-awareness. Upon such non-monitored processors, we have designed scheduling algorithms that guarantee to complete all jobs by their deadlines under normal circumstances and guarantee that all HI-criticality jobs will meet their deadlines even if the processor degrades, *without* knowing during run-time whether the processor is normal or degraded; in addition, we have provided quantitative evaluations of the efficacy of these algorithms.

We have assumed here that a platform is not aware of its execution speed during run-time. It would be interesting and important to derive algorithms for scheduling mixed-criticality systems upon platforms that have the ability of self- awareness, where many works can be done besides [4]. Another further direction would be integrating the model we are proposing here with other mixed-criticality models [8], where multiple WCET's are specified in addition to varying-speed processors.

Acknowledgements

This research has been supported in part by NSF grants CNS 0834270, CNS 0834132, and CNS 1016954; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

References

[1] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical report, The University of York, England, 1991.

[2] N. C. Audsley. *Flexible Scheduling in Hard-Real-Time Systems*. PhD thesis, Department of Computer Science, University of York, 1993.

[3] Sanjoy Baruah and Alan Burns. Sustainable scheduling analysis. In *Proceedings of the IEEE Real-time Systems Symposium*, pages 159–168, Rio de Janeiro, December 2006. IEEE Computer Society Press.

[4] Sanjoy Baruah and Zhishan Guo. Mixed-criticality scheduling upon unreliable processors. Under review; available at http://www.cs.unc.edu/~baruah/Pubs.shtml, 2013.

[5] Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*. IEEE, April 2010.

[6] 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 Transactions on Computers*, 61(8):1140–1152, 2012.

[7] E. L. Lawler. Optimal sequencing of a single machine subject to precedence constraints. *Management Science*, 19(5):544–546, 1973.

[8] Steve 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.