2013 IEEE 34th Real-Time Systems Symposium

Mixed-criticality scheduling upon varying-speed processors

Sanjoy Baruah Zhishan Guo
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; conditions during run-time may cause it to execute more slowly (but no slower than at its degraded speed).**

The problem of executing an integrated workload, consisting of some more important components and some less important ones, upon such a 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.

I. Introduction

Many safety-critical real-time systems have traditionally been very carefully designed and implemented, upon highly predictable and reliable special-purpose hardware. However, two trends have recently emerged in safety-critical real-time embedded systems: (i) the use of *commodity* (or commercial off-the-shelf - COTS) hardware for implementing such systems; and (ii) the move towards *mixed-criticality* implementations, in which functionalities of different degrees of importance are implemented upon a shared platform.

Varying-speed CPUs. Special-purpose processors used in implementing safety-critical systems are designed to be highly predictable: given the specifications of the workload that is to be executed upon such a processor, it is possible to provide tight bounds on the worst-case run-time behavior of the system during system design time itself, to a very high level of assurance. Such design-time predictability is essential for safety-critical functionalities, but is difficult to achieve with COTS processors that are typically engineered to provide good average-case performance rather than worst-case guarantees. In this paper, we focus upon one aspect of guaranteeing real-time performance upon such COTS processors: *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. 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 programmer may use the userspace Linux command `cpuspeed` to configure a system to reduce CPU clock speeds if the core temperature gets too high. At the hardware level, too, innovations in computer architecture for increasing clock frequency can lead to variable-speed clocks during run-time: e.g., [5] describes a recently-introduced technique for detecting whether signals are late at the circuit level within a CPU micro-architecture, and if so to recover by delaying the next clock tick so that logical faults do not propagate to higher (i.e., the software) levels.

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 [6] 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.

In this paper we take a somewhat different perspective on mixed-criticality scheduling: the system is analyzed only once, under a single set of assumptions. The mixed-criticality nature of the system arises in the fact that while we would like all functionalities to execute correctly under normal circumstances, it is essential that the more critical functionalities execute correctly even 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 worst-case execution time (WCET), and a deadline; each job is further designated as being HI-criticality (more important) or LO-criticality (less important). We desire to schedule the system upon a single processor. This processor is a varying-speed one, modeled 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), it may at any instant lapse into a "degraded" mode during which it can only complete fewer than one, but at least s, units of execution during each time unit, for some (known) constant $s < 1$. It is not *a priori* known when, or whether, such degradation will occur[1]. We seek a scheduling strategy that *guarantees to complete all jobs by their deadlines if the performance of the processor does not degrade during run-time, while simultaneously guaranteeing to complete all* HI-*criticality jobs if the processor does suffer a degradation in performance*.

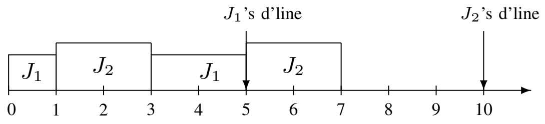
Example 1: Consider the following collection of two jobs, to be scheduled on a preemptive processor with normal speed 1 and degraded speed $s = \frac{1}{2}$:

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

An Earliest Deadline First (EDF) [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 at any instant during this execution then J_1 is immediately discarded and the processor executes J_2 exclusively.

It may be verified, by exhaustive consideration of all possible instants at which the processor may degrade, that this scheduling strategy will result in J_2 completing by its deadline regardless of when (if at all) the processor degrades 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 that it is imperative that real-time scheduling theory understand 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*. Commodity processors tend to execute at varying speeds as ambient conditions change; in order to make correctness guarantees at very high levels of assurance, 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 II we introduce a very simple model for MC systems, that allows for the representation of systems consisting of a finite number of independent jobs. In Sections III-IV we present, analyze, and evaluate algorithms for the preemptive scheduling of MC systems that can be represented using this model; in Section V, we consider the problem when preemption is forbidden. In Section VI, we consider a more general model for MC systems: one that allows for the modeling of systems comprised of recurrent tasks. We conclude in Section VII 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 1 and degraded speed $s(s < 1)$. This job may be represented in the multiple-WCET model as a job with a normal WCET of C and a more pessimistic WCET of (C/s); 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 (but no job executes beyond its more pessimistic WCET) then only the more critical jobs are required to execute correctly. It is not difficult to show that the algorithms proposed in

