Response-Time Analysis for Mixed Criticality Systems

S.K. Baruah
Department of Computer Science,
University of North Carolina, US.
Email: baruah@cs.unc.edu

A. Burns
Department of Computer Science,
University of York, UK.
Email: burns@cs.york.ac.uk

R.I. Davis
Department of Computer Science,
University of York, UK.
Email: rob.davis@cs.york.ac.uk

***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 schedulability analysis and run-time monitoring. This paper considers a novel implementation scheme for 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). An optimal priority assignment scheme is derived and sufficient response-time analysis is provided. The new scheme formally dominates those previously published. Evaluations illustrate the benefits of the scheme.**

I. Introduction

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 [16] and static verification [23], [9].

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, IMA, 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 behavior 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 this with a contrived example.

Example 1: Consider a system to be implemented on a preemptive fixed priority uniprocessor, that comprises just 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 is not.

*The system designer is confident that each job has a worst-case execution time (WCET) not exceeding 1. Hence all three jobs will complete by their deadlines as long as J_1 is **not** given the lowest priority.*

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. Even if J_1 executes for only 1 time unit, jobs J_2 and J_3 are only schedulable if they are given the highest two priority levels; but if this is done J_1 will miss its deadline even if J_2 and J_3 only execute for 1 time unit.

One might therefore conclude that the system cannot be scheduled in a manner acceptable to both the system designer and the certification authority. Fortunately there is an implementation scheme (and priority assignment) that can satisfy both parties. Consider the priority ordering J_2, then J_1 and finally J_3; and the following run-time behaviour from time 0:

- *Execute the highest priority job J_2 over $[0, 1)$.*
- *If J_2 completes execution by time-instant 1, then execute J_1 over $[1, 2)$ and J_3 over $[2, 3)$, thereby ensuring that all deadlines are met (J_3 could therefore execute over $[2, 3.5)$ if needed).*
- *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 $[1.5, 3)$.*

So if the system designer is right all jobs execute for 1 time unit and meet their deadlines. If the CA is right then the two high-criticality jobs execute for 1.5 time units each and meet their deadlines. Both parties are satisfied. ■

Related work. In prior work [7], [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 systems are better modeled as collections of *recurrent tasks* that are specified using, e.g., the sporadic tasks model [19], [8]. Schedulability analysis of such systems is typically far more difficult than the analysis of systems modeled as collections of independent jobs. Vestal [23] 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 [2], for assigning priorities optimally to the tasks in such a system. Some other works (e.g., [9], [18]) have considered algorithms that are not fixed-priority, for scheduling mixed-criticality systems in a certifiably correct manner.

In this paper we develop the scheduling scheme illustrated in the example above. Sufficient response-time analysis is derived and evaluated via comparisons with previous schemes. This evaluation indicates that the scheme leads to a significant improvement in system schedulability.

II. System Model

A system is defined as a finite set of components $\mathcal{K}$. Each component has a level of criticality (defined by the system's engineer responsible for the entire system), L, and contains a finite set of sporadic tasks. Each task, τ_i, is defined by its period (minimum arrival interval), deadline, computation time and criticality level: (T_i, D_i, C_i, L_i). These parameters are however not independent, in particular the worst-case computation time, C_i, 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_i.

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 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 (executing beyond its deadline) will reduce for higher levels of L (due to C being monotonically non-decreasing 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, $\vec{C}$, L), where $\vec{C}$ is a vector of values – one per criticality level, with the constraint:

$$L1 > L2 \Rightarrow C(L1) \geq C(L2)$$

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

The general task τ_i with criticality level L_i will have one value from its $\vec{C_i}$ 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.

Definition 1 (Behaviors): During different runs, any given task system will, in general, exhibit different *behaviors*: different jobs may be released at different time instants, and may have different actual execution times. Let us define the *criticality level of a behavior* to be the smallest criticality level such that no job executed for more than its C value at this criticality level.

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* $\geq L$ *will complete by their deadlines in any criticality-L behavior*.

In this paper we consider only the issues surrounding the static verification of a mixed criticality (MC) system scheduled by the standard fixed priority preemptive dispatcher on a single processor. We 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 III;
- Adaptive Mixed Criticality (AMC) – a novel scheme, introduced in Section IV.

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 (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 representative 'worst-case execution time') will not impact on any higher criticality jobs. No run-time monitoring is required.

However, if run-time support is provided then the kinds of performance guarantees that can be made in scheduling mixed-criticality systems are enhanced. 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., [13]) monitor execution times so that erroneous behavior 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 (e.g., [1], [11]), 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 [24].

The other two priority assignment schemes 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. The *Adaptive* scheme (AMC) goes further and does not allow jobs of criticality L to execute *at all* if any job (of equal or higher criticality) executes for more than its $C(L)$ parameter. The main contribution of this paper is the introduction of response-time analysis for this Adaptive Mixed Criticality scheme.

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. We also consider only independent tasks with constrained deadlines. We have explored the more general M-criticality level (M>2) model and it introduces no added fundamental issues apart from the need to deal with concurrent criticality changes (for example a change from LO to ME (medium) in progress when a change from ME to HI occurs) – this topic is left to 'future work'.

III. 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, descheduled until it is safe for them to execute again. Means of programming recovery are explained elsewhere [6]. In this section we first review response-time analysis for SMC and then consider priority assignment.

A. Scheduling Analysis for SMC

The distinctive feature of *mixed criticality* as opposed to *partitioned criticality* is that schedulability is obtained from optimising the temporal characteristics of the tasks rather than their importance/criticality parameter.

