Timing Faults and Mixed Criticality Systems

Alan Burns[1] and Sanjoy Baruah[2]

[1] The University of York, UK
[2] The University of North Carolina, USA

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, or is certified to a lower level. The resulting mixed criticality system offers challenges both for static analysis and run-time monitoring. This paper is concerned with timing failures and how they can arise and be tolerated. The main causes of these errors are faults in the estimation of worst-case execution times (WCETs). For different levels of criticality, different forms of static analysis for WCET are employed. This give rise to a novel implementation scheme for the fixed priority uniprocessor scheduling of mixed criticality systems. The scheme requires that jobs have their execution times monitored (as is usually the case in high integrity systems). This results in higher levels of schedulability than previously published.

1 Introduction

An important class of faults in many computer-based systems is that which relates to the time at which key interactions with the system's environment take place. In general, faulty interactions occur either too early or too late. To avoid premature I/O is usually straightforward, most real-time programming languages and operating systems provide a *delay* primitive that allows the system to 'wait' until the environment is ready. More difficult, is to ensure that the system executes quickly enough to meet the *deadline* constraints there may be on its I/O operations.

Real-time software is usually constructed as a multi-tasking concurrent program where each *task* gives rise to a series of *jobs* on which deadline constraints are assigned. To verify that all deadlines are met for all valid executions of the program usually requires two forms of analysis. First, *timing analysis* determines the worst-case execution time (WCET) of each task (and hence of each job of each task). Second, *scheduling analysis* determines the worst-case completion time for each job when its execution competes with all the other tasks in the system. Knowing the worst-case completion time (also known as response time) enables a simple comparison with the job's deadline to be made. Scheduling analysis uses the resource management rules of the implementation to determine the order in which tasks make use of the available resources. In this paper

we restrict ourselves to single processor systems which are scheduled using the standard fixed priority scheme.

At run-time it is difficult to recover from a deadline miss; time cannot be reversed and hence all that can be done is to attempt to repair the damage caused by the timing fault. The cause of this failure is usually traced to a fault in the timing analysis – an underestimation of WCET; but it can also be attributable to faulty scheduling analysis or system overload. Here we focus on faulty timing analysis that could deliver optimistic WCET values. Most high integrity systems will include run-time monitoring facilities that will allow a WCET violation to be identified 'as it happens'. This error recognition can allow error recovery to be programmed with the result that all deadlines are met (even though the functional quality of the output may be downgraded).

One of the ways that scheduling analysis has been extended in recent years is the removal of the assumption that all tasks in the system have the same level of criticality or importance. Models have been produced that allow mixed criticality levels to co-exist on the same execution platform. For systems that contain components that have been given different criticality designations there are two, mainly distinct, issues: run-time robustness [1] and static verification [2, 3].

Run-time robustness is a form of fault tolerance that allows graceful degradation to occur in a manner that is mindful of criticality levels: informally speaking, in the event that all components cannot be serviced satisfactorily the goal is to ensure that lower-criticality components are denied their requested levels of service before higher-criticality components are.

Static verification of mixed-criticality systems is closely related to the problem of *certification* of safety-critical systems. The current trend towards integrating multiple functionalities on a common platform (for example in Integrated Modula Avionics, IMS, systems and in the Automotive Open System Architecture, Autosar) means that even in highly safety-critical systems, typically only a relatively small fraction of the overall system is actually of critical functionality and needs to be certified. In order to certify a system as being correct, the certification authority (CA) must make certain assumptions about the worst-case behaviour of the system during run-time. CA's tend to be very conservative, and hence it is often the case that the assumptions required by the CA are far more pessimistic than those the system designer would typically use during the system design process if certification was not required. However, 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 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 three jobs 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 worst-case execution time (WCET) not exceeding* 1. *Hence executing the jobs in earliest deadline first (EDF) order will ensure that all three complete by their deadlines.*
– *However, the CA requires the use of more pessimistic WCET estimates during the certification process, and allows for the possibility that jobs* J_2 *and* J_3 *may each need* 1.5 *time units of execution*[3].

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 a "correct" scheduling strategy[4] *for this system is as follows:*

– *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.*

The reader may verify that under this scheduling strategy, the system both passes certification, and *meets all deadlines when it behaves as expected to by the system designer.* □

Static verification is concerned with the necessary timing (WCET) analysis and scheduling analysis that will ensure that all tasks in a system are guaranteed (to meet their deadlines) in a manner commensurate with their criticality level. Robustness is a form of fault tolerance that will allow graceful degradation to occur that is again mindful of criticality levels. In this paper we focus on static verification.

Related work. In prior work [4, 5], 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

[3] The CA required method 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 executes for more than 1 time-unit.

[4] The notion of a correct scheduling strategy is formally defined later in this document.

systems are better modeled as collections of *recurrent tasks* that are specified using, e.g., the sporadic tasks model [6, 7]. 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 potentially unboundedly many jobs during any one run; and (ii) the collection of jobs generated during different runs of the system may be different. Vestal [2] initiated the study of certification-cognizant scheduling of such sporadic task systems, and proposed a static fixed-priority (FP) algorithm, based on a specialization of Audsley's priority-assignment technique [8], for assigning priorities optimally to the tasks in such a system. Some other works (e.g., [3, 9]) have considered algorithms that are not fixed-priority, for scheduling mixed-criticality systems in a certifiably correct manner.

2 System Model

A system is defined as a finite set of components $\mathcal{K}$. Each component has a defined level of criticality, L. Each component contains a finite set of tasks. Each task, τ_i, is defined by period, deadline, computation time and criticality level: (T_i, D_i, C_i, L_i). These parameters are however not independent, in particular the following relations are assumed to hold between L and the other parameters in any valid mixed criticality system:

- The worst-case computation time, C, will be derived by a process dictated by the criticality level. The higher the criticality level, the more conservative the verification process and hence the greater will be the value of C.
- The deadline of the task may also be a function of the criticality level. The higher the criticality level, the greater the need for the task to complete well before any safety-critical timing constraint and hence the smaller the value of D.
- Finally, though less likely, the period of a task could depend on criticality. The higher the criticality level, the tighter the level of control that may be needed and hence the smaller the value of T.

