Implementation and Evaluation of Mixed-Criticality Scheduling Approaches for Periodic Tasks

Huang-Ming Huang, Christopher Gill, Chenyang Lu
Department of Computer Science and Engineering, Washington University
St. Louis, MO, USA
{hh1, cdgill, lu}@cse.wustl.edu

Abstract—**Traditional fixed-priority scheduling analysis for periodic task sets is based on the assumption that all tasks are equally critical to the correct operation of the system. Therefore, every task has to be schedulable under the scheduling policy, and estimates of tasks' worst case execution times must be conservative in case a task runs longer than is usual. To address the significant under-utilization of a system's resources under normal operating conditions that can arise from these assumptions, three main approaches have been proposed: priority assignment, period transformation, and zero-slack scheduling. However, to date there has been no quantitative comparison of system schedulability or run-time overhead for the different approaches. In this paper, we present what is to our knowledge the first side-by-side evaluation of those approaches, for periodic mixed-criticality tasks on uniprocessor systems, under a mixed-criticality scheduling model that is common to all three approaches. To make a fair evaluation of zero-slack scheduling, we also address two previously open issues: how to accommodate execution of a task after its deadline, and how to account for previously unidentified forms of interference between mixed-criticality tasks. Our simulations show that while priority assignment and period transformation are most likely to be able to schedule a randomly selected task set, a small fraction of the task sets are schedulable only under the zero-slack approach. Our empirical evaluation demonstrates that user-space implementations of mechanisms to enforce period transformation and zero-slack scheduling can be achieved on Linux without kernel modification, with suitably low overhead for mixed-criticality real-time task sets.**

I. Introduction

Traditional fixed-priority scheduling analysis for periodic task sets assumes that all tasks are equally critical to a system's correct operation; thus, every task has to be schedulable under the scheduling policy. To meet this assumption, the estimation of worst case execution time for tasks has to be conservative in order to accommodate the special case when a task runs longer than average. Such a conservative approach can lead to under-utilization of a system under normal operating conditions.

Mixed criticality models: To address this issue, Vestal et al. [1, 2] and de Niz et al. [3, 4] have developed alternative *mixed-criticality* models for systems in which tasks are not equally critical. In the first model [1, 2], each task τ_i may have a *set* of alternative execution times $C_i(\ell)$, each having a different level of confidence ℓ. A task τ_i is also assigned a criticality level ζ_i, which corresponds to the required level of confidence for the task and is used in schedulability analysis.

The second model [3, 4] is a special case of the first one, where each task can specify only two execution times: a *normal worst case execution time* C_i^n and an *overload budget* C_i^o. Assuming all confidence and criticality levels are positive integers, with larger values indicating higher confidence and higher criticality, the execution times of a task τ_i are

$$C_i(\ell) = \begin{cases} C_i^n & \text{if } \zeta_i > \ell, \\ C_i^o & \text{otherwise.} \end{cases} \tag{1}$$

Mixed criticality enforcement: To ensure that no lower-criticality task prevents a higher-criticality task from meeting its deadline, a scheduler could use the criticality of a task directly as its scheduling priority. However, this would unnecessarily penalize lower criticality tasks when the system is not overloaded.

To improve the schedulability of lower-criticality tasks while preserving the mixed-criticality scheduling guarantee, Vestal et al. [1, 2] proposed the use of Audsley's priority assignment scheme [5] and the period transformation technique proposed by Sha et al. [6]. The *priority assignment* approach is based on two important observations: (1) the response time of a task is determined if the set of its higher priority tasks (Γ^h) is known, regardless of the relative priority ordering within Γ^h; and (2) if a task is schedulable at a given priority level, then it remains schedulable when it is assigned a higher priority. The algorithm operates in increasing priority order, at each step selecting a task and, if it is schedulable, assigning it the current priority and then moving to the next higher priority; otherwise, another task is selected at the current priority level. The algorithm terminates when all tasks are assigned priorities or when no remaining task is schedulable at the current priority.

In the *period transformation* approach, if a higher-criticality task τ_i has a longer period than a lower-criticality task τ_j, the higher-criticality task is equally sliced into n_i sections such that τ_i has a smaller transformed period and execution time. Task priorities are then assigned in rate monotonic order according to the transformed periods under the assumption (which the run-time environment may need to enforce) that τ_i can run for no more than C_i^o/n_i time units within each transformed period.

More recently, de Niz et al. [3, 4] proposed a *zero-slack scheduling* method for mixed-criticality tasks under the restricted model shown in Equation 1. Because zero-slack scheduling, priority assignment, and period transformation approaches share only that model in common, we will consider it instead of the more general model unless explicitly noted otherwise. Zero-slack scheduling is a bi-modal scheduling policy, where every task has a *normal mode* and a *critical mode*. When a task is in its normal mode, it is scheduled based on its priority (assigned by a rate-monotonic or deadline-monotonic policy). When a task τ_i is in critical mode, the scheduler will suspend all lower-criticality tasks; in other words, τ_i will steal slack from lower-criticality tasks when it is in critical mode.

Limitations of the state of the art: Despite their potential to improve schedulability of mixed-criticality task sets, there has been no practical comparison of the system schedulability or run-time implementation overhead implications for these different approaches. There are also two important issues with the zero-slack scheduling approach which must be addressed in practice to ensure that overloaded lower-criticality tasks cannot impair the schedulability of higher-criticality tasks: (1) since it is difficult to simply halt threads safely atop commonly available operating systems, the implicit assumption that real-time tasks that miss their deadlines are simply dropped rather than being allowed to continue to run must be removed; (2) a particular form of interference that is not accounted for in the previously published analyses of zero-slack scheduling also must be addressed. Finally, after these issues are addressed, efficient mechanisms to support zero-slack scheduling and period transformation atop commonly available operating systems must be implemented, and their expected overheads must be quantified empirically.

Contributions of this work: The primary contribution of this work is a practical implementation and evaluation of different mixed-criticality scheduling approaches, atop a realtime capable version of the commonly available Linux operating system. To our knowledge this is the first system implementation and comparative evaluation of different mixed-critical scheduling approaches for periodic tasks. Specifically, our contributions are four-fold.

- Extensions and improvement of the zero-slack scheduling algorithm and analysis to (i) accommodate execution of tasks beyond their deadlines and (ii) refine the calculation of zero-slack instants to account for forms of interference not considered in previously published analyses.
- Simulations of schedulability under different approaches to mixed-criticality scheduling which show that while the application of Audsley's priority assignment and period transformation are most likely to be able to schedule randomly selected mixed criticality task sets. Nevertheless, a fraction of the task sets are schedulable only under the zero-slack approach.
- User-space implementation of enforcement mechanisms for zero-slack scheduling and period transformation on Linux.
- Empirical evaluation on a Linux platform which demonstrates the efficiency and efficacy of these mechanisms. Our results show that both zero-slack scheduling and period transformation impose only 0.2% and 0.4% additional overhead respectively, which demonstrates their viability in practice.