[1]We do however assume that the system is capable of self-monitoring: it immediately knows if and when such degradation occurs. I.e., it has access to some facility similar to the capabilities offered by the Linux `cpufreq-info` command.

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, we show in this paper that we can actually do better, since 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 [4] to be NP-hard in the strong sense, in Section III we present an optimal polynomial-time algorithm for solving the same problem in our model. For the case of implicit-deadline sporadic tasks on preemptive uni-processors, a speedup bound of $4/3$ had been established [2] for the multiple-WCETs model, whereas we again show an optimal (speedup-1) algorithm in Theorem 3 (Section VI) here.

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 faulty communication media. Specifically, they are particularly relevant to data-communication problems in which time-sensitive data and data-streams must be transmitted over potentially faulty communications media which can provide a high bandwidth under most circumstances but can only *guarantee* a lower bandwidth: the high bandwidth would correspond to the normal processor speed, and the lower bandwidth to the degraded speed. We therefore believe that this work is relevant to problems of factory communication, communication within automobiles or aircraft, wireless sensor networks, etc., in addition to processor scheduling of mixed-criticality workloads.

II. Model

We start out considering a workload model consisting of *independent jobs*; a model for representing *recurrent tasks* is considered in Section VI. 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 $\chi_i \in \{\text{LO}, \text{HI}\}$. 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 both a normal speed (without loss of generality, assumed to be equal to one) and a specified *degraded processor speed* $s < 1$.

The interpretation is that the jobs 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 unit-speed processor and hence completes one unit of execution per unit time, whereas in degraded mode it completes less than one, but at least s, units of execution per unit time. The processor starts out executing at its normal speed. It is not *a priori* known when, if at all, the processor will degrade: 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***:

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

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

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

In Section III below, we consider the problem of determining such correct scheduling strategies. In Section IV, we consider an optimization version of this problem: given the collection of MC jobs $\mathcal{J}$, what is the *smallest* s such that there is a correct scheduling strategy for the instance $(\mathcal{J}, \mathrm{s})$?

III. Preemptive scheduling

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

In the remainder of this section we present, and prove the correctness of, a simple polynomial-time algorithm for constructing these scheduling tables $S(I)$ optimally. By *optimal*, we mean that if there is a correct scheduling strategy (Definition 1 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 conditions for MC-schedulability:

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

Given any instance I, it can be efficiently determined whether I satisfies the necessary conditions of Lemma 1: simply simulate the EDF scheduling of all the jobs in I upon a unit-speed processor, and of the HI-criticality jobs in I upon a speed-s processor. In the remainder of this section, let us therefore assume that any instance under consideration satisfies these necessary conditions. (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 $I = (\{J_1, J_2, \ldots, J_n\}, s)$ that satisfies the conditions of Lemma 1, 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 $S(I)$. Without loss of generality, assume that the HI-criticality jobs in I are indexed $1, 2, \ldots, n_h$ and the LO-criticality jobs are indexed $n_{h+1}, \ldots, n$. Let $t_1, t_2, \ldots, t_{k+1}$ denote the at most $2n$ distinct values for the release date and deadline parameters of the n jobs, in increasing order ($t_j < t_{j+1}$ for all j). These release dates and deadlines partition the time-interval $\big[\min_i\{a_i\}, \max_i\{d_i\}\big)$ into k intervals, which we will denote as $I_1, I_2, \ldots, I_k$, with I_j denoting the interval $[t_j, t_{j+1})$.