These relations are formalised with the following axioms: if a task, τ_i is moved to criticality level L_i^1 from criticality level L_i^2 then

$$L_i^1 > L_i^2 \Rightarrow C_i^1 \geq C_i^2$$
$$L_i^1 > L_i^2 \Rightarrow D_i^1 \leq D_i^2$$
$$L_i^1 > L_i^2 \Rightarrow T_i^1 \leq T_i^2$$

At run-time a task will have fixed values of T, D and L. Its actual computation time is however unknown; it is *not* directly a function of L. The code of the task will execute on the available hardware, and apart from catching and/or

dealing with overruns the task's actual criticality level will not influence the behaviour of the hardware. Rather the probability of failure (actual computation time being greater then C) will reduce for higher levels of L (due to C monotonically increasing with L).

In a mixed criticality system further information is needed in order to undertake schedulability analysis. Tasks can depend on other tasks with higher or lower levels of criticality. In general a task is now defined by: (T, D, $\boldsymbol{C}$, L), where $\boldsymbol{C}$ is a vector of values – one per criticality level, with the constraint:

$$C^{L1} > C^{L2} \Rightarrow L1 > L2$$

for any two criticality levels $L1$ and $L2$.

The general task τ_i with criticality level L_i will have one value from its $\boldsymbol{C_i}$ vector that defines its *representative* computation time. This is the value corresponding to L_i, ie. $C_i^{L_i}$. This will be given the normal symbol C_i.

In more detail, a task τ_i is characterized by

- A criticality L_i, which is a positive integer.
- A WCET function $C_i : \mathcal{N}^+ \rightarrow \mathcal{R}^+$; $C_i(\ell)$ denotes the WCET of each jobs of τ_i, estimated at a level of assurance consistent with criticality level ℓ. We assume that $C_i(\ell) \leq C_i(\ell + 1)$ for all ℓ; i.e., the WCET estimates are monotonically non-decreasing with increasing criticality level.
- A relative deadline parameter D_i.
- A period parameter T_i.

We assume that $C_i(L_i) \leq D_i$ for each τ_i. That is, we assume that each task's representative computation time is small enough to be able to complete prior to its deadline, if it were to execute in isolation – clearly, any task not satisfying this property cannot possibly be certified correct. We also restrict our attention here to task systems in which $D_i \leq T_i$ for each τ_i.

Definition 1 (Behaviours) During different runs, any given sporadic task system will, in general, exhibit different *behaviours*: different jobs may be released at different instants, and may have different actual execution times. We define the *criticality level of a behaviour* to be the smallest criticality level such that no job executed for more than its WCET at this criticality level.

As previously stated, two distinct issues have been addressed concerning the scheduling of mixed-criticality systems: *static verification*, and *run-time robustness*.

Static verification. From the perspective of static verification, the correctness criterion expected of an algorithm for scheduling mixed-criticality task systems is as follows: for each criticality level ℓ, *all jobs of all tasks with criticality* $\geq \ell$ *will complete by their deadlines in any criticality-*ℓ *behaviour*.

Run-time robustness. Static verification is concerned with *certification* – it requires that all deadlines of all tasks τ_i with $L_i \geq \ell$ are guaranteed to be met, provided that *no* job executes for more than its level-ℓ WCET. Run-time robustness, in addition, seeks to deal satisfactorily with *transient overloads* due either to errors in the control systems or to the environment behaving outside of the assumptions used in the analysis of the systems. Even in behaviours that have a high criticality level by Definition 1 above, it may be the case that all jobs executing beyond their WCET's at some lower criticality level did so only for a short duration of time (i.e., a transient overload can be thought to have occurred from the perspective of the lower criticality level). A robust scheduling algorithm would, informally speaking, be able to "recover" from the overload once it was over, and go back to meeting the deadlines of the lower-criticality jobs as well [10].

In this paper we consider only the issues surrounding static verification, and evaluate three possible priority assignment schemes:

- Partitioned Criticality (PC) – a standard scheme sometimes called *criticality monotonic priority assignment*
- Static Mixed Criticality (SMC) – a previously published scheme, reviewed in Section 3;
- Adaptive Mixed Criticality (AMC) – a novel scheme, introduced in Section 4.

In *Partitioned Criticality*, priorities are assigned according to criticality, so all jobs of criticality L1 have a higher priority than all jobs of criticality L2 if L1>L2. Within a criticality, priorities are assigned according to a standard optimal scheme such as deadline monotonic priority assignment[11] (for tasks with constrained deadlines, i.e., those that have relative deadline no greater than period: $D \leq T$). Each job is assumed to have an execution time no greater than its representative value. This partitioned approach has the advantage that a timing error in a low criticality job (i.e., executing for longer than its 'worst-case execution time' will not impact on any higher criticality jobs. No run-time monitoring is required.

However, the kinds of performance guarantees that can be made in scheduling mixed-criticality systems depend upon the forms of support provided by the run-time environment upon which the system is being implemented. An important form of platform support is the ability to *monitor* the execution of individual jobs, i.e., being able to determine how long a particular job has been executing. For instance, many safety critical systems that have replicated computing systems (*channels*– see, e.g., [12]) monitor execution times so that erroneous behaviour can be identified and the associated channel closed down (and possibly restarted). A strong case can be made for this ability to be part of the standard mechanisms for safety-critical applications.

Such functionality is already commonly available on many real-time platforms and is widely assumed in, for example, many implementations of servers (such as the Constant Bandwidth Server (CBS) [13] or their fixed priority counterparts [14–16]), or in real-time "open" environments that support the policing

or budget-enforcement of individual jobs or of collections of jobs in order to ensure that they do not exceed their execution allowances [17].

The other two priority assignment schemes (SMC and AMC) utilise forms of execution time monitoring and allow the priorities of different criticality jobs to be interleaved. This improves schedulability. The *Static* scheme (SMC) does not allow a job to execute for more than its representative execution time, C_i – this is similar to the approach introduced by Vestal [2]. The *Adaptive* scheme (AMC) goes further: for each criticality level L, it does not allow jobs of criticality L to execute if any job (of equal or higher criticality) executes for more than $C(L)$. The main contribution of this paper is the introduction and evaluation of this Adaptive Mixed Criticality scheme. The static approach is reviewed in Section 3; the adaptive scheme is introduced in Section 4.

For ease of presentation, in this paper we will restrict our attention to *dual-criticality* systems: systems in which there are only two criticality levels: HI (high) and LO (low), with HI>LO.