Consider the common deadline-monotonic priority assignment scheme. Here the key operational parameter, priority (P), is derived solely from the deadlines of the tasks. For any two tasks τ_i and τ_j: $D_i < D_j \Rightarrow P_i > P_j$. For mixed criticality systems this means that a task may suffer interference from another task with a higher priority but a lower criticality level. (A phenomenon referred to as *criticality inversion*.)

To test for schedulability, the standard Response Time Analysis (RTA) [17], [3] approach first computes the worst-case completion time for each task (its response time, R) and then compares this value with the task's deadline D (i.e. tests for $R_i \leq D_i$ for all tasks τ_i). The response time value is obtained from the following (where $\mathbf{hp}(i)$ denotes the set of tasks with priority higher than that of task τ_i):

$$R_i \;=\; C_i \;+\; \sum_{\tau_j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad (1)$$

This is solved using standard techniques for solving recurrence relations.

For a criticality monotonic system, eqn (1) is applied directly once the priorities have been assigned according to a 'deadline monotonic within criticality monotonic' algorithm.

For SMC three cases need to be considered depending on whether the arbitrary higher priority task τ_j has an equal, higher or lower criticality than τ_i [6]. For each case the correct value of C_j must be ascertained:

1) If $L_i = L_j$ then the tasks are at the same level of criticality and the normal representative value C_j is used.
2) If $L_i < L_j$ then it is not necessary to use the large value of computation time represented by C_j, rather the smaller amount corresponding to the criticality level of τ_i should be used (as this is the level of assurance needed for this task). Hence eqn (1) should use $C_j(L_i)$.
3) If $L_i > L_j$ then we have criticality inversion. One approach here would be to again use $C_j(L_i)$, but this is allowing τ_j to execute for far longer than the task is assumed to do at its own criticality level. Moreover, it would require all low criticality tasks to be verified to the highest levels of importance, which would be prohibitively expensive (and in many ways undermine one of the reasons for having different criticality levels in the first place). Rather we should employ C_j, **but the run-time system must ensure that τ_j does not execute for more than this value.**

The latter point is crucially important. Obviously all the shared run-time software must be verified to the highest criticality level of the application components. One aspect of this is the platform functionality that monitors the execution time of tasks and makes sure they do not ask for more resource than was catered for during the analysis phase of the system's verification.

The response time equation, eqn (1), can be rewritten as:

$$R_i \;=\; C_i \;+\; \sum_{\tau_j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(min(L_i, L_j)) \qquad (2)$$

Note that this use of minimum implies that values of C are only required for the task's criticality level and all lower criticality levels. This is in contrast to the scheme originally defined by Vestal [23] in which there is no monitoring of computation time and hence all LO-critical tasks must also be analysed for their C(HI) value. As a result eqn (2) becomes:

$$R_i \;=\; C_i \;+\; \sum_{\tau_j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(L_i) \qquad (3)$$

As indicated earlier, this requirement to verify LO-criticality tasks to the same level of assurance as HI-criticality tasks would be prohibitively expensive in practice. In the evaluation work (Section V) this version of SMC is termed **SMC-NO** (meaning no run-time support required).

B. Priority Assignment for SMC

Vestal [23] showed that deadline monotonic priority assignment is not optimal for schemes in which tasks have more than one 'worst-case execution time'. As a result, SMC assigns priorities by applying a version of Audsley's priority assignment algorithm [2]. 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 priority assignment is recursively obtained for the remaining tasks. Proof that Audsley's approach is applicable to SMC has already been published [23], [6]. The advantage of applying Audsley' algorithm is that it delivers an optimal assignment in a maximum of $n(n+1)/2$ steps. If the algorithm were not applicable then an exhaustive search over all $n!$ possible orderings would be required.

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

τ_i	L	$C_i(\text{LO})$	$C_i(\text{HI})$	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 the 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 the lowest priority if it would meet its deadline as the lowest-priority job. According to eqn (2), the worst-case response time of any of τ_3's jobs is equal to the smallest positive solution to the following recurrence relation:

$$R_3 = 20 + \left\lceil \frac{R_3}{2} \right\rceil \cdot 1 + \left\lceil \frac{R_3}{10} \right\rceil \cdot 2$$

It is easily verified that 68 is a solution to this recurrence, since for $t = 68$ the RHS evaluates to $(20 + 34 + 14) = 68$, 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 the 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; meaning it is unschedulable according to SMC. We shall return to this example later. ■

Run-time complexity. As noted above, the use of Audsley's algorithm reduces the complexity of finding the optimal priority ordering from $n!$ to $n(n+1)/2$ schedulability tests. Here we observe that for mixed criticality systems, with $D \leq T$ for all tasks, the priority assignment problem is even more straightforward.

Theorem 1: An optimal priority ordering exists that has all tasks with the same criticality assigned priorities in deadline monotonic priority order.