The paper is organized as follows. Section II presents the details of zero-slack scheduling, the unresolved issues, and our improvements. Section III presents an evaluation of different approaches to mixed-criticality scheduling. Section IV describes our user-space implementation and evaluation of enforcement mechanisms for zero-slack scheduling and period transformation. Section V summarizes related work, and Section VI offers concluding remarks.

II. Zero-slack Scheduling and our improvements

A. Background

The pre-requisite for zero-slack scheduling [3] is to decide the zero-slack instant (ZSI) of each task. Each ZSI may be computed offline and then provided to the scheduler for run-time enforcement. The objective of the ZSI computation is to find the latest possible instant for a task τ_i to switch mode in order to reduce its impact on the schedulability of lower-criticality tasks while still maintaining the schedulability of τ_i. In [3], de Niz et al. detailed such an algorithm for calculating ZSIs for independent task sets on uniprocessor systems.

The algorithm starts with the worst case assumption that τ_i is executed only in critical mode. Based on this assumption, it computes the time k_i needed for a job $J_{i,1}$ of τ_i to execute up to its overload budget C_i^o under interference from its higher-criticality tasks. Let the release time of $J_{i,1}$ be time 0, and the deadline of τ_i be D_i. Then $t = D_i - k_i$ is the instant that $J_{i,1}$ can switch from the normal mode to the critical mode, so that $J_{i,1}$ will meet its deadline even when it is overloaded. However, setting the ZSI of τ_i to $D_i - k_i$ so that $J_{i,1}$ switches mode at that time may be too pessimistic. $J_{i,1}$ may have executed for a certain amount of time in normal mode, and thus the time budget in critical mode can be over-estimated. To reduce this pessimism, the algorithm finds the minimum amount (θ_i) of slack available for a task in normal mode and then deducts that slack from the overload budget. With the reduced budget in critical mode, the ZSI Z_i of τ_i then can be moved closer to the deadline. The algorithm repeats this recalculation of k_i and Z_i until no more slack is available in normal mode for τ_i.

How much slack is available for a task τ_i to be executed in normal mode is affected by the ZSIs of other tasks which may interfere with τ_i. That is, there are dependencies for ZSI calculations among tasks. To make the ZSI of a task as late as possible, the algorithm calculates the ZSIs of all tasks with an assumption of maximum interference (i.e., $\theta_i = 0$ for all τ_i), updates θ_i with each computed ZSI, re-calculates all ZSIs with the updated θ_i, and then continues until the ZSIs of all tasks converge. Since the algorithm relies on the convergence of ZSIs, [3] also provides a proof that the algorithm will converge as long as the deadline of each task is less than its period.

For convenience of presentation, we will assume that task deadlines equal their corresponding task periods and that task priorities are assigned in accordance with the rate monotonic scheduling policy in all subsequent zero-slack scheduling (ZSRM) examples.

B. Execution After a Missed Deadline

The original zero-slack scheduler [3] is based on the scheduling guarantee that if a task τ_i is admitted, it will be able to run up to its overloaded budget C_i^o within its deadline as long as no higher-criticality task is overloaded. A task is referred to as *schedulable* if it satisfies this scheduling guarantee. However, this is based on the assumption that no lower-criticality task misses its deadline or that if it does it is simply dropped rather than allowing it to execute beyond its deadline.

Table I: A two task ZSRM example

Task	C^n	C^o	Period	Criticality	Priority	ZSI
τ_1	4	5	9	2	Low	6
τ_2	2	3	5	1	High	0

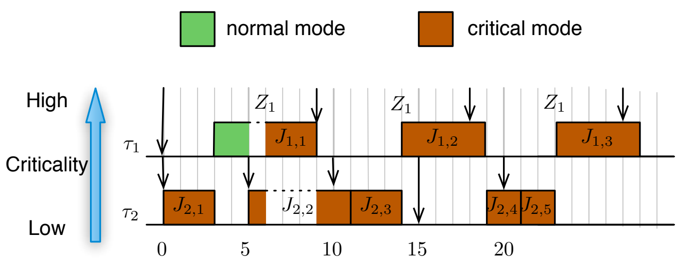


Figure 1: Zero-slack rate-monotonic scheduling of the task set in Table I

Figure 1 illustrates that, in the example task set shown in Table I, the overloading of a lower-criticality task could trigger the deadline miss of a higher-criticality task under the original zero-slack scheduling approach in [3]. In this example, all jobs of τ_1 are overloaded and run for 5 time units, and $J_{2,1}$, $J_{2,2}$, and $J_{2,3}$ are also overloaded and run for 3 time units. From Figure 1, we can see that job $J_{1,2}$ misses its deadline because the lower-criticality task τ_2 misses its deadline. Furthermore, $J_{2,3}$ also misses its deadline. In other

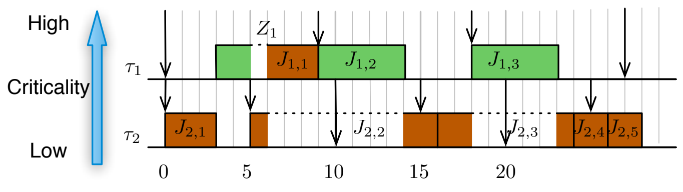


Figure 2: Illustration of zero-slack scheduling with demotion-on-deadline rule for the task set in Table I

words, $J_{1,2}$ and $J_{1,3}$ break the scheduling guarantee even though no higher-criticality task is present.

In theory, this problem can be solved by terminating a job when it misses its deadline. In practice, this may be problematic because the target job could be holding resources such as mutexes, which could lead to deadlocks and other problems. Except in special cases where jobs of the same task cannot share resources or where tasks can be made aware of their deadlines' expirations and can cooperatively release resources and halt execution, that approach is thus impractical on standard platforms.

A better approach is for the scheduler to demote task τ_i to the lowest priority when it misses its deadline; at the same time, all lower-criticality tasks have to be suspended and can be restored to their original priorities only if the job that missed its deadline terminates. τ_i can miss its deadline only if one or more higher-criticality tasks is overloaded, and thus neither τ_i nor its lower-criticality tasks are required to remain schedulable. In the scenario where more than one task misses its deadline, only the ones at the highest criticality level among them will be in the runnable state with the lowest priority level; the others will be suspended. We refer to this new scheduling rule as the *demotion-on-deadline rule*. Figure 2 shows the schedule for the task set in Table I using the demotion-on-deadline rule.