Each dual-criticality job is thus 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, and $d_i \in R^+$ the deadline. We require 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).

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; we will assume in this paper that $D_k \leq T_k$. 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.

3 Static Mixed Criticality - SMC

With this scheme all jobs can execute up to their representative execution time C_i but are prevented from executing further – they are either aborted or, if error recovery is desirable, suspended until it is safe for them to execute again. Means of programming recovery are explained elsewhere [10].

We now describe (a slight variation) of Vestal's algorithm [2] for assigning priorities to sporadic task systems, scheduled by the SMC scheme, to yield a certifiably correct scheduling strategy.

In essence, SMC assigns priorities to the tasks in the MC sporadic task system by applying the Audsley strategy [8]. That is, it first identifies some task which may be assigned the lowest priority; having done so, this task is removed from the task system and a priority assignment is recursively obtained for the remaining tasks.

To completely specify this strategy, it remains to specify how it is determined whether a particular job may be assigned lowest priority. Suppose we are seeking to determine whether τ_i, with criticality level $\chi_i \in \{\text{LO}, \text{HI}\}$, can be the lowest-priority task. Mixed-criticality semantics specify that τ_i should meet its deadline, provided all jobs of all tasks in the system execute for at most their χ_i-WCET. If each job of each task τ_j may execute for as much as its χ_i-WCET $C_j(\chi_i)$, it follows from response-time analysis (RTA) that the maximum response time of τ_i's jobs is given by the smallest fixed-point solution of the following recurrence:

$$t = \sum_{\forall\ j} \left\lceil \frac{t}{T_j} \right\rceil C_j(\chi_i) \tag{1}$$

This recurrence can be solved using standard techniques from RTA; if the solution is no larger than D_i then τ_i can indeed be assigned lowest priority.

Example 2 Consider an example task system τ comprised of three tasks, as follows:

τ_i	χ_i	$C_i(\text{LO})$	$C_i(\text{HI})$	D_i	T_i
τ_1	LO	1	-	2	2
τ_2	HI	1	2	10	10
τ_3	HI	20	20	100	100

It is evident (since τ_3's WCET, at both criticality levels, exceeds both D_1 and D_2), that neither τ_1 nor τ_2 can possibly be assigned lowest priority. We seek to determine whether τ_3 can be assigned lowest priority.

Based on the arguments above, we see that τ_3 may be assigned lowest priority if it would meet its deadline as the lowest-priority job. According to RTA, the worst-case response time of any of τ_3's jobs is equal to the smallest positive solution to the following recurrence is ≤ 100:

$$t = \left\lceil \frac{t}{2} \right\rceil \cdot 1 + \left\lceil \frac{t}{10} \right\rceil \cdot 2 + \left\lceil \frac{t}{100} \right\rceil \cdot 20$$

It is easily verified that 68 is a solution to this recurrence, since for $t = 68$ the RHS evaluates to $(34 + 7 \cdot 2 + 1 \cdot 20) = 68$, and hence the smallest positive solution is ≤ 68 (in fact, the smallest positive solution is 68). Since 68 is no larger than τ_3's deadline, we conclude that task τ_3 may indeed be assigned lowest priority.

However, the reader may verify that if $C_2(\text{HI})$ had been equal to 5 then the response-time of τ_3 would not be smaller than τ_3's deadline of 100, and we would therefore fail to assign priorities to this particular task system. We shall return to this example later. □

4 Adaptive Mixed Criticality - AMC

Upon a platform that can monitor how much individual jobs have been executing, the following adaptive run-time scheduling algorithm exploits this ability to obtain enhanced performance over the static scheme. The algorithm is provided with a mixed-criticality sporadic task system along with an assignment of unique distinct priorities to the tasks in the system. Dispatching of jobs for execution occurs according to Algorithm DISPATCH, which is given in Figure 1.

Algorithm DISPATCH:

1. There is a *criticality level indicator* Γ, initialized to LO.
2. While ($\Gamma \equiv$ LO), at each instant the waiting job generated by the task with highest priority is selected for execution.
3. If the currently-executing job executes for more than its LO-criticality WCET without signalling completion[5], then $\Gamma \leftarrow$ HI.
4. Once ($\Gamma \equiv$ HI), jobs with criticality level $\equiv$ LO will *not* receive any further execution. Henceforth, therefore, at each instant the waiting job generated by the HI-criticality task with highest priority is selected for execution.
5. An additional rule could specify the circumstances when Γ gets reset to LO. This could happen, for instance, if no HI-criticality jobs are active at some instant in time. (We will not discuss the process of resetting $\Gamma \leftarrow$ LO any further in this document, since this is not relevant to the certification process – LO-criticality certification assumes that the system *never* exhibits any HI-criticality behaviour, while HI-criticality certification is not interested in the behaviour of the LO-criticality tasks.)

Fig. 1. Run-time dispatching algorithm used in AMC

4.1 Criticality-Aware Assignment of Priorities (CAAP)

We now describe a priority-assignment scheme called CAAP (for *Criticality-Aware Assignment of Priorities*), that may be used to assign priorities for run-time dispatching according to Algorithm DISPATCH of Figure 1. As in the SMC algorithm CAAP, too, assigns priorities according to the Audsley strategy [8]. That is, CAAP first seeks to identify some task which may be assigned the lowest priority; having done so, this task is removed and CAAP (recursively) seeks to obtain a priority assignment for the remaining tasks.

We introduce some notation: Let L_{LO} (L_{HI}, respectively) denote an *upper bound* on the length of the longest busy interval when any LO-criticality (HI-criticality, resp.) behaviour of τ–i.e., any behaviour in which no job executes for more than its LO-criticality (HI-criticality, resp.) WCET. It is evident that any LO-criticality task τ_i satisfying $D_i \geq L_{\text{LO}}$ may be assigned lowest priority: since no LO-criticality behaviour can span the entire interval between the release of

any job of τ_i and its deadline, no such job will miss its deadline if τ_i is assigned lowest priority. Similarly, any HI-criticality task τ_i satisfying $D_i \geq L_{\text{HI}}$ may be assigned lowest priority.

