An algorithm for scheduling certifiable mixed-criticality sporadic task systems

Haohan Li Sanjoy Baruah
The University of North Carolina at Chapel Hill

Abstract

Many safety-critical embedded systems are subject to certification requirements. However, only a subset of the functionality of the system may be safety-critical and hence subject to certification; the rest of the functionality is non safety-critical and does not need to be certified. Certification requirements in such "mixed-criticality" systems give rise to some interesting scheduling problems, that cannot be satisfactorily addressed using techniques from conventional scheduling theory. In prior work, we have studied the scheduling and analysis of mixed criticality systems that are specified as finite collections of jobs executing on a single shared preemptive processor. In this paper, we consider mixed criticality systems that are comprised of finite collections of recurrent tasks, specified using a mixed-criticality generalization of the widely-used sporadic tasks model. We design a priority-based algorithm for scheduling such systems, derive an algorithm for computing priorities, and obtain a sufficient schedulability condition for efficiently determining whether a given mixed-criticality system can be successfully scheduled by this algorithm.

Keywords. Preemptive uniprocessors; certification; sporadic task systems; on-line scheduling.

1 Introduction

Many safety-critical systems are subject to *certification* requirements: their safety-critical functionalities must be certified correct by statutory certification authorities (CAs). However, the current trend towards integrating multiple functionalities on a common platform, driven primarily by cost and related concerns, means that it is typically the case that only a relatively small fraction of the overall system is of critical functionality and hence needs to be certified. The remainder of the system is comprised of non-critical code that performs non-critical functions and is therefore not subject to certification. Such *mixed criticality* systems are becoming increasingly common in embedded systems, and coming up with procedures that will allow for the cost-effective certification of mixed-criticality systems has been identified as a unique, particularly challenging, collection of problems [3]. Recognizing these challenges, several US government R&D organizations including AFRL, NSF, NSA, NASA, etc., have led initiatives such as the Mixed Criticality Architecture Requirements (MCAR) program aimed at streamlining the certification process for safety-critical embedded systems; these initiatives have brought together participants from industry, academia, and standards bodies to seek out more advanced, efficient, and cost-effective certification processes. The research reported in this paper is part of this attempt: we seek scheduling policies for such mixed-criticality systems that are able to both facilitate the certification process *and* make good use of the computing resources of the embedded platform.

In order to certify a system as being correct, the certification authority (CA) makes some assumptions about the worst-case behavior of the system. In this paper, we focus on one particular aspect of run-time behavior: the *worst-case execution time (WCET)* of pieces of code. CA's tend to be very conservative, and hence the WCET estimates used by the CA is often far more pessimistic than those the system designer would typically use during the system design process. On the other hand, while the CA is only concerned with the correctness of the safety-critical part of the system the system designer wishes to ensure that the entire system is correct, including the non-critical parts. We illustrate with a simple (contrived) example.

Example 1 Consider a system to be implemented on a preemptive uniprocessor, that is comprised of three jobs J_1, J_2, and J_3, all of which are released at time zero. Job J_1 has a deadline at time-instant 2, while the other two jobs have their deadlines at time-instant 3.5. Jobs J_2 and J_3 are high-criticality and subject to certification, whereas J_1 is low-criticality and hence not.

- The system designer is confident that each job has a WCET not exceeding 1. Hence executing the jobs in earliest deadline first (EDF) [11, 8] order will ensure that all three meet their deadlines.
- However, the CA uses more pessimistic WCET estimates during the certification process, and claims that jobs J_2 and J_3 may each need 1.5 time units of execution[1].

[1] The CA may also determine that the low-criticality job J_1 needs more than 1 time unit to complete execution; however, let us assume for now that the system is implemented to abort the execution of J_1 if it fails to complete execution within 1 time-unit.

If the system were indeed scheduled using EDF, the CA would determine that in the worst case, J_1 executes over $[0, 1)$ and the job from among J_2 and J_3 that is next chosen for execution will execute for 1.5 time units, thereby causing the other high-criticality job to miss its deadline at time 3.5. *The system scheduled using EDF would therefore fail certification.*

On the other hand if we were to assign greater priority to the high-criticality jobs, then they would both meet their deadlines even under the worst-case scenarios envisioned by the CA. *However the low-criticality job J_1 will miss its deadline even when each job executes for at most 1 time unit (as predicted by the system designer).*

It turns out that the following scheduling strategy for this system both passes certification *and* meets all deadlines if the jobs behave as expected by the system designer:

- Execute J_2 over $[0, 1)$.
- If J_2 completes execution at time-instant 1, then execute J_1 over $[1, 2)$ and J_3 over $[2, 3.5)$, thereby ensuring that all deadlines are met.
- If J_2 does not complete execution by time-instant 1, then discard J_1 and continue the execution of J_2, following that with the execution of J_3 over $[1.5, 3)$. Both the high-criticality jobs will complete by their deadlines in the worst-case scenario envisioned by the CA.

■

This research. The central thesis of our research is that the efficient utilization of computing resources in mixed-criticality systems that are subject to certification requirements requires the development of new scheduling theory. In prior work [4, 10], we have studied mixed-criticality (MC) systems implemented on a preemptive uniprocessor platform that can be modeled, as in the example above, as finite collections of jobs. However, most real-time systems are better modeled as collections of *recurrent processes* that are specified using, e.g., the sporadic tasks model [13, 5]. Schedulability analysis of such systems is typically far more difficult than the analysis of systems modeled as collections of independent jobs, since (i) a sporadic task system can generate infinitely many jobs during any one run; and (ii) the collection of jobs generated during different runs of the system may be different: in general, a single system may legally give rise to infinitely many different collections of jobs. In this paper, we study this more difficult problem of *scheduling mixed-criticality systems modeled as collections of sporadic tasks, upon a single shared preemptive processor.* As in [4, 10] the insight we seek to exploit is this. Certification is performed under conservative assumptions: the CA makes very pessimistic assumptions about the run-time behavior of the system, and requires that it be demonstrated correct under these pessimistic assumptions. In order to perform system certification under such pessimism one must, informally speaking, severely *over-provision* computing resources to the part of the system needing certification. Some of this over-provisioned capacity could then be *reclaimed*, during system design and analysis time itself, to make performance guarantees to the remainder of the system, since these guarantees are made under a correspondingly lower degree of pessimism.