C. Unaccounted Interferences

Let π_i be the priority of task τ_i, with a larger value representing higher priority. The original analysis of ZSI calculation [3] is based on a particular worst case phasing assumption: given a job $J_{i,1}$ of task τ_i which is released at time 0, $J_{i,1}$ suffers the maximum interference while in normal mode from τ_j when the jobs of higher-or-same-priority tasks are released at time 0. In addition, the instant is also aligned to the ZSIs of the jobs from the set of tasks with *lower priority* and *higher criticality* than task τ_i, $L_i^{hc} = \{\tau_j | \pi_j < \pi_i \text{ and } \zeta_j > \zeta_i\}$. However, that formulation considers only tasks that can interfere directly with τ_i, although some tasks which cannot preempt τ_i directly may also interfere with τ_i through tasks from the set of tasks with *greater or equal priority* and *lower criticality* than task τ_i, $H_i^{lc} = \{\tau_j | \pi_j \geq \pi_i \text{ and } \zeta_j < \zeta_i\}$.

Table II: Example task set for worst case phasing condition.

	C^n	C^o	Period	Criticality	Priority	ZSI
τ_1	2	5	10	3	Med	8
τ_2	4	5	15	2	Low	9
τ_3	2	4	7	1	High	0

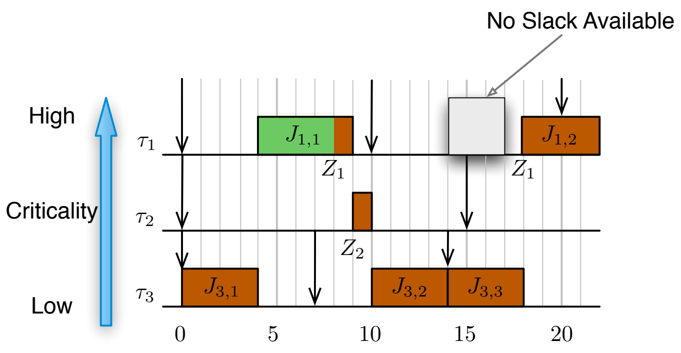


Figure 3: Zero-slack scheduling of the task set in Table II

For example, consider the task set in Table II scheduled with ZSRM. Figure 3 shows a schedule where the second and third jobs of τ_3 are overloaded and run for 3 and 5 time units respectively; in addition, the first job of τ_2 runs only for 1 time unit (valid because the specification does not require τ_2 to run for at least C_2^n time units). By the original analysis, the normal mode slack vector of τ_1 is $\{(4, 3)\}$, i.e., τ_1 can run for at least three time unit starting from time 4. However, as is shown in Figure 3, $J_{1,2}$ does not have any slack before its ZSI (at time 18). If $J_{1,2}$ is also overloaded and runs for more than 2 time units, τ_1 would miss its deadline because of the interference from lower-criticality task τ_3, and thus the scheduling guarantee would be violated.

We therefore introduce a revised ZSI calculation algorithm which addresses the previously unaccounted for interference. We define $\theta_i(\zeta_m)$ to be the minimum slack that can be used by τ_i before Z_i at criticality level ζ_m. This value is initialized to 0 for all tasks and can be increased during the ZSI calculation.

Let $C_j(\zeta_m)$ be the maximum execution time of τ_j at criticality level ζ_m, as described in Equation 1. In addition, let I_j^i be the effective execution interval of τ_j that can interfere with τ_i at criticality level ζ_m, which can be expressed by the following equation if the priority assignment is rate-monotonic or deadline-monotonic:

$$I_j^i(\zeta_m) = \begin{cases} \max(C_j(\zeta_m) - \theta_j(\zeta_m), 0) & \text{if } \tau_j \in L_i^{hc} \\ C_j(\zeta_m) & \text{otherwise.} \end{cases} \quad (2)$$

The rationale for this equation is based on the observation that when $\tau_j \in L_i^{hc}$, the minimum amount of time τ_j spends in its normal mode is θ_j and τ_j only interferes with τ_i when τ_j is in critical mode; therefore, I_j^i is $C_j(\zeta_m) - \theta_j(\zeta_m)$ (bounded by 0 if $C_j(\zeta_m) < \theta_j(\zeta_m)$).

D. *Worst Case Alignment*

To obtain θ_i, we need to know the maximum possible amount of interference τ_i can suffer from other tasks. Let $H_i^{hc} = \{\tau_j | \pi_j \geq \pi_i \text{ and } \zeta_j > \zeta_i\}$ be the set of tasks with *greater or equal priority* and *higher criticality* than task τ_i. Let $H_i^{sc} = \{\tau_j | \pi_j \geq \pi_i \text{ and } \zeta_j = \zeta_i\}$ be the set of tasks with *greater or equal priority* than, and *the same criticality* as, task τ_i. Let $\Gamma_i^n = L_i^{hc} \cup H_i^{hc} \cup H_i^{sc} \cup H_i^{lc}$ be the set of interfering tasks for task τ_i in normal mode. The ZSI calculation in [3] assumed that all release times of tasks from $\Gamma_i^n - L_i^{hc}$ are aligned to the release time of τ_i, as are the ZSIs of tasks from L_i^{hc}. However, as we show in Figure 3, this may not be the worst case phasing condition.

The key to the worst case phasing condition for zero-slack scheduling is the alignment between τ_i and the other tasks in Γ_i^n. Let $t_{i,1}^r$ be the time at which job $J_{i,1}$ is released; let $J_{j,0}$ be the last job of $\tau_j \in \Gamma_i^n$ which is released no later than $t_{i,1}^r$. To maximize the interference with τ_i from τ_j, the time when $J_{j,0}$ is able to interfere with τ_i should be no earlier than $t_{i,1}^r$ and as close to $t_{i,1}^r$ as possible. For the example in Figure 3, τ_1 suffers the maximum interference from τ_3 when the instant $t_{3,2}^b$ at which $J_{3,2}$ starts execution aligns with the release time of $J_{1,2}$, and then $J_{3,3}$ releases immediately after $J_{3,2}$ terminates. Based on this observation, in Theorem 1 we formally state the conditions for the worst case phasing that maximizes the interference that must be considered for the ZSI calculation.