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

$$\Big(\sum_{j \mid t_j \ge a_i \ \wedge \ d_i \ge t_{j+1}} x_{i,j}\Big) \ge c_i, \text{ for each } i, 1 \le i \le n \quad (1)$$

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

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

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

It should be evident that any scheduling table generated in this manner from $x_{i,j}$ values satisfying the above $(n+k)$ constraints will execute all jobs to completion upon a normal (non-degraded) processor. It now remains to write constraints for specifying the requirements that the HI-criticality jobs complete execution even in the event of the processor degrading into faulty mode. We observe that the worst-case scenarios occur when the processor transits to degraded mode at the very *beginning* of a contiguous block of HI-criticality execution in the scheduling table, since that would leave the maximum amount of HI-criticality execution remaining to be done on the degraded processor. For each ℓ, $1 \le \ell \le k$, we represent the possibility that this transition occurs at the start of the interval I_ℓ in the following manner:

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

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

(iii) This is ensured by writing a constraint

$$\Big(\sum_{i:(\chi_i=\text{HI})\wedge(d_i \le t_m)} \big(\sum_{j=\ell}^{m-1} x_{i,j}\big)\Big) \le s(t_m - t_\ell) \quad (3)$$

To see why this represents the requirement stated in (ii) above, note that for any job J_i with $d_i \le t_m$, $\big(\sum_{j=\ell}^{m-1} x_{i,j}\big)$ represents the remaining execution requirement of job J_i at time-instant t_ℓ. The outer summation on the LHS is simply summing this remaining execution requirement over all the HI-criticality jobs that have deadlines at or prior to t_m.

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

(v) The Constraints (3) above only prevent missed deadlines after t_ℓ when the (degraded) processor is continually busy over the interval between t_ℓ and the missed deadline; what about deadline misses when the processor is

---

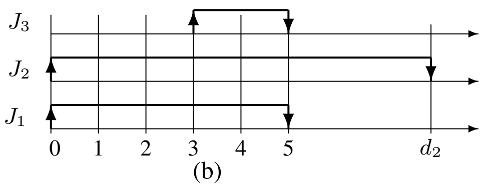
Given MC instance $(\{J_1, J_2, \ldots, J_n\}, s)$, with job release-dates and deadlines partitioning the time-line over $[\min_i\{a_i\}, \max_i\{d_i\})$ into the k intervals $I_1, I_2, \ldots, I_k$
Determine values for the x_{ij} variables, $i = 1, \ldots, n, j = 1, \ldots, k$ satisfying the following **constraints:**

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

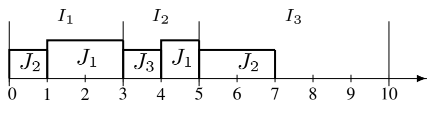
Fig. 1. Linear program for constructing the scheduling tables

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

(a)



(b)



(c)

Fig. 2. Illustrating Example 2. The jobs are listed in (a), and depicted graphically in (b). The scheduling table that is constructed is depicted in (c).

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

Given a solution to this linear program, we construct a scheduling table that assigns job J_i an amount $x_{i,j}$ of execution during the interval I_ℓ, for each pair (i, ℓ); in I_ℓ, HI-criticality execution is performed before LO-criticality execution – the jobs may be executed in any order within each criticality level. During run-time, scheduling decisions are initially made according to this scheduling table. If a processor failure is detected, the table is no longer used; instead, all LO-criticality jobs are discarded and the remaining HI-criticality jobs are executed acording to EDF.

The entire linear program is listed in Figure 1; we now illustrate the construction of such a linear program by means of a simple example.