Organization of this paper. In Section 2, we present the formal model for representing mixed-criticality real-time systems that is used in this research. This formal model extends the conventional models of real-time jobs and of sporadic tasks [13, 5] by allowing for the specification of a criticality level and two different WCET's, one at each criticality level, for each job. In Section 3 we derive, and prove the correctness of, a new algorithm for scheduling mixed-criticality sporadic task systems upon a preemptive uniprocessor platform. We also derive a sufficient condition for determining whether any given task system can be correctly scheduled by our scheduling algorithm. In Section 4 we briefly survey some other work on mixed-criticality real-time systems, highlighting in particular on research that has focused on certification-cognizant scheduling in such systems.

2 Model and definitions

In this section we formally define the mixed-criticality workload model that is used in this paper, and explain terms and concepts used throughout the remainder of this document. 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 sporadic tasks, each of which will generate a potentially infinite sequence of MC jobs.

§. **MC jobs.** Each job is characterized by a 5-tuple of parameters: $J_i = (a_i, d_i, \chi_i, c_i(\text{LO}), c_i(\text{HI}))$, where

- $a_i \in R^+$ is the release time.
- $d_i \in R^+$ is the deadline. We assume that $d_i \geq a_i$.
- $\chi_i \in \{\text{LO}, \text{HI}\}$ denotes the criticality of the job. A HI-criticality job (a J_i with $\chi_i = \text{HI}$) is one that is subject to certification, whereas a LO-criticality job (a J_i with $\chi_i = \text{LO}$) is one that does not need to be certified.
- $c_i(\text{LO})$ specifies the worst case execution time (WCET) estimate of J_i that is used by the system designer (i.e., the WCET estimate at the LO criticality level).
- $c_i(\text{HI})$ specifies the worst case execution time (WCET) estimate of J_i that is used by the certification authorities (i.e., the WCET estimate at the HI criticality level).We assume that
  - $c_i(\text{HI}) \geq c_i(\text{LO})$ (i.e., the WCET estimate used by the system designer is never more pessimistic than the one used by the CA), and

– $c_i(\text{HI}) = c_i(\text{LO})$ if $\chi_i = \text{LO}$ (i.e., a LO-criticality job is aborted if it executes for more than its LO-criticality WCET estimate[2]).

The MC job model has the following semantics. Job J_i is released at time a_i, has a deadline at d_i, and needs to execute for some amount of time γ_i. However, the value of γ_i is not known beforehand, but only becomes revealed by actually executing the job until it *signals* that it has completed execution. If J_i signals completion without exceeding $c_i(\text{LO})$ units of execution, we say that it has exhibited LO-criticality behavior; if it signals completion after executing for more than $c_i(\text{LO})$ but no more than $c_i(\text{HI})$ units of execution, we say that it has exhibited HI-criticality behavior. If it does not signal completion upon having executed for $c_i(\text{HI})$ units, we say that its behavior is erroneous.

In prior work [4, 10], we have studied the scheduling of MC ***instances*** that are specified as collections of such MC jobs: some of the results in [4, 10] are discussed in Section 2.1.

§. **MC sporadic tasks.** Each sporadic task in the MC model is also characterized by a 5-tuple of parameters: $\tau_k = (\chi_k, C_k(\text{LO}), C_k(\text{HI}), D_k, T_k)$, with the following interpretation. Task τ_k generates a potentially infinite sequence of jobs, with successive jobs being released at least T_k time units apart. Each such job has a deadline that is D_k time units after its release. The criticality of each such job is χ_k, and it has LO-criticality and HI-criticality WCET's of $C_k(\text{LO})$ and $C_k(\text{HI})$ respectively.

A MC ***sporadic task system*** is specified by specifying a finite number of such sporadic tasks. As with traditional (non-MC) systems, such a MC sporadic task system can potentially generate infinitely many different MC instances (collections of jobs), each instance being obtained by taking the union of one sequence of jobs generated by each sporadic task.

A note on terminology. In this paper, we use the term *instance* to denote a collection of jobs, and the term *system* to denote a collection of tasks. Thus, each system τ can legally generate many (except in trivial cases, infinitely many) distinct instances.

§. **Loads ℓ_{LO} and ℓ_{HI}.** In classical real-time scheduling theory (see, e.g.,[12, page 81]), the *load* of an instance denotes the maximum over all time intervals, of the cumulative execution requirement by jobs of the instance over the interval, normalized by the interval length. Informally, the load of an instance represents a lower bound on the speed of any processor upon which it can meet all deadlines.

[2]We assume that the run-time system provides support for ensuring that jobs do not execute for more than a specified amount.

Analogous to this concept, we define two loads, $\ell_{\text{LO}}(I)$ and $\ell_{\text{HI}}(I)$, of a MC instance I:

Definition 1 *The* LO*-criticality load $\ell_{\text{LO}}(I)$ and the* HI*-criticality load $\ell_{\text{HI}}(I)$ of a mixed-criticality instance I are defined according to the following two formulas:*

$$\ell_{\text{LO}}(I) = \max_{0 \le t_1 < t_2} \frac{\sum_{J_i \,:\, t_1 \le a_i \wedge d_i \le t_2} c_i(\text{LO})}{t_2 - t_1}$$

$$\ell_{\text{HI}}(I) = \max_{0 \le t_1 < t_2} \frac{\sum_{J_i \,:\, \chi_i = \text{HI} \wedge t_1 \le a_i \wedge d_i \le t_2} c_i(\text{HI})}{t_2 - t_1}$$

These definitions extend in the obvious manner to systems of sporadic tasks: The LO-criticality load $\ell_{\text{LO}}(\tau)$ of sporadic task system τ is defined to be the largest value that $\ell_{\text{LO}}(I)$ can have, for any instance I generated by τ. The HI-criticality load $\ell_{\text{HI}}(\tau)$ is defined analogously: it is the largest value that $\ell_{\text{HI}}(I)$ can have, for any instance I generated by τ. For any τ, $\ell_{\text{LO}}(\tau)$ and $\ell_{\text{HI}}(\tau)$ can be computed using well-known techniques (see, e.g., [6]) for determining the loads of "regular" (i.e., non MC) sporadic task systems. Specifically, $\ell_{\text{LO}}(\tau)$ is the load of the regular sporadic task[3] system $\{(C_k(\text{LO}), D_k, T_k) \mid \tau_k \in \tau\}$ while $\ell_{\text{HI}}(\tau)$ is the load of the regular sporadic task system $\{(C_k(\text{HI}), D_k, T_k) \mid \tau_k \in \tau \wedge \chi_k = \text{HI}\}$.

