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

HUANG-MING HUANG, CHRISTOPHER GILL, and CHENYANG LU,
Washington University in St. Louis

Traditional fixed-priority scheduling analysis for periodic and sporadic 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 chosen 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 underutilization of a system's resources under normal operating conditions that can arise from these assumptions, several *mixed-criticality scheduling* approaches have been proposed. However, to date, there have been few quantitative comparisons of system schedulability or runtime overhead for the different approaches.

In this article, we present a side-by-side implementation and evaluation of the known mixed-criticality scheduling approaches, for periodic and sporadic mixed-criticality tasks on uniprocessor systems, under a mixed-criticality scheduling model that is common to all these approaches. To make a fair evaluation of mixed-criticality scheduling, we also address previously open issues and propose modifications to improve particular approaches. Our empirical evaluations demonstrate that user-space implementations of mechanisms to enforce different mixed-criticality scheduling approaches can be achieved atop Linux without kernel modification, with reasonably low (but in some cases nontrivial) overhead for mixed-criticality real-time task sets.

Categories and Subject Descriptors: C.4.1 [**Operating Systems**]: Process Management

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Real-time systems, mixed-criticality scheduling

ACM Reference Format:
Huang-Ming Huang, Christopher Gill, and Chenyang Lu. 2014. Implementation and evaluation of mixed-criticality scheduling approaches for sporadic tasks. ACM Trans. Embedd. Comput. Syst. 13, 4s, Article 126 (March 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/2584612

1. INTRODUCTION

Traditional fixed-priority scheduling analysis 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 the 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 underutilization of a system under normal operating conditions.

Mixed-Criticality Models. To address this issue, Vestal et al. [Vestal 2007; Baruah and Vestal 2008] and de Niz et al. [de Niz et al. 2009; Lakshmanan et al. 2010] have developed alternative *mixed-criticality* models for systems in which tasks are not equally

This work was supported in part by the US National Science Foundation, under grant CNS-0834755.
Authors' address: Department of Computer Science and Engineering, Washington University Campus Box 1045, One Brookings Drive, St. Louis, MO 63130; email: {cdgill, lu}@wustl.edu; huangming.huang@gmail.com.

DOI: http://dx.doi.org/10.1145/2584612

critical. In the first model [Vestal 2007; Baruah and Vestal 2008], 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 [de Niz et al. 2009; Lakshmanan et al. 2010] 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 the *mixed-criticality scheduling guarantee*, which is 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. [Vestal 2007; Baruah and Vestal 2008] proposed the use of Audsley's priority assignment scheme [Audsley 1991] and the period transformation technique proposed by Sha et al. [1986]. This *priority assignment* approach is based on two important observations: (1) the response time of a task τ_i is determined if the set of its higher priority tasks H_i is known, regardless of the relative priority ordering within H_i; 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 runtime 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. [de Niz et al. 2009; Lakshmanan et al. 2010] proposed a *zero-slack scheduling* method for mixed-criticality tasks under the restricted model shown in Eq. (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.

The primary contribution of this work is a comprehensive practical implementation and evaluation of known mixed-criticality scheduling approaches, atop a real-time capable version of the commonly available Linux operating system. Our previous work [Huang et al. 2012], which to our knowledge provided the first side-by-side implementation (and overhead and schedulability evaluations) of multiple mixed-criticality approaches, considered *fixed task-priority static mixed criticality* (FTP-SMC), *period*

transformation, and *zero-slack scheduling* approaches. This article extends that comparison through the design, implementation, and overhead evaluation of user-space enforcement mechanisms, along with side-by-side schedulability evaluations, to include recently developed techniques based on *fixed task-priority adaptive mixed-criticality* (FTP-AMC) [Baruah and Burns 2011] and *fixed job-priority* [Li and Baruah 2010a; Guan et al. 2011] approaches. As the schedulability evaluations presented in this article demonstrate, those new approaches can provide substantial improvements in task schedulability over the mixed-criticality scheduling approaches we investigated previously.