Theorem 1. *Given two jobs $J_{i,1}$ and $J_{j,0}$ of tasks τ_i and τ_j respectively, let $L_i^{lc} = \{\tau_j | \pi_j < \pi_i$ and $\zeta_j < \zeta_i\}$ and $L_i^{sc} \equiv \{\tau_j | \pi_j < \pi_i$ and $\zeta_j = \zeta_i\}$. Let $t_{j,0}^r$ and $t_{j,0}^f$ be the times when $J_{j,0}$ is released and when it finishes its execution, respectively. Let $t_{j,0}^b$ be an instant between $t_{j,0}^r$ and $t_{j,0}^f$ before $J_{j,0}$ starts to execute, and let t be a time interval starting from $t_{j,0}^b$. Further, given that no task τ_k that satisfies the following two conditions is executed within the interval $[t_{j,0}^b, t_{j,0}^f]$: (1) $\tau_k \in L_i^{lc} \cup L_i^{sc}$ and $\zeta_k \geq \zeta_j$, and (2) $\tau_k \in L_i^{hc}$ and τ_k is in normal mode, then $J_{i,1}$ suffers the maximum interference from τ_j in the interval t if its release time $t_{i,1}^r$ is aligned with the time $t_{j,0}^b$.*

Proof: Illustrated in Figure 4, $J_{j,0}$ cannot be executed before $t_{j,0}^b$; therefore, $J_{i,1}$ will always suffer less interference from the subsequent jobs of $J_{j,0}$ within the interval t if $t_{i,1}^r \in [t_{j,0}^r, t_{j,0}^b)$. By definition, any task τ_k that satisfies the above two conditions cannot be executed within the interval $[t_i^r, t_i^f]$. If τ_k is executed before t_j^f, $J_{i,1}$ must have been finished; i.e., $t_i^f < t_j^f$. In this case, $J_{j,0}$ could not produce maximum interference with $J_{i,1}$. Therefore, the worst case occurs when task execution within $[t_{j,0}^b, t_{j,0}^f]$ can interfere with $J_{i,1}$, and $J_{i,1}$ will suffer less interference if $t_{i,1}^r \in (t_{j,0}^b, t_{j,0}^f]$. As a consequence, $J_{i,1}$ suffers the maximum interference from τ_j when $t_{j,0}^b = t_{i,1}^r$. ∎

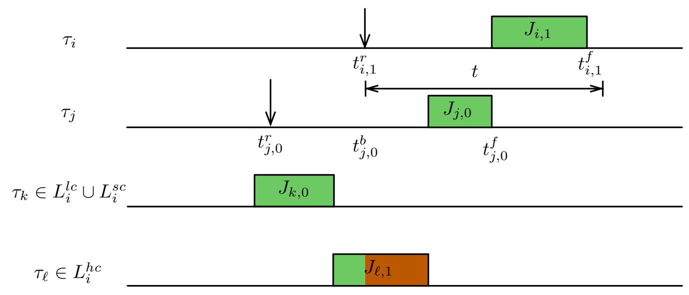


Figure 4: Illustration of Theorem 1

E. Refining the ZSI Calculation

Since rate-monotonic scheduling is a special case of deadline monotonic scheduling where deadlines of tasks are equal to their periods, we analyze the worst case phasing of zero-slack deadline monotonic scheduling instead of zero-slack rate-monotonic scheduling (while in [3] the shift from ZSRM to ZSDM involves only a single variable, the shift here is more involved, but allows a more general domain to be addressed). For simplicity of discussion, we assume that a task τ_i does not miss its deadline because even if it does the demotion-on-deadline rule discussed in Section II-B would prevent τ_i from interfering with higher-criticality tasks.

To explore the interference relationships among tasks, we bound the total time demand that can be generated by a task set as a whole. For this purpose we introduce $\delta^n_i(\zeta_m, \tau_j, t)$, the total amount of time demand generated by jobs of τ_j after a job J_i of τ_i is released and before J_i enters critical mode at criticality level ζ_m. Similarly, let $\delta^c_i(\zeta_m, \tau_j, t)$ be the total amount of time demand generated by jobs of τ_j after a job J_i of τ_i enters the critical mode and before J_i finishes execution at criticality level ζ_m.

We then define the interference function $\Delta^n_i(\zeta_m, \Gamma, t)$ ($\Delta^c_i(\zeta_m, \Gamma, t)$), which represents the maximum amount of time demand generated by a task set $\Gamma \subset \Gamma^n_i$ ($\Gamma \subset \Gamma^c_i = L^{hc}_i \cup H^{hc}_i \cup H^{sc}_i$) at criticality level ζ_m during an interval of t time units after the release of a job from τ_i when the job is in the normal (critical) mode. More formally,

$$\Delta^n_i(\zeta_m, \Gamma, t) \equiv \sum_{\tau_j \in \Gamma} \delta^n_i(\zeta_m, \tau_j, t),$$
$$\Delta^c_i(\zeta_m, \Gamma, t) \equiv \sum_{\tau_j \in \Gamma} \delta^c_i(\zeta_m, \tau_j, t).$$

With such an interference function, we use the time completion function $\mathcal{K}$ to describe the minimum time duration for a job of τ_i to execute for t time units. Within the duration, only tasks in Γ can interfere with τ_i; in addition, the amount of interference from tasks in Γ is governed by Δ_i. The time completion function can be expressed by

$$\mathcal{K}(t, u, \Gamma, \Delta_i) \equiv \min \left\{ \{u\} \cup \{t' \geq t \,|\, t' = t + \Delta_i(\Gamma, t)\} \right\},$$

where u is an upper bound for the returned result.

Note that if we adopted the worst case phasing assumption from [3], the time demand functions would be

$$\delta^n_i(\zeta_m, \tau_j, t) = \delta^c_i(\zeta_m, \tau_j, t) = \left\lceil \frac{t}{T_j} \right\rceil I^i_j(\zeta_m).$$

However, that equation fails to capture the worst time demand δ^n_i for when $\tau_j \in H^{lc}_i$ (e.g., let $i = 1$ and $j = 3$ for the example in Figure 3). Similarly, that equation also fails to capture the time demand for $\delta^c_i(\zeta_m, \tau_j, t)$ when $\tau_j \in H^{hc-}_i = \{\tau_j \in H^{hc}_i \mid H^{lc}_j \cap H^{lc}_i \neq \emptyset\}$. To correct these problems, it is necessary to derive new demand functions based on Theorem 1. For brevity, we provide only the formulas obtained, and defer a more detailed explanation to the Appendix provided in [7].

$$\delta^n_i(\zeta_m, \tau_j, t) = \left(1 + \max\left(\left\lceil \frac{t - \phi^n_j(\zeta_m)}{T_j} \right\rceil, 0\right)\right) I^i_j(\zeta_m),$$
for $\tau_j \in H^{lc}_i$, and

$$\delta^c_i(\zeta_m, \tau_j, t) = \left(1 + \max\left(\left\lceil \frac{t - \phi^c_j(\zeta_m)}{T_j} \right\rceil, 0\right)\right) I^i_j(\zeta_m),$$
for $\tau_j \in H^{hc-}_i$,

where $\phi^n_j(\zeta_m)$ and $\phi^c_j(\zeta_m)$ are the worst case phasings for τ_i in normal and critical mode, respectively.