§. **Scheduling MC sporadic task systems.** As stated above, the same sporadic task system may generate different instances of jobs during different runs. Furthermore, during any given run each job comprising the instance may exhibit LO-criticality, HI-criticality, or erroneous behavior. We define an algorithm for scheduling sporadic task system τ to be *correct* if it is able to schedule every instance generated by τ such that

- If all jobs exhibit LO-criticality behavior, then all jobs receive enough execution between their release time and deadline to be able to signal completion; and
- If *any* job exhibits HI-criticality behavior, then all HI-criticality jobs receive enough execution between their release time and deadline to be able to signal completion.

Note that if any job exhibits HI-criticality behavior, we do not require any LO-criticality jobs (including those that may have arrived before this happened) to complete by their deadlines. This is an implication of the requirements of certification: informally speaking, the system designer fully expects that all jobs will exhibit LO-criticality behavior, and hence is only concerned that they behave as desired under these circumstances. The CA, on the other hand, allows for the possibility that some jobs may exhibit HI-criticality behavior, and requires that all HI-criticality jobs nevertheless meet their deadlines.

[3]Here, a regular sporadic task is represented by a 3-tuple of its WCET, relative deadline, and period parameters.

2.1 The OCBP scheduling algorithm

In prior work [4, 10], we considered the scheduling of MC instances: workloads that are specified as collections of independent jobs rather than as systems of recurrent tasks. We defined a priority-based algorithm called ***OCBP*** (*Own Criticality-Based Priorities*) for scheduling such instances, which we now describe.

The high-level description of the OCBP algorithm is as follows. Given such an instance I, we determine off-line (i.e., prior to run-time) a total priority ordering of the jobs of I such that scheduling the jobs according to this priority ordering guarantees a correct schedule, where *scheduling according to a priority ordering* means that at each moment in time the highest-priority available job is executed.

The priority ordering is constructed recursively using the approach commonly referred to in the real-time scheduling literature as the "Audsley approach" [2]. We first determine the lowest priority job: a Job J_i may be assigned the lowest priority if

- it is a LO-criticality job ($\chi_i = \text{LO}$), and there is at least $c_i(\text{LO})$ time between its release time and its deadline available if every other job J_j has higher priority and is executed for $c_j(\text{LO})$ time units; or
- it is a HI-criticality job ($\chi_i = \text{HI}$), and there is at least $c_i(\text{HI})$ time between its release time and its deadline available if every other job J_j has higher priority and is executed for $c_j(\text{HI})$ time units.

In general, several jobs may be eligible to be assigned the lowest priority; we can arbitrarily choose one of these. The above procedure is then repeated on the collection of jobs excluding this lowest priority job, until all jobs are ordered, or at some iteration no job is eligible to be assigned lowest priority. (If this happens, the priority-assignment algorithm reports failure and we say that the instance is not OCBP-schedulable.) We illustrate the operation of the OCBP priority assignment algorithm by an example:

Example 2 Consider the instance comprised of the following three jobs. J_1 is not subject to certification, whereas J_2 and J_3 must be certified correct.

J_i	a_i	d_i	χ_i	$c_i(\text{LO})$	$c_i(\text{HI})$
J_1	0	4	LO	2	2
J_2	0	5	HI	2	4
J_3	0	10	HI	2	4

Let us determine which, if any, of these jobs could be assigned lowest priority according to the OCBP priority assignment algorithm:

- If J_1 were assigned lowest priority, J_2 and J_3 could consume $c_2(\text{LO}) + c_3(\text{LO}) = 2 + 2 = 4$ units of processor capacity over $[0, 4)$, thus leaving no execution for J_1 prior to its deadline.

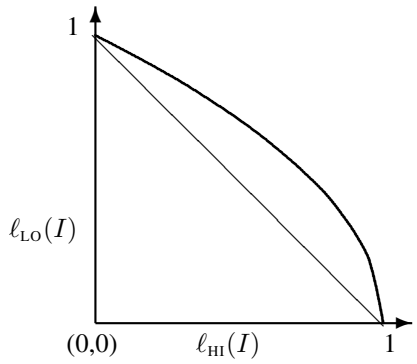

Figure 1. Bound on the LO-criticality load (ℓ_{LO}) as a function of HI-criticality load (ℓ_{HI}).

- If J_2 were assigned lowest priority, J_1 and J_3 could consume $c_1(\text{HI})+c_3(\text{HI}) = 2+4 = 6$ units of processor capacity over $[0, 6)$, thus leaving no execution for J_2 prior to its deadline at time-instant 5.
- If J_3 were assigned lowest priority, J_1 and J_2 could consume $c_1(\text{HI}) + c_2(\text{HI}) = 2 + 4 = 6$ units of processor capacity over $[0, 6)$. This leaves 4 units of execution for J_3 prior to its deadline at time-instant 10, which is sufficient for J_3 to execute for $c_3(\text{HI}) = 4$ time units. *Job J_3 may therefore be assigned the lowest priority.*

Next, the OCBP priority assignment algorithm would consider the instance $\{J_1, J_2\}$, and seek to assign one of these jobs the lower priority:

- If J_1 were assigned lower priority, J_2 could consume $c_2(\text{LO}) = 2$ units of processor capacity over $[0, 2)$. This leaves 2 units of execution for J_1 prior to its deadline at time-instant 4, which is sufficient for J_1 to execute for $c_1(\text{LO}) = 2$ time units. *Job J_1 may therefore be assigned the lowest priority from among $\{J_1, J_2\}$.*

The final OCBP priority ordering is therefore as follows. Job J_2 has the greatest priority, job J_1 has the next-highest priority, and J_3 has the lowest priority. It may be verified that scheduling according to these priorities is a correct MC scheduling strategy for the instance $\{J_1, J_2, J_3\}$. ■

The following properties of OCBP were proved in [4, 10]:

1. If the OCBP priority assignment algorithm succeeds in assigning priorities to the jobs of an instance I, then priority-based scheduling of I according to these priorities is a correct MC scheduling strategy.
2. The OCBP priority assignment algorithm succeeds in assigning priorities to the jobs of any instance I that satisfies

$$\ell_{\text{LO}}(I)^2 + \ell_{\text{HI}}(I) \leq 1\,. \tag{1}$$

We plot in Figure 1 the bound of Equation 1 on the LO-criticality load of an instance I as a function of its HI-criticality load $\ell_{\text{HI}}(I)$, in order for I to be successfully scheduled. The curve that connects the points $(1, 0)$ and

$(0,1)$ has equation $\ell_{\rm LO}(I)^2 + \ell_{\rm HI}(I) = 1$, and represents the OCBP schedulability condition of Equation 1. Any instance that maps on to a point beneath this curve is guaranteed, as a consequence of Equation 1, to be schedulable by OCBP.