To further strengthen our runtime performance evaluation of the different mixed-criticality scheduling approaches, and to assess and demonstrate their applicability in practice, in this article we also present a new scalability evaluation for cases involving up to 32 tasks. Our evaluations of enforcement mechanisms for the different mixed-criticality scheduling approaches demonstrate that a wide-range of mixed-criticality scheduling policies can be realized efficiently and effectively in user space atop Linux. Our results show that the overheads for the different mixed-criticality scheduling approaches correlate strongly with the number of thread switches in their schedules. FTP-SMC and period transformation showed the least and most runtime overheads respectively, among the mixed-criticality scheduling approaches we evaluated. However, fixed job-priority scheduling imposed no more than an additional 4% overhead compared to FTP-SMC, which may make its use suitable for many real-time applications.

To provide more relevant schedulability comparisons, we also present revisions to the schedulability analysis for FTP-AMC [Baruah and Burns 2011] and extend it for systems with more than two criticality levels. Our schedulability evaluations demonstrate that our revised analysis can provide a tighter schedulability bound. Also for purposes of comparative evaluation, we present a new method to estimate the worst case job arrival pattern for the *priority list reuse scheduling* (PLRS) [Guan et al. 2011] fixed job-priority approach.

2. MIXED-CRITICALITY SCHEDULING

In this section, we discuss the mixed criticality scheduling of periodic and sporadic tasks on uniprocessor systems. First, we introduce a classification of existing mixed-criticality scheduling approaches on uniprocessor systems. Second, we summarize enforcement mechanisms for different categories of mixed-criticality scheduling approaches. Third, we compare and contrast fixed task-priority, fixed job-priority, and dynamic priority approaches.

2.1. Classification of Mixed-Criticality Scheduling Approaches

In recent years, several algorithms and approaches have been published which focus on the scheduling of mixed-criticality uniprocessor systems. In this section, we classify these approaches according to their runtime enforcement mechanisms and means of priority assignment. Runtime enforcement mechanisms determine how long a job can execute and when a job should be aborted or terminate. Priority assignment determines the precedence of execution among jobs which are eligible to run at any given time. Notice that runtime enforcement mechanisms and priority assignment are not always orthogonal, as some runtime enforcement mechanisms can only be coupled with certain priority assignment strategies.

2.2. Runtime Enforcement Mechanisms

—*Zero-Slack Scheduling (ZSS).* Each task has a fixed zero-slack instant which is offset by a time interval relative to the release time of a job and is not associated with the runtime progression of the job. A job which has not signaled completion after its zero

slack instant is said to be in *critical mode*. Whenever a job enters its critical mode, all lower criticality tasks should be suspended.

—*Period Transformation (PT).* The periods of some tasks are equally divided into several segments. The execution time of a task within a period is also equally distributed into the segments to become the execution budgets of the task in the segment.

—*Static Mixed Criticality (SMC).* All jobs of a task τ_i can execute up to their representative execution time $C_i(\zeta_i)$, but are prevented from executing further.

—*Adaptive Mixed Criticality (AMC).* There is a system-wide criticality-level indicator, initialized to 0. Whenever the criticality-level indicator is κ, and there exists a task τ_i which executes up to $C_i(\kappa)$ time units without signaling completion, the criticality-level indicator is increased to $\kappa + 1$ and all tasks whose criticality levels are less than or equal to κ are prevented from executing.

2.3. Priority Assignment

The purpose of priority assignment is to determine which job has precedence when more than one job is eligible for execution. According to Baruah and Vestal [2008], priority assignment algorithms for a periodic or sporadic mixed-criticality system can be categorized into three different classes: fixed task priority (FTP), fixed job-priority (FJP), and dynamic priority.

Fixed Task-Priority (FTP). For fixed task-priority assignment, all jobs of the same task are statically assigned the same priority. The most basic strategies for FTP are rate monotonic (RM) and criticality monotonic (CM) which assign higher priority to higher rate or higher criticality tasks, respectively. Other mixed-criticality scheduling algorithms are also designed to assume a specific priority assignment strategy. For example, zero-slack scheduling is coupled with rate monotonic assignment; period transformation chooses criticality monotonic assignment but transforms the task periods so that it can behave in a rate-monotonic fashion. Traditionally, in rate-monotonic scheduling (RMS) and criticality-monotonic scheduling (CMS), their respective priority assignment strategy is coupled with a runtime enforcement mechanism that allows jobs to execute up until they finish. In this article, we will follow this tradition (i.e., RMS=RM+SMC, CMS=CM+SMC) for the convenience of our discussion.