Example 2: We will consider a MC instance I consisting of three jobs with parameters as depicted in Figure 2(a), with c_2's value left unspecified for now, and d_2 assumed to be larger than 5. The release dates and deadlines of these three jobs define three intervals: $I_1 = [0, 3)$; $I_2 = [3, 5)$; $I_3 = [5, d_2)$, as illustrated in Figure 2(b).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

As a consequence, the scheduling table would be as depicted in Figure 2(c). We can easily see that this scheduling table yields a correct scheduling strategy: observe that there are three contiguous blocks of HI-criticality execution: $[0, 1)$, $[3, 4)$, and $[5, 7)$, and consider the possibility of the processor degrading at the start of each:

- If the processor failed during $[0, 1)$, then J_2 can execute over $[0, 3)$ and $[5, 8)$, while J_3 can execute over $[3, 5)$. Both HI-criticality jobs would meet thus their deadlines on the speed-0.5 processor.
- If the processor failed during $[3, 4)$, then J_3 would execute over $[3, 5)$. J_2 will have completed one unit of

execution prior to the processor failing, and therefore need two additional units of execution. This it will obtain by executing over $[5, 9)$ on the speed-0.5 processor. If the processor failed during $[5, 7)$, then J_2 will have completed one unit of execution prior to the processor failing. It needs two more units, which it will obtain by executing over $[5, 9)$ on the speed-0.5 processor.

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

Bounding the size of this LP. It is not difficult to show that the LP of Figure 1 is of size polynomial in the number of jobs n in MC instance I:

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

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.

IV. An optimization problem

Given a collection $\mathcal{J}$ of MC jobs and a degraded processor speed s, in Section III above we described how to obtain a correct scheduling strategy for the MC instance $(\mathcal{J}, s)$. We now consider an *optimization* version of this problem: given the collection of MC jobs $\mathcal{J}$, what is the *smallest* s such that there is a correct scheduling strategy for the instance $(\mathcal{J}, \mathrm{s})$? Lemma 1 gives us a lower bound: s can be no smaller than the speed of the slowest processor upon which the HI-criticality jobs in $\mathcal{J}$ would be correctly scheduled by EDF. But is this lower bound tight? The following example illustrates that it is not:

Example 3: Consider the following three MC jobs:

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

It is evident that

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

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

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

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

It turns out that a slight modification to the linear program of Figure 1 can be used to determine the smallest speed s: we simply add the objective function

$$\textit{minimize } s$$

to our linear program of Figure 1. That is, our modified linear program computes those values of the $x_{i,j}$ parameters that yield a scheduling strategy guaranteeing to meet all deadlines on a unit-speed processor, and HI-criticality jobs' deadlines when the degraded speed is *the smallest possible*; this smallest speed is the desired solution to the optimization version of our MC scheduling problem.

We have implemented this optimization algorithm, and have conducted simulation experiments on randomly-generated MC instances to try and gain some insight into the tradeoffs involved in MC scheduling upon varying-speed processors. We now describe these empirical investigations.

Workload generation. Each randomly-generated MC instance is characterized by four parameters:

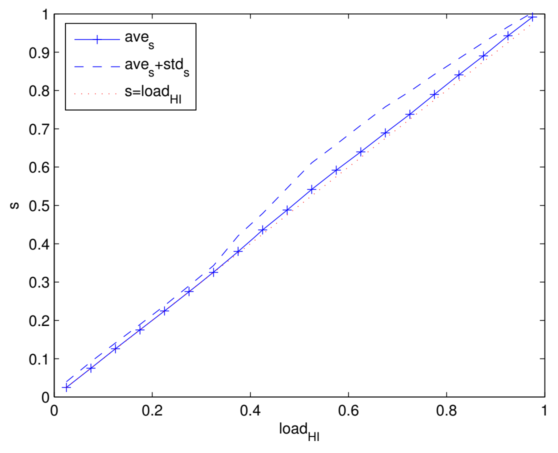
1) n, the total number of jobs in the instance.
2) u_{all}, a measure of the computational load of the instance. This is equal to the sum of the WCETs of all the jobs in the instance, normalized by the duration of time spanned by their scheduling windows[2].
3) γ, the expected fraction of jobs that are of HI criticality.
4) ζ, the expected number of jobs with scheduling windows that overlap (cover) each time instant. A value $\zeta = 1$ suggests that there are no overlaps between the scheduling windows of any pair of jobs, while $\zeta = n$ means that all jobs have the same release date and deadline).