The following technical lemma, which we will use later, follows from the properties of OCBP priority assignment.

Lemma 1 *Let $\{J_i\}_{i=1}^n$ denote a collection of jobs, all of which are released at time-instant t. Let $\mathbf{pr} : \{J_i\}_{i=1}^n \longrightarrow \{1, 2, \ldots, n\}$ be any bijective function from $\{J_i\}_{i=1}^n$ to the integers $\{1, \ldots, n\}$. Under the interpretation that $\mathbf{pr}(J_k)$ denotes the priority assigned to job J_k (in keeping with convention, we assume that smaller numbers denote greater priority), $\mathbf{pr}$ is an OCBP priority assignment for $\{J_i\}_{i=1}^n$ if and only if*

$$\forall\, k \;::\; \sum_{J_i \,:\, \mathbf{pr}(J_i) \leq \mathbf{pr}(J_k)} c_i(\chi_k) \leq (d_k - t)\;. \qquad (2)$$

Proof: Job J_k may be assigned the priority $\mathbf{pr}(J_k)$ if and only if it receives at least $c_k(\chi_k)$ units of execution by its deadline when each job J_i that is (eventually) assigned a priority $\mathbf{pr}(J_i) < \mathbf{pr}(J_k)$ executes for as much as $c_i(\chi_k)$ units. Under our assumption that all the jobs are released at time-instant t (and the processor is therefore not idled while there is work remaining to be executed), this is equivalent to asserting that

$$c_k(\chi_k) + \sum_{J_i \,:\, \mathbf{pr}(J_i) < \mathbf{pr}(J_k)} c_i(\chi_k) \leq (d_k - t)$$
$$\equiv \sum_{J_i \,:\, \mathbf{pr}(J_i) \leq \mathbf{pr}(J_k)} c_i(\chi_k) \leq (d_k - t)$$

which is as claimed by this lemma. ■

3 Scheduling MC sporadic task systems

Prior research (described in Section 2.1 above) has shown that any MC instance I satisfying Condition 1 is schedulable using OCBP. Our objective here is to apply this result to obtain an algorithm for scheduling MC sporadic task systems.

Suppose that we had a sporadic task system τ satisfying the condition

$$\ell_{\rm LO}(\tau)^2 + \ell_{\rm HI}(\tau) \leq 1\;. \qquad (3)$$

It follows from the definition of $\ell_{\rm LO}$ and $\ell_{\rm HI}$ for sporadic task systems, that any instance I that is generated by τ satisfies Condition 1 as well, and is therefore schedulable using OCBP. It therefore appears trivial at first glance to extend the results of Section 2.1 to obtain an algorithm for scheduling an MC sporadic task system τ: during any given run of τ, simply apply OCBP to the instance I generated during that run.

Unfortunately, this argument does not quite work: There are (at least) two problems with directly applying the results in Section 2.1 to systems of sporadic tasks:

1. The OCBP algorithm has two phases: an off-line phase during which priorities are computed for all the jobs, followed by the run-time phase which deploys priority-based dispatching using the priorities assigned during the off-line phase. But since any instance generated by a sporadic task system may contain infinitely many jobs, the procedure (described in Section 2.1) for computing all the priorities prior to run-time is not guaranteed to terminate.
2. The algorithm for determining OCBP priorities requires the *complete* specification of all the jobs in the instance. However, under the (reasonable) assumption that our run-time scheduling algorithm is not clairvoyant, we do not know this information beforehand for sporadic task systems: although we may know a lower bound on the release times of jobs, a job's exact release time only becomes known when it is actually released.

We will deal with the first of these problems –potentially infinitely many jobs– by only assigning priorities, at each instant in time, to those jobs that arrive during the current busy interval, where the *busy interval* refers to a maximal continuous interval of time during which the processor is not idled. (This is reasonable: since OCBP scheduling never idles the processor while there are jobs awaiting execution, scheduling decisions made within a particular busy interval are not impacted by the priorities assigned to jobs arriving outside that busy interval.) For any sporadic task system τ with both $\ell_{\rm LO}(\tau)$ and $\ell_{\rm HI}(\tau)$ strictly less than one, prior techniques from real-time scheduling theory can be applied to bound the maximum length of the longest busy interval, and thereby determine the largest collection of jobs that can possibly execute before the processor is idled. This is done by a straightforward application of previously-proposed techniques, that we briefly describe in Section 3.2.

To deal with the second problem –job release times (and hence deadlines) not known in advance, we will assign priorities under the assumption that all jobs in the current busy interval are released as soon as legally permitted to do so under the constraints of the sporadic task model[4]. During run-time, we will monitor the actual job-release times; as long as they conform to the ones we had used in assigning priorities, we need do nothing. When they do *not* so conform, we will, under some circumstances, need to re-compute the priorities assigned to some of the jobs: the exact details are provided in Section 3.1 below.

3.1 Computing OCBP priorities

Suppose that a busy interval begins at some instant, designated t_o, during run-time. That is, the processor is idle

[4]We note that such an instance of jobs generated by any τ satisfying Condition 3 can indeed be assigned OCBP priorities by the results in Section 2.1, since this instance is one of the instances of jobs that can ge generated by τ and consequently satisfies Condition 1.

immediately prior to t_o, and some job arrives at t_o. At this instant we will assign priorities to all the jobs that could possibly be scheduled during the busy interval beginning at t_o (Section 3.2 explains how this collection of jobs is determined) if each such job is released as soon as it is legally permitted to do so. For the purposes of assigning the priorities, we will assume that all these jobs are "early-released". We illustrate this step by an example.