Another notable strategy is *Audsley's priority assignment method*, which as we noted in Section 1 is based on two important observations: (1) the response time of a task τ_i is determined if the set of its higher priority tasks (H_i) is known, regardless of the relative priority ordering within H_i; 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.

Audsley's approach can be coupled with SMC or AMC to improve schedulability over RMS or CMS. However, Audsley's approach requires scheduling analysis to check whether a task can be schedulable at a given priority. Given a task τ_i, a scheduling analysis is used to calculate the response time R_i of τ_i. If R_i is smaller than the respective deadline D_i of τ_i, then τ_i is considered schedulable.

For the rest of this article, *fixed task-priorty* (FTP) will specifically refer to Audsley's priority assignment approach unless otherwise specified.

For CMS, RMS and FTP-SMC, the response time R_i^* of a task τ_i is τ_i's worst-case execution time $C_i(\zeta_i)$ plus the ζ_i confidence level time demand for H_i; that is,

$$R_i^* = C_i(\zeta_i) + \sum_{\tau_j \in H_i} \left\lceil \frac{R_i^*}{T_j} \right\rceil C_j(\zeta_i). \tag{2}$$

As for period transformation, the response time of τ_i is the fixed point of

$$R_i = \sum_{\tau_j:\pi_j \geq \pi_i} \left(\left\lceil \frac{R_i}{T_j} \right\rceil C_j(\zeta_i) + \min \left(\left\lceil \frac{R_i \mod T_j}{T_j/n_j} \right\rceil \frac{C_j(\zeta_j)}{n_j}, C_j(\zeta_i) \right) \right). \quad (3)$$

To determine whether a task is schedulable, we then can simply check if the computed response time of a task is smaller than its deadline. In contrast, the response-time analyses for ZSS and FTP-AMC are somewhat more complicated.

The ZSS approach was proposed by de Niz et al. [2009] in an effort to overcome the limitations of the DM and PT approaches. In Huang et al. [2012], we described two concerns with that scheduling method and analysis: (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; and (2) given a task τ_i, the assumption that the adjacent jobs of all higher priority tasks of τ_i can always maintain minimum separation based on their period arrival pattern is invalid due to the influence of the zero-slack instant of τ_i. In Huang et al. [2012], we showed how the first issue can be addressed atop operating systems like Linux, by simply demoting a job to the lowest priority when it misses its deadline, and how the second issue can be addressed by modifying the zero-slack instant calculation to consider the worst-case phasing of tasks. In all of the evaluations of ZSS presented in this article, both of those refinements were applied. Because our evaluations show that even with our improvements the zero-slack scheduling approach under-performs other approaches in most cases, for brevity we omit a more detailed explanation of those issues and our approaches to resolving them in this article, and instead refer the interested reader to our previous paper [Huang et al. 2012].

Baruah et al. [2011] provided two sufficient analysis methods for a two-criticality FTP-AMC system. In Section 3.1, we generalize that analysis to systems with more than two criticality levels. In addition, we refine the second method presented in that paper to provide a tighter bound than the original.

Fixed Job-Priority (FJP). A fixed-job priority strategy assigns priorities to each job of a system regardless whether two jobs are of the same task. Since jobs of the same task may have different priorities, at least part of the priority assignment needs to be deferred to runtime instead of binding it statically as in FTP. On the other hand, fixed job-priority will dominate fixed task priority in terms of theoretical schedulability because every FTP assignment is also an FJP assignment [Baruah and Vestal 2008]. However, given a task which is FTP schedulable, whether an FJP-specific assignment algorithm can always find a feasible job-priority assignment for that task without resolving FTP analysis is yet another important question. We discuss FJP further in Section 3.2.