Proof. The proof follows the standard method of proving that deadline monotonic priority ordering is optimal for tasks with constrained deadlines (as described in any standard textbook such as [14]). Assume τ_i is deemed schedulable at the lowest priority (i.e., $R_i \leq D_i$). Let τ_k be any task with the same criticality level as τ_i but a larger deadline: $D_k > D_i$. If τ_i is exchanged with τ_k (so it becomes the lowest priority task) then τ_k will suffer exactly the same interference from LO and HI tasks and hence the busy period for τ_k will be the same as that for τ_i when it was assigned the lowest priority (both will contain a single C_i and a single C_k term as $D \leq T$ for all tasks). Hence $R_k = R_i \leq D_i \leq D_k$ (where R_i is the value computed when τ_i was assigned the lowest priority). It follows that τ_k is schedulable at the lowest priority and that the task with the longest deadline (but identical criticality) is the best candidate to place at this level. □

Hence in seeking to determine whether some task may be assigned the 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 a total of $2n-1$ tests are needed – compared to $n(n+1)/2$ for general systems that are compatible with Audsley's algorithm.

Note this observation about priority assignment for SMC is also applicable to the scheme used with AMC.

IV. Adaptive Mixed Criticality - AMC

With a platform that can monitor for how long individual jobs have been executing, the following adaptive run-time scheduling algorithm can 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 the following rules:

R1: There is a *criticality level indicator* Γ, initialized to LO.

R2: While $(\Gamma \equiv LO)$, at each instant the waiting job generated by the task with highest priority is selected for execution.

R3: If the currently-executing job executes for its LO-criticality WCET without signalling completion, then $\Gamma \leftarrow HI$.

R4: Once $(\Gamma \equiv HI)$, jobs with criticality level $\equiv LO$ will *not* be executing. Henceforth, therefore, at each instant the waiting job generated by the HI-criticality task with the highest priority is selected for execution.

R5: 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 paper – a robust scheme for SMC is described elsewhere [6] and could be adapted for AMC.)

Note, no run-time servers or dynamic slack reclaiming algorithm is needed for AMC – just execution time monitoring.

It is assumed that a switch from LO to HI occurs, due to the lack of a completion signal, as the computation time $C(LO)$ is reached (i.e., not at $C(LO) + \delta$: for some positive small value δ). Hence a task with a period of 10 and a response time in LO (R_i^{LO}) of 6 can invoke a criticality switch at time 6. Note, a task could execute for $C_i(LO)$ before R_i^{LO} (due to experiencing less interference from higher priority tasks) and hence the criticality switch could occur earlier. And for a

sporadic task, released later, the switch could occur later than R_i^{LO}. To cater for this we define an interval during which the switch could occur and derive safe bounds for this interval (see below).

To summarise the main difference between SMC and AMC, in SMC any LO-critical task τ_l is descheduled if it executes for more than $C_l(LO)$. While in AMC, *all* LO-critical tasks are descheduled if *any* job (from any task τ_a) executes for more than $C_a(LO)$. If a HI-critical job (τ_h) executes for more than $C_h(LO)$ (but no greater than $C_h(HI)$) then, under SMC, LO-critical tasks continue to execute but may miss their deadlines; but under AMC they stop executing. In both schemes HI-critical tasks continue to meet their deadlines. It follows that AMC's behaviour subsumes that of SMC. We shall show that this leads to the property that AMC dominates SMC in terms of schedulability.

A. Response-Time Analysis for AMC

The analysis presented in this section has a number of similarities to that for systems subject to mode changes [21], [22], [20]. Fortunately the model here is simpler than the one applicable to general mode changes.

The form that the analysis takes has three phases:

1) Verifying the schedulability of the LO-criticality mode,
2) Verifying the schedulability of the HI-criticality mode,
3) Verifying the schedulability of the criticality change itself.

Note the third phase is necessary as it cannot be deduced from the schedulability of the stable modes [22]. For these stable modes, standard response time analysis can be applied – eg. eqns (4) and (5).

$$R_i^{LO} \;=\; C_i \;+\; \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{R_i^{LO}}{T_j} \right\rceil C_j(LO) \qquad (4)$$

where $\mathbf{hp(i)}$ is the set of all tasks with priority higher than that of task τ_i.

$$R_i^{HI} \;=\; C_i \;+\; \sum_{j \in \mathbf{hpH}(i)} \left\lceil \frac{R_i^{HI}}{T_j} \right\rceil C_j(HI) \qquad (5)$$

where $\mathbf{hpH(i)}$ is the set of HI-critical tasks with priority higher than, or equal to, that of task τ_i – we shall also use $\mathbf{hpL(i)}$ to denote the LO-critical tasks with priority higher than, or equal to, that of task τ_i. Note R_i^{HI} is only defined for tasks of criticality HI.

The rest of this section deals with the analysis of the criticality change. We will use for illustration the task set given earlier in Example 2, but with $C_2(HI) = 5$. With this task set the LO and HI response times are calculated by the above equations to be: $R_1^{LO} = 1$, $R_2^{LO} = 2$, $R_2^{HI} = 5$, $R_3^{LO} = 50$ and $R_3^{HI} = 40$. Both modes are therefore schedulable.

In general exact tractable response-time analysis is possible if the critical instant is clear (and is ideally when all tasks are released together) and the worst-case occurs when sporadic tasks arrive at their maximum rate. As the following analysis of our example illustrates, this second property is unfortunately not the case with AMC.

In the example, and considering the response time of τ_3, a criticality change can be invoked by τ_2 if it executes for 1 unit of time (without signaling completion) at any of its first 5 releases (i.e., at times 0, 10, 20, 30 or 40). The next release is not before R_3^{LO} and hence cannot have an impact. Testing each of these releases shows that the release at time 40 is the worst. From time 40, τ_1 would execute for 1 unit, τ_2 would execute for 1 unit and invoke the criticality change. So the response time of τ_3 during the criticality change (R_3^*) could be computed by:

$$R_3^* = 20 + 21 + 4 + \left\lceil \frac{R_3^* - 40}{10} \right\rceil \cdot 5$$

which has a solution of 50 (the value 4 comes from the first 4 executions of τ_2). But if the final release of τ_2 is delayed by 4 units then τ_1 would execute two more times:

$$R_3^* = 20 + 23 + 4 + \left\lceil \frac{R_3^* - 44}{10} \right\rceil \cdot 5$$

which has a solution of 52.