With values specified for these four parameters, the individual jobs comprising the instance are generated randomly according to a procedure that is described in detail in the appendix.

Experiments and Observations. We generated a total of 30,000 MC workload instances, for various different combinations of the four parameters described above. For each instance that we generated, we also computed two *load*[3] parameters — its HI-criticality load (load$_{\text{HI}}$) and its total load (load$_{\text{ALL}}$). Our observations are depicted in graphical form in Figures 3-4.

[2]The *scheduling window* of a job is the duration between the job's release time and its deadline.

[3]See, e.g. [13, p. 81] for the definition of the *load*, or *loading factor*, of a collection of jobs; it is known that the load is equal to the speed of the smallest processor upon which such a collection can be scheduled using preemptive EDF. For our instances, the HI-criticality load is the load of only the HI-criticality jobs in the instance, whereas the total load is the load of all the jobs in the instance.



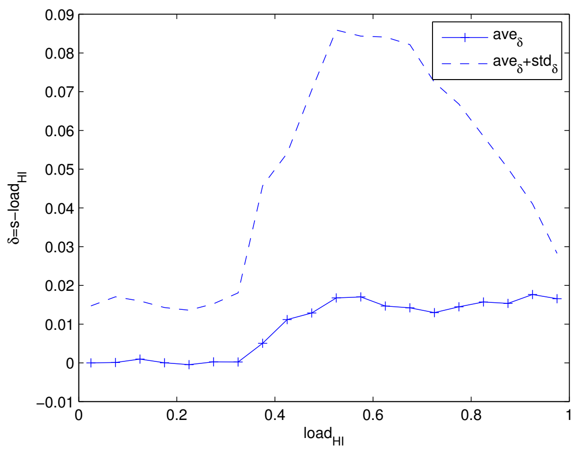


Fig. 3. Degraded speed as a function of HI-criticality load

In both graphs of Figure 3, the x-axes represent the HI-criticality load of the MC instance under consideration. The y-axis of the left graph represents the degraded speed s that the instance can tolerate, as computed by our optimization algorithm. By Lemma 1 the loading factor of the HI-criticality jobs is a lower bound on the degraded speed for which a correct scheduling strategy may exist — this lower bound is depicted as a dotted line in this graph, while the y-axis of the right graph represents the amount by which the computed degraded speed s exceeds this lower bound. Although we do not claim that our simulations are extensive or comprehensive enough to draw conclusions with absolute certainty, the evidence presented in these graphs does indicate that the actual minimum speed (as computed by our linear program) for which the typical randomly-generated MC instance is correctly schedulable, is very close to the lower bound implied by Lemma 1.

Figure 4 depicts the relationship between the *total* load of the instance, and the amount by which the computed degraded speed s exceeds the lower bound of Lemma 1. It is not surprising that s tends to diverge from the lower bound with increasing load$_{\text{ALL}}$: the intuition behind this is that since the contribution of the LO-criticality jobs to load$_{\text{ALL}}$ also increases, LO-criticality jobs leave fewer time demands for the HI-criticality jobs to "extend" in degraded mode.