Based on widely-known results from Response-Time Analysis (RTA) [18, 19] it is straightforward to observe that L_{LO} can be set equal to the smallest positive value of t satisfying

$$t = \sum_{\forall j} \left\lceil \frac{t}{T_j} \right\rceil C_j(\text{LO}) \tag{2}$$

We seek to determine L_{HI} next. Without loss of generality, let us suppose that the longest busy interval in any HI-criticality behaviour occurs on a sequence of jobs of τ in which the first job arrives at time zero. Let t_1 denote the time-instant at which the criticality level indicator Γ sees its value changed from LO to HI. (That is, t_1 is the first instant at which some job does not signal completion despite having executed for its LO-criticality WCET.) According to rule R4 of Algorithm DISPATCH no job of any LO-criticality task will receive any execution after time-instant t_k. Hence for any τ_j with $\chi_j = \text{LO}$, at most $\lceil t_1/T_j \rceil$ jobs of τ_j may execute in this longest busy interval.

Since L_{LO} is, by definition, an upper bound on the length of the largest busy interval in any LO-criticality behaviour, it follows that $t_1 \leq L_{\text{LO}}$. Hence the total amount of execution by jobs of LO-criticality tasks in this longest busy interval of any HI-criticality behaviour is bounded from above by $\sum_{j:\chi_j=\text{LO}} \big(\lceil L_{\text{LO}}/T_j \rceil \cdot C_j(\text{LO}) \big)$. And for any value of t, the total amount of execution of HI-criticality jobs over the interval $[0, t)$ in any HI-criticality behaviour is bounded from above by $\sum_{j:\chi_j=\text{HI}} \big(\lceil t/T_j \rceil \cdot C_j(\text{HI}) \big)$. It therefore follows that L_{HI}, an upper bound on the length of the longest HI-criticality busy interval, can be set equal to the smallest value of t that is $\geq L(\text{LO})$, satisfying

$$t = \sum_{j:\chi_j=\text{LO}} \left\lceil \frac{L_{\text{LO}}}{T_j} \right\rceil C_j(\text{LO}) + \sum_{j:\chi_j=\text{HI}} \left\lceil \frac{t}{T_j} \right\rceil C_j(\text{HI}) \tag{3}$$

Plugging the value for L_{LO} obtained by solving Equation 2 into recurrence Equation 3, we can determine the value of L_{HI} by using standard techniques for determining fixed-points.

τ_i	χ_i	$C_i(\text{LO})$	$C_i(\text{HI})$	D_i	T_i
τ_1	LO	1	-	2	2
τ_2	HI	1	5	10	10
τ_3	HI	20	20	100	100

Table 1. Task system for Example 3

Example 3 Consider again the task system of Example 2, with task τ_2's HI-criticality WCET increased from 2 to 5. (This modified task system is reproduced in Table 1.) As seen in Example 2, SMC fails to assign priorities to this task system; let us now examine how CAAP would fare.

It is evident (since τ_3's WCET, at both criticality levels, exceeds both D_1 and D_2), that neither τ_1 nor τ_2 can possibly be assigned lowest priority. We seek to determine whether τ_3 can be assigned lowest priority.

- Let us first compute L_{LO}, the upper bound on the length of the longest busy interval for any LO-criticality behaviour. According to Equation 2, L_{LO} is equal to the smallest positive solution to the following recurrence:

$$t = \left\lceil \frac{t}{2} \right\rceil \cdot 1 + \left\lceil \frac{t}{10} \right\rceil \cdot 1 + \left\lceil \frac{t}{100} \right\rceil \cdot 20$$

  It may be verified that this smallest positive solution is equal to 50; hence $L_{\text{LO}} \leftarrow 50$.
- Next, let us compute L_{HI}, the upper bound on the length of the longest busy interval for any HI-criticality behaviour. According to Equation 3, L_{HI} is equal to the smallest positive solution to the following recurrence:

$$t = \left\lceil \frac{50}{2} \right\rceil \cdot 1 + \left\lceil \frac{t}{10} \right\rceil \cdot 5 + \left\lceil \frac{t}{100} \right\rceil \cdot 20$$

  (here, we are using the fact, derived above, that $L_{\text{LO}} = 50$.) It is easily verified that this has a fixed point at $t = 90$, meaning that $L_{\text{HI}} \leftarrow 90$.

We had observed that τ_i may be assigned lowest priority if $L_{\chi_i} \leq D_i$. Since $L_{\chi_3} = L_{\text{HI}} = 90$ while $D_3 = 100$, we conclude that task τ_3 may indeed be assigned the lowest priority.

We had noted (at the end of Example 2) that SMC fails to assign priorities to this example task system. This task system thus bears witness to the fact that there are task systems that AMC with CAAP can schedule in a certifiably correct manner, that SMC cannot.

□

Run-time complexity. Notice that the values of L_{LO} and L_{HI}, as defined by Equations 2 and 3, depend only upon the tasks in the task system τ, and that these values are monotonically non-increasing with decreasing τ (i.e., the values of L_{LO} and L_{HI} for a given τ will not be larger than for any subset of the tasks in τ). Hence for each of $\ell \in \{\text{LO}, \text{HI}\}$, the task of criticality level ℓ for which it is "most likely" that it can be assigned the lowest priority, is the one with the largest relative deadline. This observation is formalized into the following fact:

Fact 1 *Tasks with the same criticality level may be assigned priority by CAAP in deadline-monotonic order.*

Hence in seeking to determine whether some task may be assigned lowest lowest priority, there are only two potential candidates to consider: the LO-criticality task with the largest relative deadline and the HI-criticality task with the largest relative deadline. This implies that we need solve Recurrences 2 and 3 a total of at most $\mathcal{O}(|\tau|)$ times in order to assign priorities to the tasks in the MC sporadic task system τ (where $|\tau|$ denotes the number of tasks in τ).

5 Analysis of CAAP

Demand bound function and load. For any "regular" (i.e., non-MC) sporadic task system, and for any non-negative real-number t, the *demand bound function* [7] of the task system for an interval-length t, denoted $\text{DBF}(t)$, represents the maximum cumulative execution requirement by jobs of the task system that can both arrive in, and have their deadlines within, any contiguous interval of length t. The *load* λ of the task system is defined as $\max_{t>0} \text{DBF}(t)/t$.