Example 3 Suppose that we had determined that a particular task τ_i could have 3 jobs in the longest busy interval, and that τ_i is eligible to release a job at t_o (i.e., no job of τ_i has been released over $(t_o - T_i, t_o]$). At the earliest, therefore, these 3 jobs could be released at times t_o, $t_o + T_i$, and $t_o + 2T_i$ respectively, and would have deadlines at $t_o + D_i$, $t_o + T_i + D_i$, and $t_o + 2T_i + D_i$ respectively. We will, for the purposes of priority assignment, consider that these jobs all arrive at t_o, and have deadlines at $t_o + D_i$, $t_o + T_i + D_i$, and $t_o + 2T_i + D_i$ respectively. ■

Let $\{J_i\}_{i=1}^{n}$ denote all these jobs at t_o; $J_i \stackrel{\text{def}}{=} (t_o, \chi_i, c_i(\text{LO}), c_i(\text{HI}), d_i)$.

3.1.1 Priority assignment

We *assign priorities* to the jobs $\{J_i\}_{i=1}^{n}$ according to the OCBP priority assignment scheme. Let us denote this priority assignment as the bijective function $\pi : \{J_i\}_{i=1}^{n} \longrightarrow \{1, 2, \ldots, n\}$. Condition 4 immediately follows, from Lemma 1

$$\forall\, k \;::\; \sum_{J_i \,:\, \pi(J_i) \leq \pi(J_k)} c_i(\chi_k) \leq (d_k - t_o) \tag{4}$$

3.1.2 Run-time scheduling

We now *dispatch* jobs according to these priorities: at each instant, the job J_i with the smallest value of $\pi(J_i)$ that has been released but not yet signalled completion is selected for execution. This continues until one of the following events has occurred:

E-1. Some job J_i executes for more than $c_i(\text{LO})$ without signalling that it has completed execution. This implies that the system is now in HI-criticality mode, and LO-criticality jobs are no longer required to complete by their deadlines. *We may therefore discard all* LO*-criticality jobs.* It follows from the correctness of the OCBP priority assignment at t_o that all HI-criticality jobs that will arrive during the current busy interval are guaranteed to complete by their deadlines.

E-2. Under our priority-based scheduling model, the processor is idled at some time-instant t only if all jobs that had arrived prior to t have completed execution by time-instant t. If this happens, *the current busy interval has ended*, and priorities that were assigned at t_o to jobs that ended up not arriving during this busy interval are "canceled." We await the release of some job, which will signal the start of a new busy interval — at that time, we will recompute the priorities of all jobs that could possibly be scheduled during that busy interval.

E-3. The execution of some lesser-priority job J_x is preempted due to the release of some greater-priority job J_y (i.e., $\pi(J_y) < \pi(J_x)$), say at time-instant t_1. *We must recompute the priority function π at this point in time*. This is dealt with below, in the remainder of this section. We will formally prove that no job that had been assigned a priority below J_x's need have its priority changed (i.e., all jobs J_z with $\pi(J_z) \geq \pi(J_x)$ retain their priorities), but the remaining jobs may need to have new priorities computed.

Once these new priorities are computed, we set $t_o \leftarrow t_1$ and resume run-time dispatching using these newly-computed priorities, as described in this section – Section 3.1.2.

§**E-3: Recomputing priorities.** For each i, let Δ_i denote the amount of execution that job J_i has received over $[t_o, t_1)$. Let $c_i'(\text{LO}) \stackrel{\text{def}}{=} c_i(\text{LO}) - \Delta_i$, and $c_i'(\text{HI}) \stackrel{\text{def}}{=} c_i(\text{HI}) - \Delta_i$; these denote the remaining WCET estimates for J_i. Equivalently, we can think of the workload remaining to be executed during the current busy interval as the instance

$$\left\{ J_i' \stackrel{\text{def}}{=} (t_1, \chi_i, c_i'(\text{LO}), c_i'(\text{HI}), d_i) \right\}_{i=1}^{n}. \tag{5}$$

Since we have been dispatching according to the priorities $\pi(J_i)$, we can derive some important facts about the Δ_i values for jobs according to the priorities that they have been assigned. This is done in Lemmas 2 and 3 below.

Lemma 2 *No job J_k with lesser priority than J_x (i.e., with $\pi(J_k) > \pi(J_x)$) has executed over $[t_o, t_1)$. That is, $\Delta_k = 0$ for all such jobs.*

Proof: Suppose that jobs with lesser priority had executed during $[t_o, t_1)$, and consider the job of least priority (the J_k with largest $\pi(J_k)$) to have done so. This job must have executed to completion without having being preempted; else, its preemption would have triggered the re-computation of priorities (according to E-3 above) prior to time-instant t_1. But the instant at which it completed execution without being preempted is an idle instant[5] in the schedule, which, according to E-2 above, signals the end of the busy interval that began at t_o. This contradicts the assumption that t_o and t_1 are in the same busy interval. ■

Lemma 3 *Each job J_k of greater priority than J_x (i.e., with $\pi(J_k) < \pi(J_x)$ has either completed execution over $[t_o, t_1)$, or has not yet been released prior to time-instant t_1.*

[5]There is a technical subtlety here: it is possible that there is a job J_ℓ of lower priority than J_k awaiting execution, but a job J_p with priority greater than J_k arrives at the instant that J_k completes execution. Under these circumstances, the instant at which J_k completes execution would not constitute the end of the busy interval. We bypass this problem by mandating that in such circumstances, J_ℓ executes momentarily before J_p is released, but is then preempted by J_p's release. Such a preemption would have triggered the re-computation of priorities (according to E-3 above) prior to time-instant t_1.