Dynamic Priority. In *dynamic priority* strategies, the priority of a job may change between its release time and completion time. Strictly speaking, all the previously discussed mixed-criticality scheduling algorithms except RMS, CMS, and FTP-SMC can be considered to be dynamic priority according to that definition, because those algorithms require the scheduler to abort and/or suspend jobs before the jobs' completion under certain conditions. For the sake of discussion, however, we exclude job abort/suspend as a form of priority change.

EDF has been proved to be optimal for uniprocessor single-criticality systems. That is, a task set which is unschedulable under EDF will not be schedulable under any other scheduling policies for uniprocessor systems. However, that property does not hold for mixed-criticality systems. Baruah and Vestal [2008] proved that fixed task-priority

(FTP) and dynamic priority (DP) approaches are incomparable because there exist task sets which are FTP-schedulable (or FJP-schedulable) but not DP-schedulable, and vice-versa. In addition, they proved a task is DP-schedulable if and only if it is schedulable under the traditional EDF analysis. That is, the execution-time specification of different confidence levels cannot help schedulability as it does for FTP and FJP. Therefore, we will not further discuss dynamic priority approaches.

3. ANALYSIS OF ADAPTIVE MIXED-CRITICALITY APPROACHES

3.1. Fixed Task-Priority AMC Scheduling (FTP-AMC)

In zero-slack scheduling, priority assignment is based on task rate and the time a task switches mode (i.e., the zero slack instant) relative to the release time of a job. In adaptive mixed criticality (AMC) scheduling, on the other hand, priority assignment is based on Audsley's algorithm and the mode change is triggered by the execution progress of tasks. Baruah et al. [2011] discussed two versions of response-time analysis for FTP-AMC: AMC-RT and AMC-MAX.

The AMC-RT analysis has a potential to over-estimate because it assume all jobs of τ_i execute for $C_i(\zeta_i)$ time units, even when the criticality-level indicator is less than ζ_i. AMC-MAX, on the other hand, is difficult to extend directly to a system with more than two criticality levels. In this section, we further extend those analyses and consider systems with more than two criticality levels.

To simplify our discussion, we begin with a system with only two criticality levels (0 and 1). Let s be the last job deadline before a criticality change[1]; that is, all jobs $J_{i,j}$ whose deadlines are before s can only execute for $C_i(0)$ time units. The number of job releases for a low criticality task τ_j is thus bounded by

$$n_j(s) = \left\lceil \frac{s}{T_j} \right\rceil.$$

Since any job of a higher criticality task τ_k whose deadline is before s can execute only for $C_k(0)$ time units, we can bound the number of job releases $n_k(s)$ for τ_k that executes in low criticality mode by

$$n_k(s) = \max\left(\left\lfloor \frac{s - D_k}{T_k} \right\rfloor + 1, 0\right).$$

We use $floor + 1$ instead of ceiling because the job of τ_k which releases at exactly $s - D_k$ can only be executed for $C_k(0)$ time units based on our assumptions about s, and thus it should be counted in $n_k(s)$. Given a task $\tau_k \in H_i^{\zeta \geq \zeta_i}$ where $H_i^{\zeta \geq \zeta_i}$ is the set of tasks with priority greater than and criticality greater than or equal to τ_i, the maximum number of job releases in high criticality mode within a busy period of R is thus bounded by $(\lceil \frac{R_i^s}{T_k} \rceil - n_k(s))$. Therefore, the response time R_i^s of τ_i is the fixed point of

$$R_i^s = C_i(0) + \sum_{\tau_j \in H_i} n_j(s) C_j(0) + \sum_{\tau_k \in H_i^{\zeta \geq \zeta_i}} \left(\left\lceil \frac{R_i^s}{T_k} \right\rceil - n_k(s) \right) C_k(1).$$

Since s represents the deadline of a job, the release time of the job is bounded by $R_i(0)$; we can then enumerate all possible values of s to obtain the maximum response

[1]In Baruah et al. [2011], s is initially defined as the time point when the system criticality-level indicator changes from 0 to 1. However, in the latter part of the article, it assumes all jobs released before and at s only execute in low criticality mode.

time of τ_i by

$$R_i^* = \max\left\{R_i^s \mid \forall \tau_j, \quad s \in \left[D_j, \ldots, \left\lceil \frac{R_i(0)}{T_j} \right\rceil T_j + D_j\right]\right\}.$$

Consider an example task set comprised of three tasks as follows.