Given that the interference function returns the maximum amount of interference a task τ_i can suffer, we can use it to compute the minimum amount of slack available for a job $J_{i,1}$ of τ_i in a time interval. In addition, we are interested in the slack which starts no later than t; therefore, we define the available slack function with respect to τ_i as

$$S^n_i(t) = \max \{t' - \Delta^n_i(\zeta_i, \Gamma^n_i, t') \mid (\forall t' < t) \cup (\forall t' \geq t \text{ where } \Delta^n_i(\zeta_i, \Gamma^n_i, t') = \Delta^n_i(\zeta_i, \Gamma^n_i, t))\}. \tag{3}$$

With these interferences and available slack functions, we can compute the ZSI of a task using Algorithm 1.

Algorithm 1 GetSlackZeroInstant(τ_i)

```
1: $x \leftarrow 0$
2: repeat
3:     $x' \leftarrow x$
4:     $C^c_i \leftarrow \max(C^o_i - x\ ,\ 0)$
5:     $k \leftarrow \mathcal{K}(C^c_i, D_i, \Gamma^c_i, \Delta^c_i(\zeta_i))$
6:     $Z_i \leftarrow \max(D_i - k\ ,\ 0)$
7:     $x \leftarrow S^n_i(\Gamma^n_i, Z_i)$
8: until $x = x'$ or $Z_i = D_i$
9: return $Z_i$
```

An important property in Algorithm 1 is that the calculation of Z_i is dependent on Z_j of another task τ_j only when $\zeta_j > \zeta_i$. Based on this, we can calculate the ZSIs of a task set in decreasing criticality order. In [3], ZSIs are computed in an unspecified order, and then the algorithm keeps looping until all ZSIs of a task set are stabilized. Our algorithm improves on [3] by computing ZSIs in a deterministic order to avoid unnecessary computation, and it is guaranteed to terminate.

III. Schedulability Evaluation

Although our revised ZSI calculation algorithm corrects the unaccounted for interference issue from [3], it does so by making some task sets unschedulable. This observation motivates a broader quantitative comparison of how the different mixed-criticality scheduling approaches affect the schedulability of tasks. Under the common mixed-criticality model we consider in the evaluation presented in this section, a task is schedulable if and only if it has enough slack to finish its overload execution budget by its deadline, and no higher criticality task is overloaded. A task set is schedulable if and only if all its constituent tasks are schedulable.

We randomly generated 1000 task sets and tested the schedulability of each set under criticality monotonic (CM) scheduling, Audsley's method (AU), rate monotonic scheduling with (PT) and without (RM) period transformation, and two variations of zero-slack scheduling with our improved ZSI calculation algorithm. One variation incorporates the less constrained worst phasing condition specified by Algorithm 1 (ZN), and the other assumes all tasks start simultaneously so that the phasing assumption can be less pessimistic (ZS). Notice that the result of ZS is not derived from the algorithm in [3]; instead, the result came from simulating the dispatching of a given task set with the zero slack scheduler. We chose the comparison with ZS because we wanted to know how pessimistic the theoretical analysis based on Theorem 1 could be.

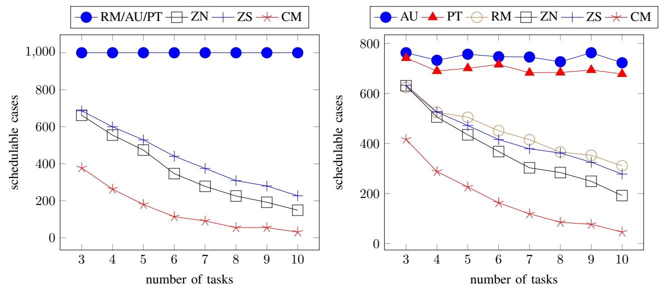
The generated task sets consisted of n (between 3 and 10) independent tasks with total CPU utilization U of 0.8 or 0.9. In our evaluation, we used both harmonic and non-harmonic task sets. For harmonic task sets, the period of a task was $2^k \times 1000$, where k was a normally distributed random integer between 0 and 5. For non-harmonic task sets, the period of a task was $k \times 2000$, where k was a normally distributed random integer number between 1 and 16. Each task τ_i was associated with a *utilization factor* u_i, a normally distributed random integer number between 1 and 1000. The worst case execution time under normal mode C_i^n of τ_i was then obtained from the equation $C_i^n = T_i \times U \times u_i / \sum_{\forall j} u_j$. The overload budget C_i^o of τ_i was always $1.2C_i^n$. The criticality level of each task was also randomly assigned.

The schedulability test we used for PT and other fixed priority schemes is a modified Joseph-Pandya worst case response time algorithm [8] in which the interference from a higher criticality task τ_j can never exceed C_j^n. Let R_i denote the worst case response time of task τ_i. Then the PT response time of τ_i is the fixed point of

$$R_i = \sum_{\tau_j:\pi_j \geq \pi_i} \left(\left\lceil \frac{R_i}{T_j} \right\rceil C_j^n + \min\left(\left\lceil \frac{R_i \mod T_j}{T_j/n_j} \right\rceil \frac{C_j^o}{n_j}, C_j^n \right) \right), \tag{4}$$

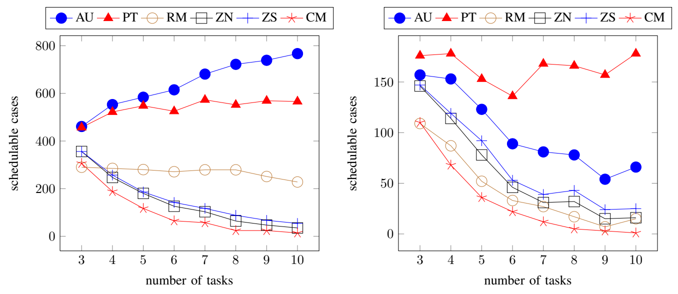
and the RM, DM, and AU response times are the fixed point of

$$R_i = \sum_{\tau_j:\pi_j \geq \pi_i} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(\zeta_j). \tag{5}$$