Proof: This follows from the properties of priority-based dispatch: once a job begins execution, no lower-priority job may execute until that job has completed execution. Since J_x is executing at t_1, it must be the case that any job J_k with $\pi(J_k) < \pi(J_x)$ has either not arrived, or has arrived and completed execution. ■

Our objective now is to construct a new priority assignment $\pi' : \{J'_i\}_{i=1}^{n} \longrightarrow \{1, 2, \ldots, n\}$ which will satisfy Lemma 1 at time-instant t_1 (and will therefore constitute a valid OCBP priority assignment for $\{J'_i\}_{i=1}^{n}$ at time-instant t_1). Lemma 4 helps us do so, by asserting, in essence, that jobs J_k that were assigned less priority than J_x by π (i.e., $\pi(J_k) > \pi(J_x)$) can have their priorities remain unchanged in π':

Lemma 4 *For each J_k with $\pi(J_k) \geq \pi(J_x)$,*

$$\sum_{J_i \,:\, \pi(J_i) \leq \pi(J_k)} c'_i(\chi_k) \leq (d_k - t_1). \tag{6}$$

Proof: For any such J_k, Equation 2 asserts that

$$\begin{aligned}
&\sum_{J_i \,:\, \pi(J_i) \leq \pi(J_k)} c_i(\chi_k) \leq (d_k - t_o) \\
\equiv &\sum_{J_i \,:\, \pi(J_i) \leq \pi(J_k)} \Big(c'_i(\chi_k) + \Delta_i\Big) \leq (d_k - t_o) \\
\equiv &\Big(\sum_{J_i \,:\, \pi(J_i) \leq \pi(J_k)} c'_i(\chi_k) + \sum_{J_i \,:\, \pi(J_i) \leq \pi(J_k)} \Delta_i\Big) \leq (d_k - t_o) \\
\equiv &\text{ By Lemma 2, all the execution over } [t_o, t_1) \text{ is of such jobs} \\
&\sum_{J_i \,:\, \pi(J_i) \leq \pi(J_k)} c'_i(\chi_k) + (t_1 - t_o) \leq (d_k - t_o) \\
\equiv &\sum_{J_i \,:\, \pi(J_i) \leq \pi(J_k)} c'_i(\chi_k) \leq (d_k - t_1)
\end{aligned}$$

and the lemma is proved. ■

Recall that our goal is to construct priority assignment $\pi' : \{J'_i\}_{i=1}^{n} \longrightarrow \{1, 2, \ldots, n\}$ to satisfy Lemma 1 at time-instant t_1. By Lemma 4, any priority assignment π' with $\pi'(J_k) \equiv \pi(J_k)$ for all J_k that have $\pi(J_k) \geq \pi(J_x)$ will satisfy Condition 2 for all J_k that have $\pi(J_k) \geq \pi(J_x)$; in other words, *all jobs that were assigned lesser priority than J_x by priority assignment π can retain their original priorities in priority assignment π'.*

What about jobs that were prioritized over J_x in π (i.e., jobs J_k with $\pi(J_k) < \pi(J_x)$)? By Lemma 3, each such job has either completed execution by time-instant t_1, or has not yet been released. I.e., *all such (non-completed) jobs have release time in the future, and hence taken together constitute a legal instance that could be generated by task system τ.* Due to the assumption that τ satisfies Condition 3, all jobs in this instance can therefore be assigned OCBP priorities. We will *recompute* priorities for just these jobs using the OCBP priority assignment algorithm; with all these jobs prioritized over the jobs J_k that satisfied $\pi(J_k) \geq \pi(J_x)$. This is formally stated in the following lemma:

Lemma 5 *Suppose that $(n - n')$ jobs have signaled completion over $[t_o, t_1)$: by Lemma 3, these are all jobs J_k with $\pi(J_k) < \pi(J_x)$. Let $\{J'_i\}_{i=1}^{n'}$ denote the remaining jobs. The priority assignment*

$$\pi' : \{J'_i\}_{i=1}^{n'} \longrightarrow \{1, 2, \ldots, n'\},$$

which is obtained from π as follows:

1. *All jobs J_k satisfying $\pi(J_k) < \pi(J_x)$ that have not yet completed execution have OCBP priorities $\pi'(J_k)$ recomputed, once again under the "early release" assumption that they are all released at time-instant t_1;*
2. *For each job J_k satisfying $\pi(J_k) \geq \pi(J_x)$, $\pi'(J_k) \leftarrow \pi(J_k) - (n - n')$*

is an OCBP priority assignment for $\{J'_i\}_{i=1}^{n'}$ at time-instant t_1.

Proof: The jobs that are assigned OCBP priorities in step 1 above have the greatest priorities; clearly, the priority assignment to these jobs trivially constitute an OCBP priority assignment. By Lemma 1, each such J_k satisfies Condition 2 under priority assignment π'.

For the remaining jobs — the J_k's that had $\pi(J_k) \geq \pi(J_x)$ and retained their relative priorities under priority assignment π' (i.e., $\pi'(J_k) \leftarrow \pi(J_k) - (n - n')$), Lemma 4 asserts that each such job also satisfies Condition 2 under priority assignment π'. It therefore follows from Lemma 1 that π' constitutes a valid OCBP priority assignment at time-instant t_1. ■

Once the OCBP priority assignment π' is computed, we can repeat the entire argument in this section — Section 3.1.2, with t_o set to t_1; $n \leftarrow n'$, each J_i set to the corresponding J'_i, and the priority assignment π taking on the value of the priority assignment π'. That is, we continue priority-based scheduling using their newly-computed priorities, awaiting the occurrence of one of the three events E-1, E-2, or E-3.

3.2 Busy interval size bound

Given the specifications of a MC sporadic task system τ we can bound the longest busy interval of τ in the following manner.

According to our run-time dispatching algorithm (Section 3.1.2, E-1) no LO-criticality job is executed once any job executes for more than its LO-criticality WCET. Let us therefore consider the longest busy interval as being comprised of two parts: (i) from the beginning of the busy interval up to the instant (if any) at which some job executes for more than its LO-criticality WCET, and (ii) from that instant to the end of the busy interval. Without loss of generality,

we assume that the busy interval starts at time-instant zero, some job executes for more than its LO-criticality WCET at time-instant x_1, and the busy interval ends at time-instant $x_1 + x_2$.