We conclude that exact analysis for AMC is unlikely to be tractable (as all release patterns of all sporadic tasks would need to be tested). Indeed even for a periodic task model there is the problem that the critical instant is not straightforward to compute – in the above example if τ_2 is strictly periodic but initially released at time 4 rather than 0 then the same impact on the worst-case occurs. In the remainder of this section we therefore concentrate on sufficient analysis. We derive two forms; one is an adaptation of the analysis presented in Section III for SMC, the other considers a number of different points at which the criticality change could occur and takes the maximum value obtained from each of these points. The latter is still tractable, but involves more computation.

B. Sufficient Analysis for AMC - Method 1

A simple form of sufficient analysis can be derived from that described earlier for the SMC approach. Consider eqn (2) rearranged to separate out the two criticality levels:

$$R_i \;=\; C_i \;+\; \sum_{\tau_j \in \mathbf{hpH}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(min(L_i, L_j)) \;+\; \sum_{\tau_k \in \mathbf{hpL}(i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k(LO)$$

During the criticality change we are only concerned with HI-critical tasks, so for $L_i = HI$:

$$R_i \;=\; C_i(HI) \;+\; \sum_{\tau_j \in \mathbf{hpH}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(HI) \;+\; \sum_{\tau_k \in \mathbf{hpL}(i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k(LO) \qquad (6)$$

This equation for SMC is conservative for AMC as it does not take into account the fact that LO-critical tasks cannot execute for the entire busy period of a high criticality task in the HI-mode. A change to HI-criticality must occur before R_i^{LO} and hence eqn (6) can be modified to the following:

$$R_i^* \;=\; C_i(HI) \;+\; \sum_{\tau_j \in \mathbf{hpH}(i)} \left\lceil \frac{R_i^*}{T_j} \right\rceil C_j(HI) \;+\; \sum_{\tau_k \in \mathbf{hpL}(i)} \left\lceil \frac{R_i^{LO}}{T_k} \right\rceil C_k(LO) \tag{7}$$

this 'caps' the interference from LO-critical tasks as the response-time during the change, R_i^*, must be greater than R_i^{LO}. In the evaluation (see Section (V)) this form of sufficient analysis is denoted as **AMC-rtb** (for response time bound). One obvious property of this analysis is that a task system that is schedulable under SMC is also schedulable under AMC (as analysed by Method 1, i.e.; eqn (7)). The analysis of SMC is identical apart from the 'cap' that reduces the interference from LO-critical tasks.

If this analysis is applied to the task set of Example 2) then

$$R_3^* = 20 + \left\lceil \frac{R_3^*}{10} \right\rceil \cdot 5 + \left\lceil \frac{50}{2} \right\rceil \cdot 1$$

which has a solution of 85 which is less than the task's deadline (100). Recall that when using SMC the system is unschedulable.

C. Sufficient Analysis for AMC - Method 2

In this approach we derive an expression for the maximum interference on a HI-critical task if a criticality change, invoked by task τ_s, occurs at some arbitrary time s. The criticality change is triggered by any task executing for more than $C(LO)$. If this event impacts on task τ_i then $s < R_i^{LO}$, and the priority of τ_s must be equal or greater than that of τ_i; otherwise task τ_i will have completed before the criticality change happened. A formulation for R_i^s is constructed from the different forms of interference it experiences[1].

$$R_i^s = C_i(HI) + I_L(s) + I_H(s) \tag{8}$$

where $I_L(s)$ is the interference from low criticality tasks, and $I_H(s)$ is the interference from tasks with higher or equal criticality.

In this formulation we could differentiate between those tasks that have a priority greater than τ_s, and those that have a lower priority. Those with priority greater than τ_s must have completed this 'current' job (so only executed for $C(LO)$), while those with priority equal or less may not yet have completed and hence their current job must be assumed to need $C(HI)$. However, we will later want to use this analysis in an optimal priority ordering scheme based on Audsley's algorithm. For this algorithm to be applicable the response time of τ_i can be a function of the set of higher priority tasks, but must *not* depend on the relative priority ordering of these tasks. To distinguish between these two groups of HI-criticality tasks would break this rule, and so we use the simpler form of eqn (8).

The low criticality tasks are prevented from executing after s so their worst-case interference is bounded by:

$$I_L(s) = \sum_{j \in \mathbf{hpL}(i)} \left(\left\lfloor \frac{s}{T_j} \right\rfloor + 1 \right) C_j(LO) \tag{9}$$

The value *floor + 1* is used rather than *ceiling* as we need to include interference from any low criticality task as soon as it is released. Note this formulation for $I_L(s)$ is an upper bound, to compute this value exactly would require the amount of computation completed before s to be calculated. Here we assume, for the analysis, any task started at or before s completes. For a possible alternative system model (in which all low criticality tasks, that have been released but not yet completed, are allowed to consume up to $C(L)$ before being descheduled) this bound is tight.