We will extend the definitions of demand bound function and load to mixed-criticality task systems, by defining separate LO-criticality and HI-criticality versions of these metrics for each MC system. Let τ denote such a task system. Informally speaking, $\text{DBF}_{\text{LO}}(t)$ and λ_{LO} will denote the DBF and load parameters for the regular sporadic task system that, in the expectation of the system designer, represents the behaviour of the system. $\text{DBF}_{\text{HI}}(t)$ and λ_{HI}, on the other hand, will denote the DBF and load parameters for the regular sporadic task system that the certification authorities believe represents the behaviour of the part of the system that they seek to certify. More formally, $\text{DBF}_{\text{LO}}(t)$ denotes the demand bound function for the "regular" sporadic task system

$$\Big\{ (C_i(\text{LO}), D_i, T_i) \mid \tau_i \in \tau \Big\} ;$$

and λ_{LO} is defined as follows:

$$\lambda_{\text{LO}} \stackrel{\text{def}}{=} \max_{t>0} \Big(\text{DBF}_{\text{LO}}(t)/t \Big) . \tag{4}$$

$\text{DBF}_{\text{HI}}(t)$ and λ_{HI} are defined in a similar vein. $\text{DBF}_{\text{HI}}(t)$ is the demand bound function of the regular sporadic task system

$$\Big\{ (C_i(\text{HI}), D_i, T_i) \mid \tau_i \in \tau \ \wedge \ \chi_i = \text{HI} \Big\} ;$$

and

$$\lambda_{\text{HI}} \stackrel{\text{def}}{=} \max_{t>0} \Big(\text{DBF}_{\text{HI}}(t)/t \Big) . \tag{5}$$

Observe that in order for MC task system τ to be scheduled correctly, it is necessary that neither λ_{LO} or λ_{HI} exceeds 1; if either were > 1 it would not be possible to guarantee that all behaviours of that particular criticality can be scheduled by even a clairvoyant scheduling algorithm. More generally, if we had a *speed-s* processor – a processor that completes s units of execution per time

unit – then ($\lambda_{\text{LO}} \leq s \bigwedge \lambda_{\text{HI}} \leq s$) is a necessary (albeit not sufficient) condition for τ to be schedulable on this processor in a certifiably correct manner.

In Lemma 1 below, we prove that from the perspective of schedulability by Algorithm DISPATCH (described in Figure 1), the CAAP priority assignment scheme strictly dominates deadline-monotonic (DM) priority assignment: any task system schedulable in a certifiably correct manner by DM is also scheduled in a certifiably correct manner by CAAP, whereas the converse of this is not true.

Lemma 1 ***(i)*** *Any mixed-criticality system that is successfully scheduled by Algorithm* DISPATCH *using DM priorities can also be scheduled by this algorithm using CAAP priorities.* ***(ii)*** *There are task systems that successfully scheduled by Algorithm* DISPATCH *using CAAP priorities that are not schedulable by this algorithm using CAAP priorities.*

Proof Sketch: In determining which task be assigned lowest-priority task at each stage, CAAP considers all the tasks that have not yet been assigned priorities, including the one with the largest deadline. Hence if the largest-deadline task is the only one that can be assigned lowest priority, CAAP will discover this fact. Therefore any task system that is schedulable using the DM priority assignment is also schedulable using the CAAP priority assignment.

To see that the converse is not true, observe that the system consisting of the following two tasks

τ_i	χ_i	C_i(LO)	C_i(HI)	D_i	T_i
τ_1	HI	0	10	10	∞
τ_2	LO	5	5	5	∞

is scheduled under CAAP priorities (in which τ_2 gets lower priority than τ_1), but not under DM priorities. □

In the remainder of this section, we derive quantitative bounds on the performance of Algorithm DISPATCH when scheduling a task system according to DM priorities. As a consequence of Lemma 1 above, our quantitative bounds hold for the CAAP priority assignment as well.

The reason we have chosen to derive the bounds for DM priorities and then appeal to Lemma 1 to draw conclusions about CAAP, rather than directly derive the bounds for CAAP, is that this DM-based approach allows us to reuse a prior result given in Theorem 1 below. This result characterizes the deadline-monotonic (DM) scheduling of regular (i.e., not mixed-criticality) constrained-deadline sporadic task systems; it links the load of such a system with the response-time of the lowest-priority task and the amount of *blocking* [18] that the task may experience (in addition to interference by the higher-priority tasks).

Theorem 1 (from [20]) *Consider a DM-scheduled instance τ, in which the lowest priority (i.e., largest relative deadline) task has a relative deadline D. Suppose further that this lowest-priority task is subject to* blocking *for an amount $B < D$. Suppose that the worst-case response time of this lowest-priority task is*

$\geq R$, *for some* $R \leq D$. *The load* $\lambda(\tau)$ *of* τ *is bounded from below by the value of* x *that solves the following equation*

$$x = \ln\Big(\frac{R/D}{x + \frac{B}{D}}\Big) \tag{6}$$

□

We now derive, in Theorem 2, a sufficient DM-schedulability condition for mixed-criticality systems in terms of the LO-criticality and HI-criticality load parameters (λ_{LO} and λ_{HI}, as defined in Equations 4 and 5) of the system.

Theorem 2 *Any mixed-criticality system* τ *satisfying*

$$e^{\lambda_{\text{HI}}}\big(\lambda_{\text{HI}} + \lambda_{\text{LO}} \times e^{\lambda_{\text{LO}}}\big) \leq 1 \tag{7}$$

is guaranteed to be schedulable by the scheduling algorithm of Section 4 when priorities are assigned according to deadline-monotonic (DM) order. (Here, e *denotes the base of the natural logarithms:* $e \approx 2.71828...$*)*

Proof: We will derive properties that must be satisfied by every task system that is *not* correctly scheduled by the scheduling algorithm of Section 4 when priorities are assigned according to deadline-monotonic (DM) order; the negation of these properties will yield sufficient schedulability conditions.

Let τ denote a *minimal* task system that is not schedulable by the scheduling algorithm of Section 4 under DM priorities.[6] We consider separately the two possibilities that the lowest-priority (i.e., largest deadline) task in τ is a LO-criticality or a HI-criticality task.

§ *1.* If the lowest-priority task is LO-criticality, τ not being DM-schedulable means that the regular task system $\big\{\big(C_i(\text{LO}), D_i, T_i\big)\big\}_{\tau_i \in \tau}$ is not DM-schedulable. It is therefore the case that when τ is scheduled using DM scheduling the lowest-priority task has a response-time greater than its relative deadline. Applying Equation 6 of Theorem 1 to this scenario, we can therefore set $R \leftarrow D$, where D denotes the relative-deadline of this lowest-priority task. Furthermore since no task in such a system experiences any blocking, we set $B \leftarrow 0$ and conclude that the load of the system – λ_{LO} – must be larger than the value of x satisfying the equation $x = \ln(1/x)$. This value of x is the well-known mathematical constant called the Ω ("Omega") constant, and has a value ≈ 0.567 (see also [21]).