V. Non-preemptive scheduling

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

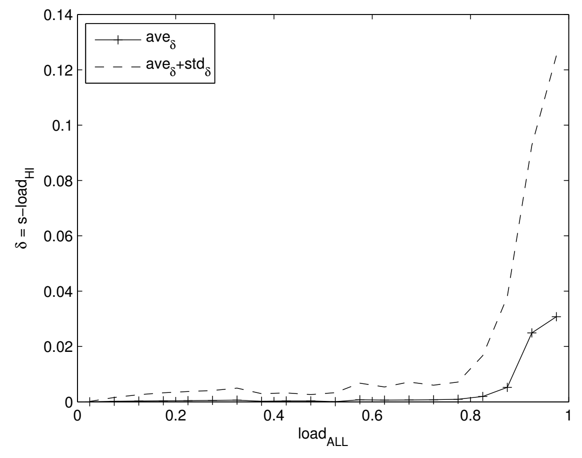


Fig. 4. Degraded speed as a function of total load

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 1: 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: By transformation from the partitioning problem [7]; details omitted. ■

VI. Recurrent tasks

In Sections III and V 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], [14]. In this section, we briefly consider this more difficult problem of scheduling mixed-criticality systems modeled as collections of sporadic tasks. As with traditional (i.e., non MC) real-time systems, we will model a MC real-time system τ as being comprised of a finite specified collection of MC recurrent tasks, each of which will generate a potentially infinite sequence of MC jobs. We restrict our attention here to *implicit-deadline MC sporadic tasks*. Each task is characterized by a 3-tuple of parameters: $\tau_i = (C_i, T_i, \chi_i)$, with the following interpretation. Task τ_i generates a potentially infinite sequence of jobs, with successive jobs being released at least T_i time units apart. Each such job has a criticality χ_i, a WCET C_i, and a deadline that is T_i time units after its release. The quantity $U_i = C_i/T_i$ is referred to as the *utilization* of τ_i. An *implicit-deadline MC sporadic task system* is specified by specifying a finite number $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of such sporadic tasks, and the degraded processor speed $s < 1$ (as with MC instances of independent jobs, it is assumed that the normal processor speed is one). 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 2) a necessary condition for the existence of a correct scheduling strategy. We then present a scheduling strategy, *Algorithm preemptive-MC*, and prove (Theorem 3) that it is optimal.

Theorem 2: 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-faulty– processor), assign a share U_i of the processor to each task τ_i during each instant that is active[5].
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 3: 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 2.

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 2 ensures that there is sufficient computing capacity on the

[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.

[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.

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 1), this problem, too, can be shown to be highly intractable.

VII. Context & Conclusions

In this paper we have presented the findings of our initial research into scheduling mixed-criticality systems upon variable-speed uniprocessors. While we expect that these processors are very likely to execute at unit speed (or faster) during run-time, we can only guarantee at a high level of assurance that they will execute at some speed $s < 1$. Upon such a processor, the scheduling objective is to ensure that all jobs complete in a timely manner if the processor speed is one, while simultaneously ensuring that more critical jobs complete in a timely manner even if the processor speed falls to as low as s.

The research reported in this paper can be extended in several directions. An obvious extension would be to *more than just two criticality levels*. Such an extension gives rise to some interesting questions concerning, e.g., tradeoffs: does the processor speed at which a processor is deemed to have degraded one criticality level impact on the processor speed at which it will degrade further criticality levels? If so, what are the factors that the system designer should keep in mind in deciding what the criteria are for deeming a degradation in processor performance?

The optimization problem considered in Section IV seeks to determine the smallest processor speed for which the HI-criticality workload can be guaranteed. A different optimization problem may fix this speed, and instead seek to determine the smallest speed at which the run-time dispatcher would be forced to abandon the LO-criticality jobs.

We have assumed here that a platform "knows" its execution speed at each instant during run-time; specifically, that the scheduling algorithm knows when the processor speed falls below a certain threshold. It would be particularly interesting and important to derive algorithms for scheduling mixed-criticality systems upon platforms that do not have such self-awareness; such scheduling algorithms would need to guarantee that all jobs meet their deadlines upon a normal processor and that all HI-criticality jobs meet their deadlines on a degraded processor, *without* knowing during run-time whether the processor is normal or degraded. We have obtained some initial results [8] concerning MC scheduling on such non-monitoring platforms, but much remains to be done.