There are a number of possible ways of deriving a sufficient (minimum) value of $I_H(s)$. Here we exploit a simple technique that is clearly sufficient – we assume all jobs active at time s execute for $C(HI)$. Hence only jobs with a deadline before s contribute a $C(LO)$ value. However to be assured that the analysis is conservative the worst-case phasing of the these jobs needs to be taken into account.

Consider the interference from any such task (τ_k) at time t with $t > s$. The maximum number of releases of τ_k is

$$\left\lceil \frac{t}{T_k} \right\rceil$$

The maximum number of releases that can fit into an interval of length $t - s$ is bounded (for tasks with $T = D$) by:

$$\left\lceil \frac{t-s}{T_k} \right\rceil + 1$$

If $D < T$ then this value can be improved upon as there is forced to be an interval (for a schedulable system) in which τ_k is not executing – i.e., the time between its deadline and next release, so the above value becomes (assuming $(t-s) > (T_k - D_k)$):

$$\left\lceil \frac{t-s-(T_k-D_k)}{T_k} \right\rceil + 1 \tag{10}$$

If s is small and D_k is close to T_k then eqn (10)) can be pessimistic, and include more jobs than can actually be present in an interval of length t. Therefore we define

$$M(k,s,t) = \min \left\{ \left\lceil \frac{t-s-(T_k-D_k)}{T_k} \right\rceil + 1, \left\lceil \frac{t}{T_k} \right\rceil \right\} \tag{11}$$

The interference term at time t becomes

$$I_H(s) = \sum_{k \in \mathbf{hpH}(i)} \Big\{ (M(k,s,t)\; C_k(HI)) +$$

[1] R_i^s denotes the response time of task τ_i when a criticality change occurs as time s (lower case) relative to the release of τ_i.

$$\left(\left\lceil \frac{t}{T_k} \right\rceil - M(k,s,t)\right)\ C_k(LO)\Big\}$$

And therefore

$$R_i^s = C_i(H) + \sum_{j \in \mathbf{hpL}(i)} \left(\left\lfloor \frac{s}{T_j} \right\rfloor + 1\right)\ C_j(LO) + \sum_{k \in \mathbf{hpH}(i)} \Big\{ \left(M(k,s,R_i^s)\ C_k(HI) + \left(\left\lceil \frac{t}{T_k} \right\rceil - M(k,s,R_i^s)\right)\ C_k(LO)\right)\Big\} \quad (12)$$

with

$$R_i^* = \max\left(R_i^s\right) \forall s \quad (13)$$

Finally for this approach we need to define the values of s that need to be considered. The overall interval of possible s values runs from 0 to R_i^{LO}. An examination of eqn (12) shows that the $\mathbf{hpL}$ term increases as a step function for increased values of s; while the $\mathbf{hpH}$ term decreases as s increases. It follows that R_i^s can only increase at the points at which a LO-criticality tasks are released – hence these are the points that need to be considered. Note that a LO-criticality job released at exactly R_i^{LO} will not be allowed to execute, and hence s in restricted to the interval [0,R_i^{LO}). There could be a large number LO-criticality job released in this interval for a sizeable application, but the upper bound of R_i^{LO} prevents the scheme from becoming intractable. Again pessimism is introduced here as not all the s points can actually correspond to times at which a task reaches its $C(LO)$ value.

Returning to our running example, τ_1 is released 25 times between 0 and 48. Each of these values for s needs to be checked. The worst case occurs when $s = 48$; this leads to:

$$R_3^{48} = 20 + \left(\left\lfloor \frac{48}{2} \right\rfloor + 1\right) \cdot 1 + \left\lfloor \frac{48}{10} \right\rfloor \cdot 1 + \left(\left\lceil \frac{R_3^{48}}{10} \right\rceil - \left\lfloor \frac{48}{10} \right\rfloor\right) \cdot 5$$

that is

$$R_3^{48} = 20 + 25 + 4 + \left(\left\lceil \frac{R_3^{48}}{10} \right\rceil - 4\right) \cdot 5$$

which has a solution of 59 (less than the 85 calculated by Method 1).

A formal comparison of Methods 1 and 2 shows that any task set deemed schedulable by Method 1 will also be found to be schedulable by Method 2. This follows from a direct comparison of the scheduling equations. Method 1 assumes that LO-criticality tasks can execute concurrently with HI-criticality tasks (executing at their $C(HI)$ level) for the entire interval [0,R_i^{LO}). Method 1 also assumes that HI criticality tasks execute for $C(HI)$ for all of the interval of length t). Method 2 may reduce one or both of these values; however, they are both upper bounded by the values used in Method 1. Hence Method 2 dominates Method 1.

D. Priority assignment for AMC

As AMC is an extension of SMC it follows that deadline monotonic priority ordering is again not optimal. Fortunately Audsley's algorithm is again applicable. Formally for this to be the case a schedulability test (referred to as S below) must adhere to the following conditions [15]:

- **Condition 1**: The schedulability of a task τ_k may, according to test S, depend on any independent properties of tasks with priorities higher than τ_k, but not on any properties of those tasks that depend on their relative priority ordering.
- **Condition 2**: The schedulability of a task τ_k may, according to test S, depend on any independent properties of tasks with priorities lower than τ_k, but not on any properties of those tasks that depend on their relative priority ordering.
- **Condition 3**: When the priorities of any two tasks of adjacent priority are swapped, the task being assigned the higher priority cannot become unschedulable according to test S, if it was previously schedulable at the lower priority. (As a corollary, the task being assigned the lower priority cannot become schedulable according to test S, if it was previously unschedulable at the higher priority).