(a) Harmonic Task Sets (b) Nonharmonic Task Sets

Figure 5: Schedulability Evaluation with Utilization 0.8



(a) Harmonic Task Sets (b) Nonharmonic Task Sets

Figure 6: Schedulability Evaluation with Utilization 0.9

Figures 5 and 6 show the number of schedulable harmonic and non-harmonic task sets with the different methods when CPU utilization is 0.8 or 0.9 for 1000 randomly generated task sets. From the simulation results, we can see that criticality monotonic priority assignment is least likely to make a mixed criticality task set schedulable. Although ZN is better than CM, ZN is even worse than RM, except for the non-harmonic task set with 0.9 CPU utilization. AU and PT perform consistently better than other methods in all scenarios. The schedulability of ZS is slightly better than ZN because ZS has a more strict assumption that all tasks start simultaneously. However, both zero slack scheduling variations are sensitive to the number of tasks in the system, whereas the AU and PT approaches are not. The calculation of zero slack instants requires the use of worst case phasing assumptions for all tasks in H_i^{lc} and H_i^{hc-} as described in

Table III: Number of schedulable *non-harmonic* task sets when CPU utilization is 0.8 and each task set contains 10 tasks

AU	PT	ZS(ZN)
147	146	21(14)

AU/PT	AU/ZS(ZN)	PT/ZS(ZN)
331	56(27)	18(5)

AU/PT/ZS(ZN)
189(146)

Section II-C; as the number of tasks of a system increases, the more interference a task τ_i can suffer from H_i^{lc} or H_i^{hc-}.

Table III shows the number of task sets that are schedulable by AU, PT, and/or ZS(ZN) among the 1000 randomly generated task sets. We do not list the result for RM because schedulable tasks under RM are subsets of those under AU. Table III shows that even though AU and PT perform better than ZS overall, 21 task sets were schedulable only under ZS or ZN; 14 sets out of the 21 sets are schedulable under both ZS and ZN.

IV. User Space Implementation and Evaluation

Among the scheduling approaches we have considered for mixed-criticality real-time systems, Audsley's method and rate monotonic scheduling can be easily implemented atop thread priority mechanisms commonly provided by modern operating systems. To support period transformation, additional bandwidth-preserving server mechanisms are needed to ensure a task will not execute beyond its budget within a transformed period. Enforcement of zero-slack scheduling requires the use of additional timers to trigger mode changes and deadline miss handling. The additional overheads of period transformation and zero-slack scheduling mechanisms, compared to the fixed-priority scheduling approaches are thus of practical interest. In this section, we present a user space implementation of deferrable server and zero-slack scheduling mechanisms on Linux, and evaluate the run-time overhead of the different scheduling policies for mixed-criticality systems.

A. Zero-slack Scheduling Implementation

A task is implemented as a thread with a priority assigned in accordance with the application and platform. For Linux, valid priority levels range from 0 to 99, where 99 is the highest priority. Our zero-slack scheduling mechanisms reserve priority levels 1 and 99 for criticality enforcement purposes. Task suspension is emulated by lowering the priority of a task to 0. As a result, application tasks can use only priority levels 2 through 98. To simplify discussion, we assume that the criticality levels assigned to a task set are contiguous positive integers.

Each task is associated with three periodic timers, for the job release, ZSI, and deadline. Expiration of the job release timer is received and handled in the task's associated thread. An additional enforcement thread with priority 99 is created to wait for all other timer expiration events as well as for job termination events, and to make scheduling decisions based on the events it receives. To handle task suspension and resumption correctly, the enforcement thread maintains a binary heap of criticality levels. The top element of the heap has the highest criticality among the tasks that have been suspended. For convenience, we use ζ^{s0} and ζ^{s1} to denote the criticality of the top two elements in the binary heap.

A suspension event with a criticality level is used to trigger the enforcement thread to suspend a subset of tasks. When the enforcement thread receives an suspension event with criticality x, it suspends the tasks whose criticality is less than or equal to x and higher than ζ^{s0}. In addition, criticality x is inserted into the binary heap. Notice that x could be smaller than ζ^{s0} and hence no tasks would be suspended. However, the new value of x should still be inserted into the binary heap so that the enforcement thread can keep track of which tasks are in critical mode.

When the ZSI timer of a job $J_{i,k}$ expires and the job has not finished its execution, a suspension event with criticality $\zeta_i - 1$ is sent to the enforcement thread. Deadline timer expiration of a task τ_i is handled in the same way as ZSI timer expiration, except that if $J_{i,k}$ misses its deadline, a suspension event with criticality ζ_i is sent instead. When a job $J_{i,k}$ finishes while in critical mode or after missing its deadline, an event is sent to the enforcement thread to wake up the tasks that were suspended. When the event is received, the enforcement thread restores the priority of each task τ_j where $\zeta^{s1} < \zeta_j \leq \zeta^{s0}$, and then the top element of the binary heap is removed. Priority restoration is done in non-increasing criticality order. When an awakened job $J_{i,k}$ has already missed its deadline, the priority of τ_j is changed to 1 instead of π_j. In addition, ζ_j is inserted into the binary heap so that tasks with criticality levels less than or equal to ζ_j will remain suspended.

B. Period Transformation Implementation

To support period transformation, we also implemented a *deferrable server* enforcement mechanism in user space. In the deferrable server approach, a task with a transformed period is executed within a server thread. Each server has a period, a budget, and a priority, all of which are assigned according to the transformation mechanism described in Section III. The server budget is replenished at the beginning of each period. The budget decreases while the server is executing a task and is preserved (until the end of the current period) while the server is idle. A server can execute its respective task as long as its budget has not been exhausted.

Similar to our zero-slack scheduling implementation, a manager thread at highest priority is allocated to manipulate the consumption and replenishment of servers' budgets. This thread sits in an `epoll_wait` system call and waits for the budget replenishment and exhaustion events which are generated by the POSIX real-time timer APIs. For the budget replenishment events, we use timers with the `CLOCK_MONOTONIC` clock id (wall clock timer) to generate asynchronous timeout signals. To monitor the budget consumption of a server, timers with the `CLOCK_THREAD_CPUTIME_ID` clock id (thread CPU timer) are used. However, in the implementation of our test platforms, relying on the thread CPU timer to trigger budget exhaustion events is imprecise because the timer expiration can be triggered only right after the quantum expiration. In our platform, the quantum duration is 1 ms. That is, if a thread CPU timer expires 100 μs after the periodic quantum expiration time, the expiration event of the thread CPU timer timer would have to to wait another 900 μs to be triggered by the kernel. On the other hand, wall clock timers can always be triggered with microsecond level precision, regardless of the periodic quantum expiration.