	ζ_i	$C_i(0)$	$C_i(1)$	D_i	T_i
τ_1	0	1	1	2	2
τ_2	1	1	5	10	10
τ_3	1	20	20	100	100

Suppose there is no criticality change: the response time of τ_3 can be computed by

$$R_3(0) = 20 + \left\lceil \frac{R_3(0)}{2} \right\rceil + \left\lceil \frac{R_3(0)}{10} \right\rceil,$$

which has solution 50. The deadlines of jobs between (0, 50] are the even numbers between 2 and 50. Each of these values for s needs to be checked. The worst case occurs when $s = 48$.

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

which has solution 58.

Now, consider the case where the number of criticality levels of the system is greater than 2. Let s_ℓ, referred as the *criticality change point*, be the last job deadline before the system criticality level indicator changes from $\ell - 1$ to ℓ where s_0 is defined as 0. Let $\mathbb{S} = \{s_0, s_1, \ldots\}$ be a sequence of criticality change points for a busy period of a mixed-criticality task schedule. Let $n_j^\ell(\mathbb{S})$ be the number of τ_j job releases for which the jobs execute at criticality level no larger than ℓ. The number of job releases within the criticality level ℓ is thus

$$n_j^\ell(\mathbb{S}) = \begin{cases} 0 & \text{if } \ell < 0, \\ n_j^{\ell-1} & \text{else if } \zeta_j < \ell, \\ \left\lceil \frac{S_\ell}{T_j} \right\rceil & \text{else if } \zeta_j = \ell, \\ \max\left(\left\lfloor \frac{s_\ell - D_j}{T_j} \right\rfloor + 1, 0\right) & \text{otherwise.} \end{cases}$$

Let $I^{\mathbb{S}}(t, m, \Gamma)$ be the time demand from a task set Γ within a busy interval of t, given that the system criticality change is governed by $\mathbb{S}$ and the system criticality level indicator does not exceed m in (or before) that interval. $I^{\mathbb{S}}(t, m, \Gamma)$ can be represented by

$$I^{\mathbb{S}}(t, m, \Gamma) = \sum_{\ell=0}^{m-1} \left\{ \sum_{\tau_j \in \Gamma} \left(n_j^\ell(\mathbb{S}) - n_j^{\ell-1}(\mathbb{S})\right) C_j(\ell) \right\} + \sum_{\tau_k \in \Gamma^{\zeta \geq m}} \left(\left\lceil \frac{R_i^{\mathbb{S}}(m)}{T_k} \right\rceil - n_k^{m-1}(\mathbb{S}) \right) C_k(m), \quad (4)$$

where $\Gamma^{\zeta \geq m} = \{\tau_j \in \Gamma \mid \zeta_j \geq m\}$.

Assuming that $D_i \leq T_i$, the response time $R_i^S(m)$ of τ_i if the system criticality-level indicator is no larger than m is thus

$$R_i^{\mathbb{S}}(m) = C_i(m) + I^{\mathbb{S}}\left(R_i^{\mathbb{S}}(m), m, H_i\right).$$

The relationships between the elements in $\mathbb{S}$ can be represented by

$$s_\ell \in \left(s_{\ell-1}, R_i^{\mathbb{S}}(\ell-1) + D_{max}\right),$$

where D_{max} represents the maximum relative deadline for the tasks in H_i. Then the maximum response time of τ_i is

$$R_i^* = \max\left\{R_i^{\mathbb{S}}(\zeta_i) \mid \forall\, \mathbb{S}\right\}.$$

3.2. Fixed Job-Priority AMC Scheduling (FJP-AMC)

Baruah et al. [2010] developed an algorithm called *own criticality-based priority* (OCBP), for mixed-criticality systems comprised of a finite number of (nonrecurring) jobs with an AMC scheduling policy. The job priority assignment algorithm is also based on Audsley's approach. To be more precise, within each step, the algorithm selects a job J_i whose deadline is longer than the total time demand for the remaining jobs $\mathbb{J}$ and assigns J_i the lowest priority among $\mathbb{J}$. Next, J_i is removed from $\mathbb{J}$ and the process continues until no job is available for priority assignment.