In this paper, we have considered the possibility of the processor transitioning from normal to faulty mode. It is likely in practice that a faulty processor may resume normal operation after some duration in faulty mode; were this to happen, it would be desirable to have the system recover from the fault and resume the execution of LO-criticality jobs. Devising scheduling strategies that achieve this requires careful consideration of models of desired behavior, in order to come up with appropriate quantitative metrics that a scheduling strategy may seek to optimize.

The preemptive scheduling strategies presented in Sections III-IV of this paper are designed for a model of execution that assumes that preemptions incur no cost, and therefore make no attempt to bound the number of such preemptions. We are currently working on scheduling strategies that continue to allow preemptions to occur, but seek to bound their number.

It would be interesting to integrate the model we are proposing here with other mixed-criticality models: what if we were to have multiple WCET's specified *in addition to* variable-speed processors?

Acknowledgements

This work was supported in part by NSF grants CNS 1016954, CNS 1115284, and CNS 1218693; and ARO grant W911NF-09-1-0535.

References

[1] N. Bansal, H.-L. Chan, R. Khandekar, K. Pruhs, B. Schicber, and C. Stein. Non-preemptive min-sum scheduling with resource augmentation. In *Foundations of Computer Science, 2007. FOCS '07. 48th Annual IEEE Symposium on*, pages 614–624, 2007.

[2] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012. IEEE Computer Society.

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

[4] S. K. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 61(8):1140–1152, 2012.

[5] D. Bull, S. Das, K. Shivshankarand, G. Dasika, K. Flautner, and D. Blaauw. A power-efficient 32b ARM ISA processor using timing-error detection and correction for transient- error tolerance and adaptation to PVT variation. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 284–285, 2010.

[6] A. Burns and R. Davis. Mixed-criticality systems: A review. Available at `http://www-users.cs.york.ac.uk/~burns/review.pdf`, 2013.

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

[8] Z. Guo and S. Baruah. Mixed-criticality scheduling upon unmonitored unreliable processors. In *Proceedings of the IEEE Symposium on Industrial Embedded Systems (SIES)*. IEEE Computer Society Press, June 2013.

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

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

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

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

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

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

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

[16] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.

APPENDIX: WORKLOAD GENERATION FOR SIMULATION EXPERIMENTS

Recall that each generated workload instance is characterized by the four parameters $(n, u_{\text{all}}, \gamma, \zeta)$, with n denoting the number of jobs, u_{all} a measure of the computational load, γ the expected fraction of HI-criticality jobs, and ζ the expected number of jobs with scheduling windows covering each time instant. With these four parameters specified, the individual jobs are generated as follows:

§***1: Release dates***. We model job arrivals by a (memoryless) Poisson process. I.e., we generate $(n-1)$ independent and identically distributed random variables x_i according to the exponential distribution with $\lambda = 1$. The first job is assigned release date zero ($a_1 := 0$); subsequent release dates are assigned values as $a_{i+1} := a_i + x_i$.

§***2: Deadlines***. We follow the procedure suggested in [3] and model relative deadlines (the duration between release date and deadline) as independent and identically distributed random variables drawn from the log-uniform distribution (exponential of uniform distribution $U[b_l, b_u]$).

To obtain the desired values we chose $b_l \leftarrow 0$ and b_u to be the solution to the equation $e_u^b - \zeta b_u - 1 = 0$ (the equation is solved numerically using the Newton-Raphson method), so that expectation for the log-uniform distribution is $E(c) = (e^{b_u} - e^{b_l})/(b_u - b_l) = \zeta$. Since in expectation, a job is released every ($\lambda = 1$) time unit[s], and will have a scheduling window of duration ζ time units, the expected number of jobs with scheduling windows covering each time instant approaches ζ with increasing n.

§***3: Criticality Level***. Each job is assigned criticality HI with probability γ (and hence, criticality LO with probability $(1-\gamma)$).