For a system with only a few servers, imprecise triggering might not be a significant problem. However, such jitter can aggregate as the number of servers grows. To overcome this limitation, we utilize both thread CPU timers and wall clock timers to generate budget exhaustion events of a server. Whenever a budget replenish event arrives, we set the priority of the server thread to its respective real-time priority and reset the corresponding thread CPU timer. At the same time, a wall clock timer is set up to generate asynchronous signals based on the remaining time on the thread CPU timer. Upon expiration of the wall clock timer, the corresponding thread CPU timer is checked to see if the budget has been exhausted. If the budget is not exhausted, the wall clock timer is armed again with the remaining time read from the thread CPU timer. If the budget is exhausted, the priority of the server thread is set to the lowest priority, 0.

C. Empirical Evaluation

To measure the overhead imposed by these scheduling mechanisms, we conducted experiments on a testbed consisting of a 6-core Intel core7 980 3.3GHZ CPU with hyper-threading enabled, running Ubuntu Linux 10.04 with the 2.6.33-29-realtime kernel which incorporates the Linux RT-Preempt patch [9]. To avoid task migration among cores, CPU affinity was assigned so that our test program was executed in one particular core. All hardware IRQs except those associated with timers were assigned to cores other than the one for application execution. Each task was implemented with a `for` loop with a fixed number of iterations, where every 31 iterations yielded a 1 μs workload. In each iteration, the CPU timestamp counter was read and then compared with the counter read from the previous iteration. If the difference was greater than a specified number of ticks (700), we considered the thread to have been preempted and the new timestamp counter was stored. After a specified amount of time, all stored timestamp counters were written to a file, and then the test program terminated.

Table IV: A two task example (in ms)

Task	C^n	C^o	T	ζ	π	Z
τ_h	4	6	10	2	Low	6
τ_ℓ	2	3	5	1	High	0

ZSRM Measurements: Our first experiment used the task set in Table IV, where τ_h and τ_ℓ ran workloads of 4 and 2 ms within their periods, respectively. We found that for approximately every 1000 μs, there was a 3 μs interval that was not used by the task set or by the enforcement thread, which we attributed to the fact that the Linux scheduler was invoked at every quantum, which was about 1 ms on our test machine.

As was mentioned earlier, the enforcement thread can be invoked by job termination as well as by ZSI timers and deadline timers. In this experiment, τ_h did not have a deadline timer because there was no higher-criticality task with which it could interfere if it missed its deadline. Similarly, τ_l did not have a ZSI timer because there was no lower-criticality task that it needed to suspend when it entered critical mode.

When a timer expires or a job terminates, there is an overhead to switch from the current task thread to the enforcement thread. Depending on the scheduling context, the enforcement thread may demote or promote the priorities of some tasks and then return to the task with the highest priority. Thus the overhead of every enforcement thread invocation is the sum of the overheads for preemption invocation, thread priority adjustment, and preemption return.

Comparison between Scheduling Policies: To evaluate the cost of criticality enforcement in our ZSRM and deferrable server implementations, we compared the response times seen for rate monotonic scheduling (RM), ZSRM and periodic transformation (PT), according to the busy intervals from each invocation of τ_l until the CPU again became idle. The average busy intervals for RM, ZSRM, and PT were 7742, 7757, and 7773 μs respectively. That is to say, in this example, ZSRM scheduling only added 0.2% overhead compared to rate-monotonic scheduling, while period transformation added about 0.4% overhead. The difference between ZSRM and PT was mostly because the schedule under PT involved more context switches than the schedule under ZSRM.

Zero-slack Scheduling Micro Benchmarks: To better understand the overheads of each individual segment of enforcement thread invocation and execution, we developed another test case to measure them when there are different numbers of tasks in the system. In this test case, the ZSIs of

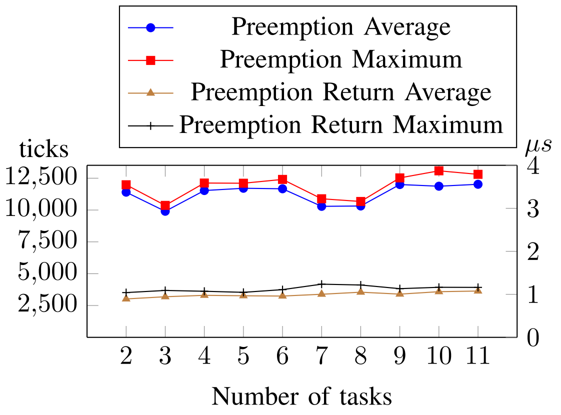


Figure 7: Preemption overhead for Linux RT kernel

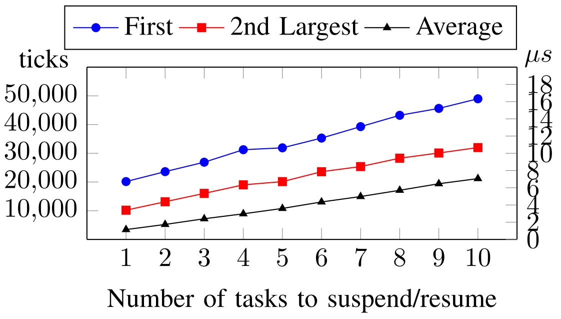


Figure 8: The cost of priority adjustment

tasks were assigned artificially so that overheads could be easily identified and measured rather than using Algorithm 1. The rationale for this artificial assignment is that those overheads are related to the number of threads and the operation performed during mode switching, rather than to the exact instants when they take place. All tasks were set to have 2 ms execution time and had the same period, 50 ms. The lowest priority task τ_0 was assigned to the highest criticality level with $Z_0 = 1$ ms. The rest of the tasks were assigned in such a way that the priority of a task was equal to its criticality level. By varying the number of tasks in the system, we obtained the overhead of preemption, preemption return, and priority adjustment, as shown in Figures 7 and 8.

As is shown in Figure 7, the overheads of preemption and preemption return are not linked to the number of tasks in the system and are about 4 μs and 1 μs respectively.

From Figure 8, we can clearly see that the cost of priority adjustment is linear with regard to the number of the threads to be promoted/demoted. However, in our experiment, the overheads for the first invocations of each such adjustment were always far higher than the rest. As a result, we present the cost of first invocations separately from the others, in the curve labeled "First". The curve of labeled "2nd Largest" and the curve labeled "Average" show the maximum and mean (respectively) of the rest of the invocations. We observe that the second largest overhead is consistently 2 to 3 μs longer than the average, which occurs when the periodic invocation of Linux scheduling occurs during the priority adjustment.

Based on these results, we can easily estimate the overhead of timer expiration or job termination. For example, the cost of a ZSI timer expiration with 8 tasks to suspend is about $4 + 5 + 1 = 10$ μs.