Li and Baruah [2010a] further developed an online job-priority assignment algorithm for scheduling sporadic tasks based on OCBP. The algorithm starts by assuming all tasks are released at the same time and assigns job priorities according to OCBP. Whenever a new job $J_{i,j}$ arrives, the online scheduler compares the previously assigned priority $\zeta_{i,j}$ of $J_{i,j}$ and the priority $\zeta_{k,\ell}$ of the current running job $\tau_{k,\ell}$. If $\zeta_{i,j} > \zeta_{k,\ell}$, the priorities of all jobs in $\{J_{i,j} \mid \zeta_{i,j} > \zeta_{k,\ell}\}$ are recomputed using OCBP based on the worst possible job-arrival pattern starting from the $\tau_{k,\ell}$ arrival time. Li also proved that no priority recomputation is needed if the newly arrived job $\tau_{k,\ell}$ has a lower priority than the current running job. However, the complexity of the online priority recomputation is pseudopolynomial in the worst case. This makes the algorithm less acceptable for systems which require stringent worst-case bounds on analysis complexity.

To reduce the runtime complexity of Li's algorithm, Guan et al. [2011] presented an algorithm called *priority list reuse scheduling* (PLRS). Instead of relying entirely on online priority recomputation for job scheduling, PLRS separates the algorithm into two stages: an offline priority computation and an online priority assignment stage. The PLRS offline priority computation is essentially the same as the online priority recomputation of Li's method, but PLRS only computes jobs of the worst-case busy period when all tasks are released at the same time. The resulting information is then used by a lighter weight online algorithm for priority assignment. The complexity of the offline stage is still pseudopolynomial but that of the online stage is reduced to be quadratic in the number of tasks.

However, both Li's and Guan's algorithms require a worst-case job-arrival pattern for job priority calculation; that is, how many jobs (along with execution criticalities) of each task can be executed in a busy period. To obtain the job-arrival pattern, the longest busy period and the distribution with each criticality level needs to be bounded. Li and Baruah [2010a] offered a busy period bound based on the *load*[2] of each criticality level. However, in our experiments, we found that the bound may be too loose to be useful, and so it is essential to have a tighter busy period bound for FJP-AMC-based algorithms.

We therefore develop a new analysis to find a job-arrival pattern for FJP-AMC job-priority calculation. Like the FTP-AMC analysis, given a task set Γ, the longest busy

[2]The load of a collection of jobs denotes the maximum execution requirements over all time intervals, normalized by the interval length.

period without any criticality change is the fixed point of

$$B(0) = \sum_{\tau_i \in \Gamma} \left\lceil \frac{B(0)}{T_i} \right\rceil C_i(0).$$

We can also express it in terms of $I^{\mathbb{S}}$ from Eq. (4) by

$$B(0) = I^{\{0\}}(B(0), 0, \Gamma),$$

where $n_k^{-1} = 0$ for all k.

For a task τ_i of criticality 0, the number of jobs n_i^0 is bounded by $\lceil B(0)/T_i \rceil$. To extend the busy period, the system criticality-level indicator cannot be changed before the last job $J_{i,j}^{\zeta=0}$ of criticality 0 finishes. The earliest time for $J_{i,j}^{\zeta=0}$ to finish is

$$s^1 = \max_{\tau_i \in \Gamma^{\zeta=0}} \left\{ \left\lfloor \frac{B(0)}{T_i} \right\rfloor T_i + C_i(0) \right\}.$$