Let $D_{\max}$ denote the largest deadline of any task in τ: $D_{\max} = \max_{\tau_i \in \tau}\{D_i\}$. All jobs executed over $[0, x_1)$ have their release times and deadlines within the interval $[0, x_1 + D_{\max})$; hence

$$\begin{aligned} & x_1 \leq \ell_{\text{LO}}(\tau)(D_{\max} + x_1) \\ \Leftrightarrow \quad & x_1(1 - \ell_{\text{LO}}(\tau)) \leq \ell_{\text{LO}}(\tau) \times D_{\max} \\ \Leftrightarrow \quad & x_1 \leq \frac{\ell_{\text{LO}}(\tau)}{1 - \ell_{\text{LO}}(\tau)} \times D_{\max} \end{aligned}$$

Since all jobs executing during $[x_1, x_1 + x_2)$ have their release times and deadlines within the interval $[0, x_1 + x_2 + D_{\max})$, it must be the case that

$$\begin{aligned} & x_2 \leq \ell_{\text{HI}}(\tau)(D_{\max} + x_1 + x_2) \\ \Leftrightarrow \quad & x_2(1 - \ell_{\text{HI}}(\tau)) \leq \ell_{\text{HI}}(\tau) \times (D_{\max} + x_1) \\ \Leftrightarrow \quad & x_2 \leq \frac{\ell_{\text{HI}}(\tau)}{1 - \ell_{\text{HI}}(\tau)} \times (D_{\max} + x_1) \\ \Leftrightarrow \quad & x_2 \leq \frac{\ell_{\text{HI}}(\tau)}{1 - \ell_{\text{HI}}(\tau)} \times \left(D_{\max} + \frac{\ell_{\text{LO}}(\tau)}{1 - \ell_{\text{LO}}(\tau)} D_{\max}\right) \\ \Leftrightarrow \quad & x_2 \leq \frac{\ell_{\text{HI}}(\tau)}{(1 - \ell_{\text{LO}}(\tau))(1 - \ell_{\text{HI}}(\tau))} \times D_{\max} \end{aligned}$$

The length of the longest busy interval is then bounded from above[6] by $x_1 + x_2$. Under the assumption that $\ell_{\text{LO}}(\tau)$ and $\ell_{\text{HI}}(\tau)$ are both bounded from above by a constant strictly less than one, this is easily seen to be pseudo-polynomial in the representation of τ. Once the length of the longest busy period has been bounded as above, it is straightforward to bound which jobs could have arrived within this interval — there will be at most pseudo-polynomially many such jobs. These are the jobs that have OCBP priorities assigned to them at any given time during the execution of our scheduling algorithm on MC sporadic task system τ.

3.3 Computational complexity

We consider the algorithm we have presented in Sections 3.1 and 3.2 very significant from a theoretical perspective, since it establishes that previously-known schedulability bounds for MC scheduling of instances comprised of independent jobs extend to MC systems of sporadic tasks as well. But how *practical* is this algorithm? – is it likely to be useful in practice, in implementing actual MC systems that are subject to certification? In order to address this question, we must determine whether the steps of the algorithm can be performed reasonably efficiently (particularly the steps that are performed during run-time).

For an instance comprised of jobs that all arrive at the same instant (as in Lemma 1) OCBP priorities satisfying Lemma 1 can be determined in time polynomial in the number of jobs, as follows. First, separately *sort* the HI-criticality jobs and the LO-criticality jobs according to deadlines. Then during the recursive priority-assignment step of OCBP, we need *only consider the latest-deadline job of each criticality* that has not yet been assigned a priority as the potential lowest-priority job; this follows from the observation that jobs of the same criticality level may be assigned relative priorities in order of deadline.

As shown in Section 3.2 above, each busy interval for sporadic task system τ with both $\ell_{\text{LO}}(\tau)$ and $\ell_{\text{HI}}(\tau)$ bounded from above by a constant strictly less than one has at most pseudo-polynomially many jobs. Therefore, the priority-assignment at the beginning of each busy interval can be done in time pseudo-polynomial in the representation of the sporadic task system τ. Indeed, the initial assignment of priorities – assuming that each task generates a job at the same instant and subsequent jobs as soon as legal – can be pre-computed during system design time, and stored for use during run-time.

What about the re-computation of priorities that may be necessitated by the occurrence of event E-3 (Section 3.1.2): the currently-executing job is preempted by the release of a job that was assigned greater priority? In the worst case, our algorithm, as described in Section 3.1, does not rule out the possibility that the currently-executing job is one of the lowest-priority ones, in which case we would need to re-compute the priorities for almost *all* the jobs. In this case the time-complexity of doing such re-computation would be pseudo-polynomial in the representation of the task system (linear in the number of jobs that need to have their priorities recomputed). However, our preliminary experience with simple examples indicates that it is rare a job of very low priority is executing and subsequently preempted; it is a far more common case that the preempted job had relatively high priority, and we therefore actually need to recompute the priorities of just a few jobs. Intuitively, this makes sense: it is typically the case that jobs with release times and deadline very late in the busy interval are assigned the lowest priorities, and will therefore not have an opportunity to execute (and thereby be preempted) near the beginning of the busy interval. Hence although the number of jobs that need to have their priorities re-computed may be large following *some* preemptions, we expect that the average number of such recomputations per preemption is very

[6]We note that this is a pessimistic bound, in the sense that we are accounting for the possible execution, over the interval $[x_1, x_1 + x_2)$, of the entire HI-criticality WCET's of all HI-criticality jobs that may have release times and deadlines in the interval $[0, x_1 + x_2 + D_{\max})$. This is despite the fact that some of these jobs may have already completed execution prior to time-instant x_1. Our goal here is to show that such bounds exist, rather than to compute the tightest bound. Techniques such as the ones in, e.g., [15, 9] can be adapted to obtain bounds superior to the one we have derived here.

small. We are currently working on obtain a bound on this average number, and are exploring strategies for doing some of this computation off-line. We are also working on approximation algorithms that would significantly reduce the frequency of such re-computations, and the number of jobs that are subject to priority re-computation during each such re-computation step, by enforcing somewhat stricter constraints on the task system than Condition 3. That is, we are studying the tradeoff is making Condition 3 stricter, and thereby reducing the number of priority re-computations.

4 Related work

To our knowledge, the scheduling problem that arises from multiple certification requirements, at different criticality levels, was first identified and formalized by Vestal in [16], in the context of the fixed-priority preemptive uniprocessor scheduling of recurrent task systems.