An inspection of the scheduling equations for both of the AMC methods shows that these properties hold. It follows that Audsley's algorithm can be employed. Moreover, the property embedded in Theorem 1 is also valid for both of the schedulability tests (Methods 1 and 2) and hence at most $2n - 1$ tests are needed to find a feasible priority ordering if one exists.

E. Comparing AMC and SMC

Here, before a more extensive evaluation, we prove that AMC strictly dominates SMC.

Theorem 2: Any sporadic task system that is schedulable under the rules and priority assignment scheme of SMC is also schedulable by the rules and priority ordering of AMC.

Proof. Considering only Method 1, we noted earlier that the scheduling equation for AMC is exactly that for SMC with some interference removed. The theorem therefore holds □

We have already shown an example that is schedulable by AMC but not SMC and hence we can conclude that AMC strictly dominates SMC. Note that this dominance holds even though SMC is analysed by a sufficient and necessary scheme while AMC has only a sufficient test. Recall also that Method 2 dominates Method 1.

V. Evaluation

In this section, we present an empirical investigation, examining the effectiveness of our analysis techniques and the AMC scheme itself.

A. Taskset parameter generation

The taskset parameters used in our experiments were randomly generated as follows:

- Task utilisations ($U_i = C_i/T_i$) were generated using the UUnifast algorithm [12], giving an unbiased distribution of utilisation values.
- Task periods were generated according to a log-uniform distribution with a factor of 100 difference between the minimum and maximum possible task period. This represents a spread of task periods from 10ms to 1 second, as found in many hard real-time applications.
- Task deadlines were set equal to their periods.
- The low criticality execution time of each task was set based on the utilisation and period selected: $C_i(LO) = U_i/T_i$.
- The high criticality execution time of each task was a fixed multiplier of the low criticality execution time, $C_i(HI) = CF \cdot C_i(LO)$ (e.g., $CF = 2.0$).
- The probability that a generated task was a high criticality task was given by the parameter CP (e.g. $CP = 0.5$).

B. Schedulability tests investigated

We investigated the performance of the following techniques and associated schedulability tests.

- UB-H&L: A composite upper bound, to pass this test, a taskset must be deemed schedulable according to both UB-L and UB-H where, UB-L is an upper bound on taskset schedulability obtained by considering execution of all tasks at the low criticality level and deadline monotonic priority ordering (DMPO) which is optimal in this case; and UB-H is an upper bound based solely on the schedulability of the high criticality tasks executing for their high criticality execution times, again assuming DMPO. Any taskset that fails the UB-H&L test cannot be schedulable using any fixed priority multi-criticality scheduling technique. This upper bound value is depicted as a dashed line in the Figures presented below[2].
- AMC-max: Method 2 described in Section IV-C.
- AMC-rtb: Method 1 described in Section IV-B.
- SMC: the approach described in Section III-A
- SMC-NO: the SMC approach without run-time monitoring (i.e., the approach published by Vestal [23]) described at the end of Section III-A.
- CrMPO: Criticality Monotonic Priority Ordering. Task priorities were ordered first according to criticality (highest criticality first) and then according to deadline (shortest deadline first). Response time analysis was then used to determine if the taskset was schedulable with high criticality tasks assumed to execute for $C(HI)$ and low criticality tasks for $C(LO)$ – i.e., one execution time parameter per task.

C. Experiments

In our experiments, the taskset utilisation was varied from 0.025 to 0.975[3]. For each utilisation value, 1000 tasksets were generated and the schedulability of those tasksets determined using the six algorithms / schedulability tests. The graphs are best viewed online in colour.

[2]For the sake of clarity and to avoid having too many lines on the graphs, UB-L and UB-H are not shown.

[3]Utilisation here is computed from the $C(L)$ values only.

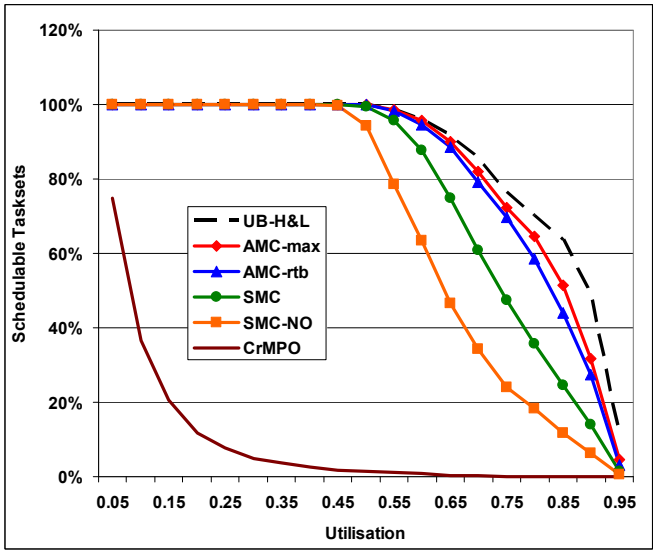

Fig. 1. Percentage of Schedulable Tasksets

Fig 1 plots the percentage of tasksets generated that were deemed schedulable for a system of 20 tasks, with on average 50% of those tasks having high criticality (CP = 0.5) and each task having a high criticality execution time that is 2.0 times its low criticality execution time (CF = 2.0).