§ *2.* Now consider the case when the lowest-priority task in τ is a HI-criticality task. Let D denote its deadline. Let R_{LO} denote a lower bound on its worst-case response time in any LO-criticality behaviour, and R_{HI} a lower bound on its worst-case response time in any HI-criticality behaviour.

By applying Equation 6 of Theorem 1 to this LO-criticality behaviour, we conclude that

[6] That is, τ is not scheduled correctly, but every subset of τ is. In particular, the task-set obtained from τ by removing the lowest-priority task in τ is DM-schedulable.

$$\lambda_{\text{LO}} = \ln \frac{R_{\text{LO}}/D}{\lambda_{\text{LO}}}$$
$$\Leftrightarrow \lambda_{\text{LO}} \times e^{\lambda_{\text{LO}}} = \frac{R_{\text{LO}}}{D} \tag{8}$$

Equation 8 bounds the LO criticality response-time of the lowest-priority task in terms of the LO-criticality load λ_{LO} and the relative deadline of the lowest-priority task.

Let us now attempt to bound the HI-criticality response-time of the lowest-priority task. Consider any HI-criticality behaviour of the system; without loss of generality, assume that the lowest-priority task has a job arrive at time-instant zero. Let t_1 denote the time-instant at which the run-time scheduling algorithm of Section 4 changed the criticality level indicator Γ from LO to HI. It is evident that $t_1 < R_{\text{LO}}$ (since the lowest-priority task would have completed execution by time-instant R_{LO} in any LO-criticality behaviour). The response-time of the lowest-priority task would therefore be no worse than if it were to be *blocked* for a duration R_{LO} by LO-criticality jobs, in addition to being interfered with by all the greater-priority HI-criticality tasks. That is, we can bound the effect of all the LO-criticality tasks by a single blocking term of duration R_{LO}, and consider only the interference from HI-criticality tasks (which together have load equal to λ_{HI}). To model this scenario in Equation 6 of Theorem 1, we would set $B \leftarrow R_{\text{LO}}$, to get

$$\begin{aligned}
\lambda_{\text{HI}} &= \ln \frac{R_{\text{HI}}/D}{\lambda_{\text{HI}} + (R_{\text{LO}}/D)} \\
&\Leftrightarrow \lambda_{\text{HI}} \times e^{\lambda_{\text{HI}}} + \frac{R_{\text{LO}}}{D} \times e^{\lambda_{\text{HI}}} = \frac{R_{\text{HI}}}{D} \\
&\Leftrightarrow \quad \text{(By Equation 8)} \\
&\qquad \lambda_{\text{HI}} \times e^{\lambda_{\text{HI}}} + \lambda_{\text{LO}} \times e^{\lambda_{\text{LO}}} e^{\lambda_{\text{HI}}} = \frac{R_{\text{HI}}}{D} \\
&\Leftrightarrow e^{\lambda_{\text{HI}}} \big(\lambda_{\text{HI}} + \lambda_{\text{LO}} \times e^{\lambda_{\text{LO}}}\big) = \frac{R_{\text{HI}}}{D}
\end{aligned}$$

Now for the lowest-priority task to not be DM-schedulable, its response-time R_{HI} must exceed its deadline D; i.e., the RHS of the expression above must exceed 1. Equivalently,

$$e^{\lambda_{\text{HI}}} \big(\lambda_{\text{HI}} + \lambda_{\text{LO}} \times e^{\lambda_{\text{LO}}}\big) \leq 1$$

is a sufficient condition for DM-schedulability. This is exactly what is claimed by the theorem, which is therefore proved. □

To better understand the implications of Theorem 2, let us first consider the special cases when one of λ_{LO} and λ_{HI} is equal to zero. If $\lambda_{\text{LO}} = 0$, Inequality 7 becomes

$$\lambda_{\text{HI}} \times e^{\lambda_{\text{HI}}} \leq 1.$$

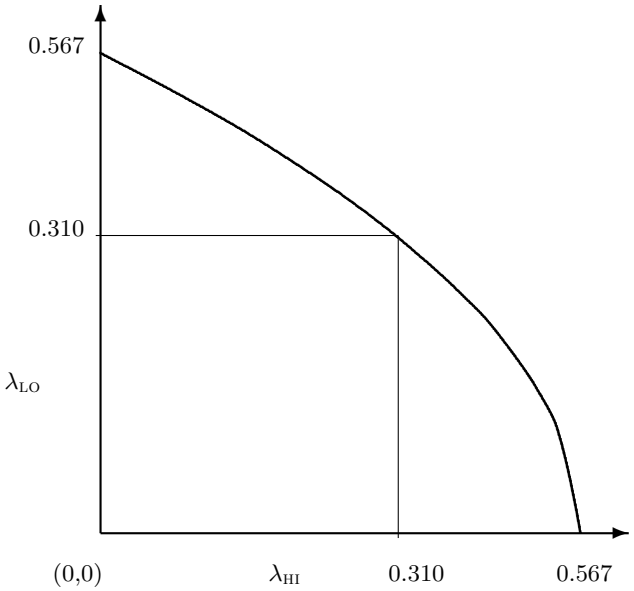

Fig. 2. Bound on the LO-criticality load (λ_{LO}) as a function of HI-criticality load (λ_{HI}).

If $\lambda_{\text{HI}} = 0$, then Inequality 7 becomes

$$\left(e^0(0 + \lambda_{\text{LO}} \times e^{\lambda_{\text{LO}}}) \leq 1\right) \Leftrightarrow \left(\lambda_{\text{LO}} \times e^{\lambda_{\text{LO}}} \leq 1\right)$$

The solution to the equation $x \times e^x = 1$ is again the Ω constant (≈ 0.567). We therefore conclude that when either of λ_{LO} or λ_{HI} equals zero, the other being no larger than Ω is sufficient to ensure that the scheduling algorithm of Section 4 schedules the system in a certifiably correct manner under DM priorities. This turns out to be a direct generalization of a result in [21], stating that any regular (non-MC) sporadic task system with load $\leq \Omega$ is DM-schedulable. Further, based on the result in [21] which states that there are regular sporadic task systems with load exceeding Ω by an arbitrarily small amount that are not schedulable by any fixed-priority scheduling algorithm, we may conclude that there are MC sporadic task systems with $\lambda_{\text{LO}} > \Omega$ or $\lambda_{\text{HI}} > \Omega$ (or both) by an arbitrarily small amount, that are not schedulable correctly by any fixed-priority scheduling algorithm.