Current practice in safety-critical embedded systems design for certifiability is centered around the technique of "space-time partitioning," as codified in, e.g., the ARINC-653 standard [1, 17]. This is one of several *reservation-based* approaches, in which a certain amount of the capacity of the shared platform is reserved for each application, that have been considered for designing certifiable mixed-criticality systems. It is known that reservation-based approaches tend to be pessimistic (in the sense of under-utilizing platform resource). Consider again the graph in Figure 1, plotting the bound on the LO-criticality load of an instance I as a function of its HI-criticality load $\ell_{\text{HI}}(I)$, in order for I to be successfully scheduled. As stated in Section 2.1, the curved line represents the OCBP schedulability bound: any instance that maps on to a point beneath this curve is guaranteed to be schedulable by OCBP. The straight line (equation $\ell_{\text{LO}}(I) + \ell_{\text{HI}}(I) = 1$) connecting the points $(1, 0)$ and $(0, 1)$ represents the schedulability condition for the space-time partitioning and other reservations-based approaches: since we must reserve a fraction $\ell_{\text{HI}}(I)$ of the processor for HI-criticality tasks in such an approach, that leaves a fraction $1-\ell_{\text{HI}}(I)$ of the processor capacity for accommodating additional LO-criticality jobs. It is evident from this plot that the schedulability region of space-time partitioning is strictly contained within the schedulability region of OCBP scheduling. Such pessimism is a consequence of the very principle of isolation between criticality levels upon which reservations-based design techniques are based: isolation rules out the possibility of reusing the resource capacity that must be assigned to high-criticality applications in order that they pass certification, but which they are unlikely to need in practice, to make performance guarantees to low-criticality applications.

Priority-based scheduling is the other technique commonly used by systems engineers in dealing with mixed criticalities. (OCBP scheduling is an example of a priority-based scheduling scheme.) Unless carefully designed, though, priority-based scheduling schemes can be even more pessimistic than reservations-based approaches. In a typical priority-based scheduling approach, for example, jobs belonging to higher-criticality applications are accorded greater priority in deciding which job to execute at each instant in time. It is not too difficult to construct simple examples in which such *criticality-monotonic* scheduling will perform arbitrarily poorly.

Some other research on mixed-criticality scheduling. Altough many other real-time scheduling papers deal with mixed-criticality systems, they do not really deal with scheduling for certification. De Niz et al. [7] deal with a different aspect of mixed-criticality systems from the one we focus on here, in that they do not directly address the certification issue. Nevertheless, [7] contains very interesting and novel ideas that merit mention. This work observes that the complete inter-criticality isolation offered by the reservations approach may cause *criticality inversion*: preventing a higher-criticality job from meeting its deadline while allowing lower-criticality jobs to complete. On the other hand, assigning priorities according to criticality may result in very poor processor utilization. An innovative *slack-aware* approach is proposed that builds atop priority-based scheduling (with priorities not necessarily assigned according to criticality), to allow for asymmetric protection of reservations thereby helping to lessen criticality inversion while retaining reasonable resource utilization.

Pellizzoni et al. [14], use a reservations-based approach to ensure strong isolation among sub-systems of different criticalities; this paper proposes innovative design and architectural techniques for preserving such isolation despite some necessary interaction (e.g., in the sharing of additional non-preemptable resources) between jobs of different criticalities. The focus is not on optimizing resource utilization, but on ensuring isolation; hence, this research does not attempt to avoid the criticality-inversion that is inherent to the reservations-based approach.

5 Conclusions

Due to the rapid increase in the complexity and diversity of functionalities that are performed by safety-critical embedded systems, the cost and complexity of obtaining certification for such systems is fast becoming a serious concern [3]. We believe that in mixed-criticality systems, these certification considerations give rise to fundamental new resource allocation and scheduling challenges which are not adequately addressed by conventional real-time scheduling theory. In prior work [4, 10], we have therefore proposed a job model that is particularly appropriate for representing

mixed-criticality workloads that can be modeled as collections of independent jobs, and have studied schedulability properties of this model. We had also derived an algorithm, called OCBP, for scheduling such mixed-criticality workloads. In this paper, extend our investigation to more general mixed-criticality workloads: those generated by recurrent processes. We have extended the sporadic tasks model, which is widely used for representing such processes in non-MC real-time systems, to be able to model recurrent MC workloads. We have extended the OCBP scheduling algorithm to schedule such MC real-time systems: such extension is quite non-trivial due to the property of sporadic task systems that release times of jobs are not known beforehand, but only become known at the instant that the job is actually released. Despite this added complexity, we have shown that the schedulability condition for MC workloads comprised of independent jobs generalizes to MC workloads comprised of sporadic tasks.

References

[1] ARINC. ARINC 653-1 Avionics application software standard interface, October 2003.

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

[3] J. Barhorst, T. Belote, P. Binns, J. Hoffman, J. Paunicka, P. Sarathy, J. S. P. Stanfill, D. Stuart, and R. Urzi. White paper: A research agenda for mixed-criticality systems, April 2009. Available at http://www.cse.wustl.edu/˜ cdgill/CPSWEEK09_MCAR.

[4] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*. IEEE, April 2010.

[5] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 182–190, Orlando, Florida, 1990. IEEE Computer Society Press.

[6] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Second edition, 2005.

[7] D. de Niz, K. Lakshmanan, and R. R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Proceedings of the Real-Time Systems Symposium*, pages 291–300, Washington, DC, 2009. IEEE Computer Society Press.

[8] M. Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[9] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Technical Report RR-2966, INRIA: Institut National de Recherche en Informatique et en Automatique, 1996.

[10] H. Li and S. Baruah. Load-based schedulability analysis of certifiable mixed-criticality systems. Available at http://www.cs.unc.edu/˜baruah/Pubs.shtml, 2010.

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

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

[13] A. K. 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.

[14] R. Pellizzoni, P. Meredith, M. Y. Nam, M. Sun, M. Caccamo, and L. Sha. Handling mixed criticality in SoC-based real-time embedded systems. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, Grenoble, France, 2009. IEEE Computer Society Press.

[15] I. Ripoll, A. Crespo, and A. K. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems: The International Journal of Time-Critical Computing*, 11:19–39, 1996.

[16] 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.

[17] J. Windsor and K. Hjortnaes. Time and space partitioning in spacecraft avionics. *Space Mission Challenges for Information Technology, IEEE International Conference on*, pages 13–20, 2009.