We observe that the SMC schedulability test outperforms that for SMC-NO by a large margin. This is expected as SMC-NO requires that low criticality tasks are schedulable with all higher priority high-criticality tasks executing for their high criticality execution times, whereas SMC only requires such schedulability with higher priority high-criticality tasks executing for their low criticality execution times. AMC-rtb further significantly improves on the performance of SMC. Again this is expected: when computing the high criticality response time of a task τ_i, SMC includes interference from jobs of higher priority, low criticality, tasks beyond the low criticality response time of task τ_i, whereas AMC-rtb does not. Finally, AMC-max makes a small but useful improvement over AMC-rtb, giving overall performance that is close to the limit illustrated by the UB-H&L upper bound.

In the following figures we show the weighted schedulability measure $W_y(p)$ [10] for schedulability test y as a function of parameter p. For each value of p, this measure combines results for all of the tasksets τ generated for all of a set of equally spaced utilization levels (0.025 to 0.975 in steps of 0.025).

Let $S_y(\tau, p)$ be the binary result (1 or 0) of schedulability test y for a taskset τ with parameter value p:

$$W_y(p) = (\sum_{\forall \tau} u(\tau) \cdot S_y(\tau, p)) / \sum_{\forall \tau} u(\tau) \qquad (14)$$

where $u(\tau)$ is the utilization of taskset τ.

The weighted schedulability measure reduces what would otherwise be a 3-dimensional plot to 2 dimensions [10]. Weighting the individual schedulability results by taskset utilization reflects the higher value placed on being able to schedule higher utilization tasksets.

We show how the results are changed by varying each of the key parameters (one at a time). Fig 2 varies the criticality factor, Fig 3 varies the percentage of tasks with high criticality and Fig 4 varies the size of the task set. A number of points are illustrated by these figures:

- CrMPO preforms very badly as priority ordering is far from optimal.
- Fig 3 has a U-spaced curve because each end of the interval represents a one-criticality task set, and hence the priorities are optimal.
- Figs 2 and 3 show a decline in schedulability for high criticality factors or high percentage of HI-critical tasks; but this is due to utilisation (as used in the Y-axis) being calculated via the $C(LO)$ values only. In effect utilisation goes up (and hence schedulability goes down) as the parameter of the figure increases.

Overall the key observation of these figures, representing an extensive set of experiments, is that the relationship between the six lines on the graphs remains stable, and that AMC is a very effective means of scheduling mixed criticality systems.

To show that task systems with deadline less than period behave in a similar way, the fifth experiment (see Fig 5) returns to the same basic parameter settings as the first experiment but fixes each task's deadline by choosing a random value between $C_i(HI)$ and T_i for HI-critical tasks and between $C_i(LO)$ and T_i for LO-critical tasks.

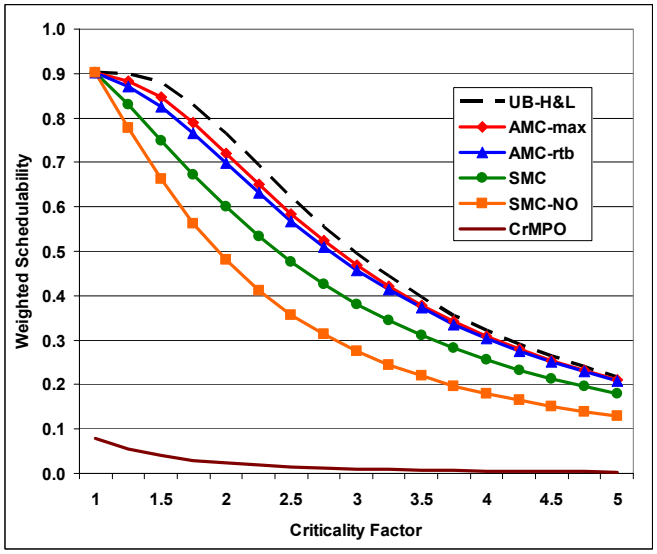

Fig. 2. Varying the Criticality Factor

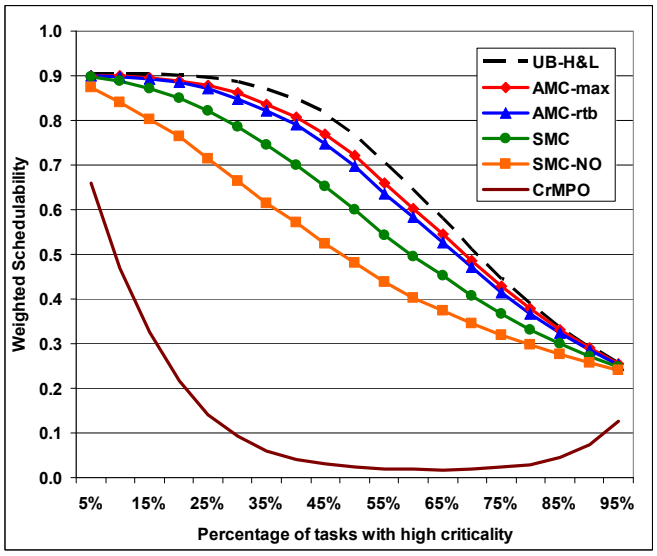

Fig. 3. Varying the Criticality Mix

VI. Conclusion

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 [4]. We believe that in mixed-criticality systems, these certification considerations give rise to fundamental new resource allocation and scheduling challenges that are not adequately addressed by conventional real-time scheduling theory.

In this paper, we consider 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 scheduling algorithms: one that assumes limited run-time support for mixed criticalities (SMC), and a new one, AMC, that requires additional run-time support but is able to provide superior schedulability/certifiability guarantees when provided with such support. Both these approaches have relatively efficient implementations: of the same order of run-time complexity as "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 [18], is an algorithm with potentially pseudo-polynomial run-time complexity.