In Figure 2, we have plotted the region of values of λ_{LO} and λ_{HI} satisfying Inequality 7; all MC systems that have their λ_{LO} and λ_{HI} parameters map onto points between the axes and the curved line are guaranteed to be scheduled in a certifiably correct manner under DM priorities.

If we were to set $\lambda_{\text{LO}} = \lambda_{\text{HI}} \stackrel{\text{def}}{=} \lambda$ in Inequality 7, we would get

$$\begin{aligned} & e^{\lambda}(\lambda + \lambda \times e^{\lambda}) \leq 1 \\ & \Leftrightarrow \lambda \times e^{\lambda} + \lambda \times e^{2\lambda} \leq 1 \ , \end{aligned}$$

the solution to which is ≈ 0.310. Based on the observation we had made earlier, that in order for a MC system to be scheduled in a certifiably correct manner on a speed-s processor by any scheduling algorithm (including an optimal clairvoyant one) it is necessary that both $\lambda_{\text{LO}} \leq s$ and $\lambda_{\text{HI}} \leq s$, we conclude

Theorem 3 *Any MC task system that can be scheduled by an optimal clairvoyant algorithm in a certifiably correct manner upon a given preemptive uniprocessor platform can be scheduled in a certifiably correct manner by the AMC scheduling algorithm under DM priorities, upon a processor that is* $\approx \frac{1}{0.310}$ *or* ≈ 3.23 *times as fast.*

6 Comparing the AMC and SMC algorithms

The result of Theorem 2 crucially depends upon the fact that AMC scheduling algorithm is able to *recognize* when some job exceeds its LO-criticality WCET, thereby signalling that the current behaviour of the system is a HI-criticality one. Since the SMC algorithm (Section 3) does not exploit this feature, no similar load-based bounds can be derived; this is formally stated in the following theorem.

Theorem 4 *The SMC algorithm may fail to schedule MC systems with both* λ_{LO} *and* λ_{HI} *arbitrarily close to zero, in a certifiably correct manner.*

Proof: Consider the task system consisting of the following two tasks: $\{\tau_1 = (\text{LO}, \epsilon, \infty, 1, 1), \tau_2 = (\text{HI}, 1, 1, 1/\epsilon, 1/\epsilon)\}$, where ϵ is an arbitrarily small positive real number. If τ_1 is assigned lowest priority, it will miss its deadline in the LO-criticality behaviour where its job arrives simultaneously with τ_2's job, and both jobs seek to execute for their LO-criticality WCET's of ϵ and 1 respectively. If τ_2 is assigned lowest priority, then it will miss its deadline in the HI-criticality behaviour where its job arrives simultaneously with τ_1's job and τ_1's job, which has a HI-criticality WCET of infinity, executes for more than $(\frac{1}{\epsilon} - 1)$ units of execution.

The HI-criticality load of this system is equal to $\frac{1}{1/\epsilon}$ or ϵ. The LO-criticality load is equal to $(\frac{\epsilon}{1} + \frac{1}{1/\epsilon})$, which equals 2ϵ. For ϵ arbitrarily small, these are both arbitrarily small, and hence this 2-task MC system bears witness to the correctness of Theorem 4. □

Theorem 4 compared the AMC and SMC algorithms in terms of sufficient schedulability conditions. We now show (Thm. 5) that AMC strictly *dominates* the SMC algorithm: by showing that (i) every task system that can be scheduled in a certifiably correct manner by SMC can also be scheduled in a certifiably correct manner by AMC (with CAAP) (Lemma 2); and (ii) there are task systems schedulable by AMC, that SMC cannot schedule in a certifiably correct manner.

Lemma 2 *If SMC successfully assigns priorities to the tasks in task system* τ*, then so does AMC with CAAP.*

Proof Sketch: Suppose that SMC (as described in Section 3) determines that some task $\tau_i \in \tau$ may be assigned lowest priority. This implies that the smallest positive solution of Equation 1 is $\leq D_i$. We will show that this task may also be assigned lowest priority under CAAP. We consider separately the cases when this lowest-priority task τ_i is a LO-criticality task ($\chi_i = \text{LO}$) and when it is a HI-criticality task ($\chi_i = \text{HI}$).

- In the case when $\chi_i = \text{LO}$, notice that Equation 2 is identical to Equation 1. Recall that $L(\text{LO})$ is, by definition, equal to the smallest positive solution to Equation 2. Hence, the fact that the smallest positive solution of Equation 1 is $\leq D_i$ implies that $L(\text{LO})$ is also $\leq D_i$. Since τ_i is a LO-criticality task, it is consequently possible under CAAP to assign it the lowest priority.
- It remains to consider the case when $\chi_i = \text{HI}$. Since we assumed that τ_i has been assigned lowest priority by the technique of Section 3, it must be the case that the smallest positive solution to Equation 1 is $\leq D_i$. Equation 1 can be rewritten as

$$\begin{aligned} t &= \sum_{\forall j} \left\lceil \frac{t}{T_j} \right\rceil C_j(\text{HI}) \\ &= \sum_{j:\chi_j=\text{LO}} \left\lceil \frac{t}{T_j} \right\rceil C_j(\text{HI}) + \sum_{j:\chi_j=\text{HI}} \left\lceil \frac{t}{T_j} \right\rceil C_j(\text{HI}) \\ &\geq \sum_{j:\chi_j=\text{LO}} \left\lceil \frac{L(\text{LO})}{T_j} \right\rceil C_j(\text{LO}) + \sum_{j:\chi_j=\text{HI}} \left\lceil \frac{t}{T_j} \right\rceil C_j(\text{HI}) \end{aligned} \tag{9}$$

  Upon comparing Inequality 9 above to Equation 3, it becomes evident that the smallest positive solution to Equation 9 (and hence, to Equation 1) is no smaller than the smallest positive solution to Equation 3. Recall that $L(\text{HI})$ is defined to be equal to the smallest positive solution to Equation 3. We may therefore conclude that $L(\text{HI}) \leq D_i$, which in turn implies that CAAP may assign lowest priority to τ_i.