Assuming any job $J_{i,j}$ released before s_1 can only execute for $C_i(0)$ time units, then the maximum number n_i^0 of jobs for τ_i to be executed for $C_i(0)$ time units in a busy period is

$$n_i^0 = \left\lceil \frac{s^1}{T_i} \right\rceil.$$

To generalize to a task set with m criticality levels, we get

$$B(m) = I^{\mathbb{S}}\left(B(m), s^m, \Gamma\right),$$

$$s^m = \max_{\tau_i \in \Gamma^{\zeta=m-1}} \left\{ \left\lfloor \frac{B(m-1)}{T_i} \right\rfloor T_i + C_i(m-1) \right\},$$

$$n_i^m = \begin{cases} 0 & \text{if } m < 0, \\ n_i^{m-1} & \text{if } \zeta_i < m, \\ \left\lceil \frac{s^m}{T_i} \right\rceil & \text{if } \zeta_i \geq m. \end{cases}$$

Consider an example with 4 tasks.

	ζ_i	$C_i(0)$	$C_i(1)$	D_i	T_i
τ_1	1	4	6	10	10
τ_2	1	3	5	20	20
τ_3	0	6	6	30	30
τ_4	0	2	2	15	15

Assuming that there is no criticality change, the criticality 0 busy period would be

$$B(0) = \left\lceil \frac{B(0)}{10} \right\rceil 4 + \left\lceil \frac{B(0)}{20} \right\rceil 3 + \left\lceil \frac{B(0)}{30} \right\rceil 6 + \left\lceil \frac{B(0)}{15} \right\rceil 2,$$

which has solution 28. That is, for any busy period of the task set, τ_3 and τ_4 can release at most 1 and 2 jobs, respectively; that is, $n_3^0 = 1$ and $n_4^0 = 2$. For a busy period in which τ_3 and τ_4 release exactly 1 and 2 jobs, the system criticality-level indicator cannot change from 0 to 1 before 17 time units after the beginning of the busy period because 17 is the earliest possible time for the second job of τ_4 to finish.

For the high-criticality task τ_1, its first job $J_{1,1}$ should execute only for $C_1(0)$ time units in order to maximize the busy period because its deadline is before 17. The

question is whether $J_{1,2}$ and $J_{2,1}$ should execute for their worst-case execution times $C_1(1)$ and $C_2(1)$.

Assume those two jobs run only for $C_1(0)$ and $C_2(0)$. Then, the number of jobs for τ_1 and τ_2 to run in confidence level 0 are 2 and 1, respectively; that is, $n_1^0 = 2$ and $n_2^0 = 1$. Thus, we can calculate the busy period when τ_1 and τ_2 execute for their worst-case execution time after their second and first job, respectively.

$$B(1) = 2 \times 4 + 3 + 6 + 2 \times 2 + \left(\left\lceil \frac{B(1)}{10} \right\rceil - 2 \right) 6 + \left(\left\lceil \frac{B(1)}{20} \right\rceil - 1 \right) 5,$$

which has solution 38. Consequently, $n_1^1 = \lceil 38/10 \rceil = 4$ and $n_2^1 = \lceil 38/20 \rceil = 2$.

With the job arrival pattern, we use OCBP to assign job priorities as follows.

τ_1	7,5,2,0
τ_2	6,1
τ_3	3
τ_4	8,4

Now let us consider the case where $J_{1,2}$ and/or $J_{2,1}$ execute for their worst-case execution times under this job-priority assignment. Since both $J_{1,2}$ and $J_{2,1}$ have higher priority (5 and 6) than $J_{3,1}$ and $J_{4,2}$ (3 and 4), both $J_{3,1}$ and $J_{4,2}$ would be aborted or suspended at the moment $J_{1,2}$ and $J_{2,1}$ execute up to their low-criticality execution time and the system is still considered schedulable.

On the other hand, if we assume $J_{1,2}$ and/or $J_{2,1}$ can execute for their worst-case execution time; that is, $n_1^1 = \lfloor s^1/T_1 \rfloor = 1$ and $n_2^1 = \lfloor s^1/T_2 \rfloor = 0$, $B(1)$ would become 59. Even though this busy period is longer, the job-arrival pattern obtained from it is not FJP-AMC schedulable.