Deferrable Server Micro Benchmarks: Similar to our zero-slack implementation, the overhead incurred by our deferrable server implementation can also be divided into three parts: thread preemption, preemption return, and manager thread handling. In our experiments, the preemption overhead and preemption return overhead for the deferrable server implementation were very close to what is shown in Figure 7; therefore, we omit those details for brevity. As was mentioned previously, the manager thread is responsible for budget replenishment and exhaustion and for adjusting server thread priorities, as as well as for canceling the budget exhaustion timer when a job finishes. Regardless of the different functionalities involved, the average and maximum response times of each manager thread invocation were about 3587 and 22448 cycles, respectively, or about 1 and 6.6 μs, respectively.

V. Related Work

In recent years, multiple papers have been published related to mixed-criticality scheduling. Vestal [1] first proposed a formal model for representing real-time mixed-criticality tasks to support analysis of the safety of software systems based on the RTCA DO-178B software standard. In [1], he used fixed-priority scheduling and provided a preliminary evaluation using three real world mixed-criticality workloads which showed that priority assignment [5] and period transformation [6] improved the utilization of the system, in comparison to deadline monotonic analysis. Baruah and Vestal [2] then studied fundamental scheduling-theoretic issues with fixed task-priority, fixed job-priority, and earliest deadline first (EDF) scheduling policies, under Vestal's model. Later, Baruah et al. also proposed the use of priority assignment on a per-job basis using Audsley's approach offline assuming the complete ordering of jobs was known a priori [10] or on-line for sporadic workloads [11].

Anderson et al. [12] developed an extension of Linux to support mixed criticality scheduling on multi-core platforms, using a bandwidth reservation server to ensure temporal isolation among tasks with different criticalities. Tasks of the same criticality are executed in one *container* with a predefined period and budget. Intra-container task scheduling for high criticality tasks uses a cyclic executive approach where scheduling decisions are statically pre-determined offline and specified in a dispatching table, whereas EDF can be used for low criticality containers.

Pellizzoni et al. [13] also used a reservation-based approach to ensure strong isolation guarantees for applications with different criticalities. Rather than emphasizing CPU scheduling policies, this work focused on the methodology and tools for generating software wrappers for hardware components that enforce at run-time the required behavior.

VI. Conclusions

In this paper, we have presented what is to our knowledge the first practical side-by-side evaluation of mixed-criticality real-time task scheduling based on priority assignment, period transformation, and zero-slack scheduling. We also have offered refinements to zero-slack scheduling and the calculation of zero-slack instants. In particular, we have characterized a scenario in which a deadline miss of a lower-criticality task could affect scheduling guarantees for a higher criticality task, and provide a simple priority demotion rule to address that problem. We also propose a new worst case phasing condition for zero-slack scheduling and show its correctness, provide an analysis of how much interference a task can suffer from other tasks, and develop a new algorithm for calculating zero-slack instants based on that analysis.

We conducted simulations to examine how the different mixed-criticality scheduling methods may impact task schedulability, which showed that although the Audsley's priority assignment and period transformation approaches were most likely to be able to schedule randomly selected task sets, the zero-slack approach was able to schedule some task sets that the others could not. We also implemented mechanisms to support zero-slack scheduling and period transformation atop Linux without modifying the kernel. Our empirical evaluation of those mechanisms showed that, compared to fixed-priority scheduling, they imposed only 0.2% and 0.4% additional overhead respectively, which demonstrates their viability in practice.

The results of this research suggest that it may be productive to extend our evaluation to consider different features of the more general mixed-criticality model presented in [1, 2]. For example, it seems worthwhile to examine how allowing tasks to specify greater or lesser numbers of execution times impacts schedulability and run-time performance, and to consider whether those results might motivate a generalization of the zero-slack scheduling approach to support multiple scheduling mode changes under the more general mixed criticality model.

Acknowledgment

This research was supported in part by NSF grant CNS-0834755.

References

[1] S. Vestal, "Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance," in *Proc. Real-Time Systems Symposium*. IEEE Press, Dec. 2007, pp. 239–243.

[2] S. Baruah and S. Vestal, "Schedulability Analysis of Sporadic Tasks with Multiple Criticality Specifications," in *Proc. Euromicro Conference on Real-Time Systems*. IEEE Press, Jul. 2008, pp. 147–155.

[3] D. de Niz, K. Lakshmanan, and R. Rajkumar, "On the Scheduling of Mixed-Criticality Real-Time Task Sets," in *Proc. Real-Time Systems Symposium*. IEEE Press, Dec. 2009, pp. 291–300.

[4] K. Lakshmanan, D. de Niz, R. Rajkumar, and G. Moreno, "Resource Allocation in Distributed Mixed-Criticality Cyber-Physical Systems," in *Proc. International Conference on Distributed Computing Systems*. IEEE Press, 2010, pp. 169–178.

[5] N. Audsley, "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times," University of York, York, Tech. Rep. November, 1991.

[6] L. Sha, J. P. Lehoczky, and R. Rajkumar, "Solutions for Some Practical Problems," in *Proc. Real-Time Systems Symposium*. IEEE Press, 1986, pp. 181–191.

[7] H.-m. Huang, C. Gill, and C. Lu, "Implementation and Evaluation of Mixed-Criticality Scheduling Approaches for Periodic Tasks," Department of Computer Science and Engineering, Washington University in St. Louis, MO, USA, Tech. Rep. WUCSE-2011-89, 2011. [Online]. Available: http://cse.wustl.edu/Research/Lists/Technical%20Reports/Attachments/964/mc_tech_report2.pdf

[8] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, May 1986.

[9] S. Rostedt and D. V. Hart, "Internals of the RT Patch," in *Proc. Linux symposium*, 2007.

[10] S. Baruah, H. Li, and C. Hill, "Towards the design of certifiable mixed-criticality systems," in *Proc. Real-Time and Embedded Technology and Applications Symposium*. IEEE Press, 2010, pp. 13–22.

[11] H. Li and S. Baruah, "An algorithm for scheduling certifiable mixed-criticality sporadic task systems," in *Proc. Real-Time Systems Symposium*. IEEE Press, 2010, pp. 183–192.

[12] J. H. Anderson, S. Baruah, and B. B. Brandenburg, "Multicore operating-system support for mixed criticality," in *Proc. Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.

[13] R. Pellizzoni, P. Meredith, M.-Y. Nam, M. Sun, M. Caccamo, and L. Sha, "Handling mixed-criticality in SoC-based real-time embedded systems," in *Proc. seventh ACM international conference on Embedded software - EMSOFT '09*. New York, New York, USA: ACM Press, 2009, p. 235.