In further work we will generalize the analysis presented here to multiple criticality levels; we shall also extend the

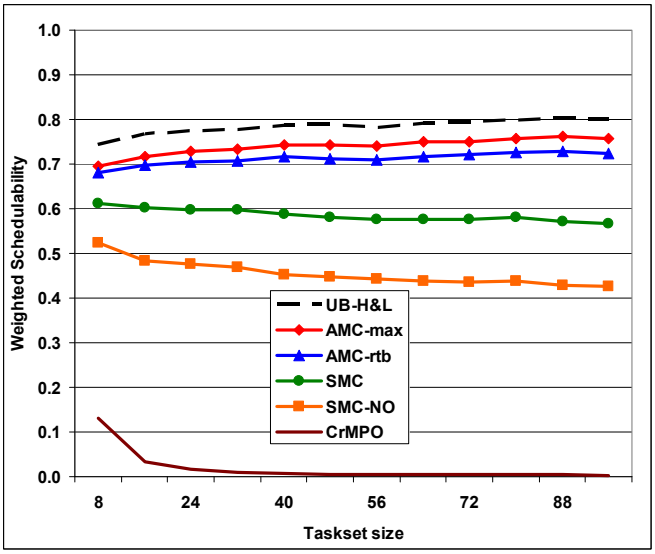

Fig. 4. Varying the Number of Tasks

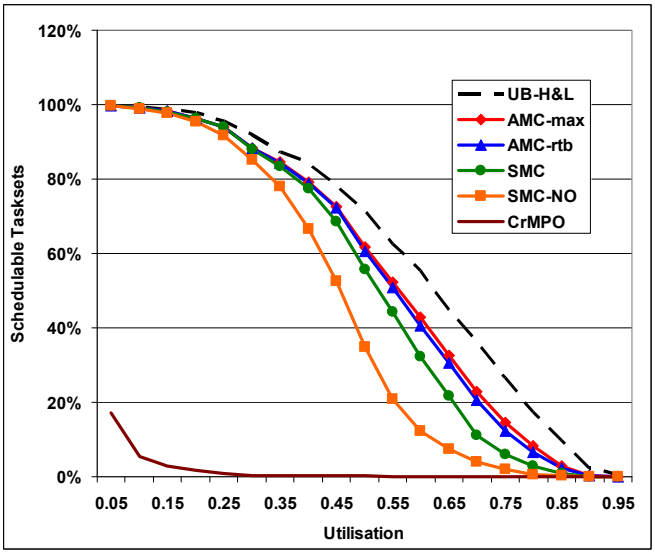

Fig. 5. Deadline less than Period

task model to include the standard notions of blocking and jitter - and arbitrary deadlines.

Acknowledgements

Work supported by NSF grants CNS 0834270, CNS 0834132, and CNS 1016954; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; AFRL grant FA8750-11-1-0033 and EPSRC(UK) Tempo grant.

REFERENCES

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the Real-Time Systems Symposium*, pages 3–13, Madrid, Spain, December 1998.

[2] N. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

[3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[4] J. Barhorst, T. Belote, P. Binns, J. Hoffman, J. Paunicka, P. Sarathy, J. Scoredos, 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.

[5] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. In P. Hlinený and A. Kucera, editors, *Proceedings of the 35th International Symposium on the Mathematical Foundations of Computer Science*, volume 6281 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2010.

[6] S. Baruah and A. Burns. Implementing mixed criticality systems in Ada. In A. Romanovsky, editor, *Proceedings of Reliable Software Technologies - Ada-Europe 2011*, pages 174–188. Springer, 2011.

[7] 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.

[8] 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.

[9] S. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *ECRTS*, pages 147–155, 2008.

[10] A. Bastoni, B. Brandenburg, and J. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *Proceedings of Sixth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 33–44, 2010.

[11] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *Proceedings 20th IEEE Real-Time Systems Symposium*, pages 68–78, 1999.

[12] E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Journal of Real-Time Systems*, 30(1-2):129–154, 2005.

[13] A. Burns and B. Littlewood. Reasoning about the reliability of multi-version, diverse real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, pages 73–81, 2010.

[14] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley Longman, 4th edition, 2009.

[15] R. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems Journal*, pages 1–40, 2010.

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

[17] M. Joseph and P. Pandya. Finding response times in a real-time system. *BCS Computer Journal*, 29(5):390–395, 1986.

[18] H. Li and S. Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *Proceedings of the Real-Time Systems Symposium*, pages 183–192, San Diego, CA, 2010. IEEE Computer Society Press.

[19] 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.

[20] P. Pedro and A. Burns. Schedulability analysis for mode changes in flexible real-time systems. In *10th Euromicro Workshop on Real-Time Systems*, pages 172–179. IEEE Computer Society, 1998.

[21] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven premptive scheduling. *Journal of Real-Time Systems*, 1(3):244–264, 1989.

[22] K. Tindell, A. Burns, and A. J. Wellings. Mode changes in priority preemptive scheduled systems. In *Proceedings Real Time Systems Symposium*, pages 100–109, Phoenix, Arizona, 1992.

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

[24] A. Zabos, R. Davis, A. Burns, and M. G. Harbour. Spare capacity distribution using exact response-time analysis. In *17th International Conference on Real-Time and Network Systems*, pages 97–106, 2009.