§***4: Worst Case Execution Time (WCET)***. Once all the release dates and deadlines have been assigned, we can determine the total duration of time covered by all the jobs' scheduling windows — this is equal to the latest deadline *minus* the duration of those intervals that do not lie within any scheduling window. Let L_{act} denote this duration. The parameter u_{all} characterizing this workload now determines the cumulative WCETs of all the jobs: $\sum_i c_i = \sigma := u_{\text{all}} L_{\text{act}}$.

An additional straightforward restriction on the WCET of each job is that it cannot exceed the relative deadline of the job. Let d'_i denote the relative deadline of the i'th job. Our method generates WCET one by one in increasing order of relative deadline: In the generation of the i'th WCET c_i, given $c_1, \ldots, c_{i-1}$, the following two inequalities may provide a tighter bound:

$$\begin{aligned} c_i &\geq \sigma - \sum_{j=1}^{i-1} c_j - \sum_{j=i+1}^{n} d'_j \\ c_i &\leq \sigma - \sum_{j=1}^{i-1} c_j. \end{aligned}$$

It is evident that if either of these equations is violated, the sum of all the WCETs will not equal σ no matter what values the remaining c_j take in their respective ranges $[0, d'_j], j = i+1, \ldots, n$.

Thus for each of $i = 2, \ldots, n-1$, the bound for generating the WCET should be

$$\begin{aligned} c_i \geq lb(c_i) &:= \max\{0, \sigma - \sum_{j=1}^{i-1} c_j - \sum_{j=i+1}^{n} d'_j\} \\ c_i \leq ub(c_i) &:= \min\{d'_i, \sigma - \sum_{j=1}^{i-1} c_j\} \end{aligned}$$

The bound of c_1 is simpler, with $lb(c_1) = \max\{0, \sigma - \sum_{j=2}^{n} d'_j\}$ and $ub(c_1) = d'_1$; and c_n is set equal to $\sigma - \sum_{j=1}^{n-1} c_j$. Note that we will only discuss how to randomly generate $c_1, \ldots, c_{n-1}$ properly in the following, thus i will only take values from 1 to $n-1$.

Although we have determined upper and lower bounds on each c_i value, we cannot simply choose the c_i's uniformly in the calculated range $[lb(c_i), ub(c_i)]$. In order to ensure an unbiased random generation, the expectation (i.e., the mean value) of each WCET needs to be fixed, and may not be $(lb(c_i) + ub(c_i))/2$. Here we assume the sum of the WCETs, which equals σ, is to be shared "fairly" according to relative deadlines. In this context, fairness would dictate that the jobs with longer relative deadline d'_i gets a relatively larger expectation of WECT c_i. More precisely, we desire that the expected values $E(C_i)$ of the WCET's – the c_i values – satisfy

$$E(c_i) = \sigma \times \left(d'_i / \Big(\sum_{i=1}^{n} d'_i \Big) \right)$$

We have chosen the beta distribution to generate these random values c_i within the computed ranges $[lb(c_i), ub(c_i)]$ and the desired expected value $E(c_i)$. One the parameters of beta distribution is fixed to be $\alpha(c_i) = 2$, and the other is given by

$$\beta(c_i) = 2 \times \left(\frac{ub(c_i) - E(c_i)}{E(c_i) - lb(c_i)} \right)$$

Since the beta distribution generates random values over $[0, 1]$ with expectation value of $\alpha/(\alpha + \beta) = (E(c_i) - lb(c_i))/(ub(c_i) - lb(c_i))$, we need to scale the values into the ranges $[lb(c_i), ub(c_i)]$ by multiplying by $(ub(c_i) - lb(c_i))$ and adding $lb(c_i)$. This ensures that the expectation of c_i is $(E(c_i) - lb(c_i))/(ub(c_i) - lb(c_i)) \times (ub(c_i) - lb(c_i)) + lb(c_i)$ which is equal to $E(c_i)$ as desired.