We thus see that for any task system for which SMC identifies a lowest-priority job, CAAP for AMC does likewise. The lemma proves, by repeated applications of this argument. □

Examples 2 and 3 bear witness to the fact that there are MC sporadic task systems to which SMC cannot assign priorities in a certifiably correct manner, whereas AMC and CAAP can; the task system depicted in Table 1 is such an example. It follows from Lemma 2 that every task system that can be scheduled in a certifiably correct manner by SMC can also be scheduled in a certifiably correct manner by AMC with CAAP, allowing us to conclude that at least from the perspective of certifiable correctness,

Theorem 5 *AMC with CAAP dominates the priority assignment technique of SMC.* □

7 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 [22]. 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 this paper, we consider the fixed-priority (FP) scheduling, upon preemptive uniprocessors, of mixed-criticality systems that can be modeled using a mixed-criticality generalization of the sporadic tasks model. We have studied two priority-assignment algorithms: one that assumes limited run-time support for mixed criticalities (SMC), and a new one, AMC with CAAP, that required additional run-time support but is able to provide superior schedulability/ certifiability guarantees when provided with such support. Both these priority-assignment algorithms have relatively efficient implementations: of the same order of run-time complexity as schedulability analysis for "regular" (i.e., non-MC) sporadic task systems; once priorities have been assigned, run-time scheduling is not much more complex than for non-MC systems. This offers up an interesting contrast with non-FP scheduling of MC sporadic task systems, in which the current state of the art, as represented in [9], is an algorithm with potentially pseudo-polynomial run-time complexity.

Acknowledgements

The authors would like to thank Rob Davis for his useful comments. The work presented here forms part of work funded by the Tempo project (EPSRC funded) in the UK. Baruah's research was supported in part by NSF grants CNS 0834270, CNS 0834132, and CNS 1016954; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

References

1. de Niz, D., Lakshmanan, K., Rajkumar, R.: On the scheduling of mixed-criticality real-time task sets. In: Proceedings of the IEEE Real-Time Systems Symposium. (2009) 291–300
2. Vestal, S.: Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In: Proceedings of the IEEE Real-Time Systems Symposium, Tucson, AZ, IEEE Computer Society Press (2007) 239–243
3. Baruah, S., Vestal, S.: Schedulability analysis of sporadic tasks with multiple criticality specifications. In: ECRTS. (2008) 147–155
4. Baruah, S., Li, H., Stougie, L.: Towards the design of certifiable mixed-criticality systems. In: Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS), IEEE (2010)
5. Baruah, S., Bonifaci, V., D'Angelo, G., Li, H., Marchetti-Spaccamela, A., Megow, N., Stougie, L.: Scheduling real-time mixed-criticality jobs. In Hlinený, P., Kucera,

A., eds.: Proceedings of the 35th International Symposium on the Mathematical Foundations of Computer Science. Volume 6281 of Lecture Notes in Computer Science., Springer (2010) 90–101
6. Mok, A.K.: 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.
7. Baruah, S., Mok, A., Rosier, L.: Preemptively scheduling hard-real-time sporadic tasks on one processor. In: Proceedings of the 11th Real-Time Systems Symposium, Orlando, Florida, IEEE Computer Society Press (1990) 182–190
8. Audsley, N.: On priority assignment in fixed priority scheduling. Information Processing Letters **79** (2001) 39–44
9. Li, H., Baruah, S.: An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In: Proceedings of the Real-Time Systems Symposium, San Diego, CA, IEEE Computer Society Press (2010) 183–192
10. Baruah, S., Burns, A.: Implementing mixed criticality systems in Ada. In Romanovsky, A., ed.: Proceedings of Reliable Software Technologies - Ada-Europe 2011, Springer (2011) 174–188
11. Leung, J., Whitehead, J.: On the complexity of fixed-priority scheduling of periodic, real-time tasks. Performance Evaluation (Netherlands) **2** (1982) 237–250
12. Burns, A., Littlewood, B.: Reasoning about the reliability of multi-version, diverse real-time systems. In: Proceedings of IEEE Real-Time Systems Symposium (RTSS), IEEE Computer Society (2010) 73–81
13. Abeni, L., Buttazzo, G.: Integrating multimedia applications in hard real-time systems. In: Proceedings of the Real-Time Systems Symposium, Madrid, Spain, IEEE Computer Society Press (1998) 3–13
14. Bernat, G., Burns, A.: New results on fixed priority aperiodic servers. In: Proceedings 20th IEEE Real-Time Systems Symposium. (1999) 68–78
15. Caccamo, M., Sha, L.: Aperiodic servers with resource constraints. In: Proceedings of the IEEE Real-Time Systems Symposium. (2001)
16. Bernat, G., Burns, A.: Multiple servers and capacity sharing for implementing flexible scheduling. Real-Time Systems Journal **22** (2002) 49–75
17. Zabos, A., Davis, R., Burns, A., Harbour, M.G.: Spare capacity distribution using exact response-time analysis. In: 17th International Conference on Real-Time and Network Systems. (2009) 97–106
18. Audsley, N., Burns, A., Richardson, M., Tindell, K., Wellings, A.: Applying new scheduling theory to static priority preemptive scheduling. Software Engineering Journal **8** (1993) 284–292
19. Joseph, M., Pandya, P.: Finding response times in a real-time system. BCS Computer Journal **29** (1986) 390–395
20. Baruah, S.: Efficient computation of response time bounds for preemptive uniprocessor deadline monotonic scheduling. Technical report, Available at http://www.cs.unc.edu/~baruah/Pubs.shtml (2010)
21. Davis, R., Rothvoss, T., Baruah, S., Burns, A.: Exact quantification of the sub-optimality of uniprocessor fixed priority pre-emptive scheduling. Journal of Real Time Systems **43** (2009) 211–258
22. Barhorst, J., Belote, T., Binns, P., Hoffman, J., Paunicka, J., Sarathy, P., Scoredos, J., Stanfill, P., Stuart, D., Urzi, R.: White paper: A research agenda for mixed-criticality systems (2009) Available at http://www.cse.wustl.edu/~ cdgill/CPSWEEK09_MCAR.