In general, using *floor* instead of *ceiling* for n_i^m when $\zeta_i \geq m$ is not necessary to produce an FJP-AMC unschedulable job-arrival pattern. However, if a job-arrival pattern derived from $n_i^m = \lfloor s^m/T_i \rfloor$ is FJP-AMC schedulable, the job-arrival pattern derived from $n_i^m = \lceil s^m/T_i \rceil$ for the same task set is always schedulable because of the lesser time demand of the latter. Therefore, using $n_i^m = \lceil s^m/T_i \rceil$ when $\zeta_i \geq m$ is a sufficient schedulability test. However, using *floor* may allow more jobs to be admitted at runtime. For PLRS, where job priorities are computed offline, it is better to use the floor version unless it produces an unschedulable job-arrival pattern.

4. SCHEDULABILITY EVALUATION FOR MIXED-CRITICALITY TASKS ON UNIPROCESSOR SYSTEMS

In this section, we present our investigation of the schedulability of different mixed-criticality scheduling approaches and demonstrate the effectiveness of our improved analyses over their original counterparts.

4.1. Task Set Generation Parameters

We studied the schedulability of randomly generated task sets, where each task set Γ_k was generated based on the following parameters.

—The CPU utilization U_k of Γ_k is a number between 0.05 and 0.95. Given a U_k, the utilization u_i of a task τ_i in Γ_k was generated by the UUnifast algorithm [Bini and Buttazzo 2005].

—The period T_i of a task τ_i was $100x$ where x was randomly sampled from a uniform distribution between 1 to 100, and the deadline D_i was the same as the period T_i.

—The criticality levels of tasks were assigned sequentially as tasks were generated; that is, if the total number of criticality levels in a task set was N, then the criticality level ζ_i of the ith generated task τ_i was $i \bmod N$.
—The execution time $C_i(0)$ of τ_i was obtained from the equation $C_i(0) = \max(\lfloor T_i u_i \rfloor, 1)$.
—The default *criticality factor* (CF) ratio between the worst-case execution time $C_i(\zeta_i)$ and $C_i(0)$ was 1.5. Given a CF, the execution-time specification of τ_i was

$$C_i(\zeta) = \begin{cases} C_i(0) & \text{if } \zeta < \zeta_i, \\ CF \times C_i(0) & \text{otherwise.} \end{cases}$$

—For each set of parameters, 1000 tasks were generated.

4.2. Scheduling Approaches and Analysis Investigated

The scheduling approaches and analysis we used are as follows.

—Rate-monotonic scheduling (RMS) with response-time analysis based on Formula 2
—Criticality-monotonic scheduling (CMS) with response-time analysis based on Formula 2
—Period transformation (PT) with response-time analysis based on Formula 3
—Zero-slack scheduling (ZSS) with improved analysis described in Huang et al. [2012]
—Static Mixed Criticality (FTP-SMC), priority assigned with Audsley's approach with response time analysis based on Formula 2
—FTP-AMC with the first analysis (AMC-RT) developed by Baruah et al. [2011]
—FTP-AMC with the second analysis (AMC-MAX) developed by Baruah et al. [2011]
—FTP-AMC with our improved analysis (AMC-IA) as described in Section 3.1
—FJP-AMC with job arrival pattern developed by Li and Baruah [2010b] (FJP-LB)
—FJP-AMC with our improved job-arrival pattern, as described in Section 3.2 (FJP-IJA)

4.3. Simulation

For convenience of presentation, we separate our evaluation into three different parts. The first part compares the effectiveness of different FTP-AMC analyses as well as FTP-SMC. The second part evaluates FJP-AMC with two different methods of job-arrival pattern calculation. Last, we compare how different scheduling approaches affect the schedulability of mixed-criticality tasks.

FTP-SMC/FTP-AMC Comparison. Figure 1 plots the result with 20 tasks in each task set (NT=20), 2 levels of criticality (NC=2), and the ratio between $C_i(\zeta_i)$ and $C_i(0)$ is 1.5 (i.e., $CF = C_i(\zeta_i)/C_i(0) = 1.5$). We conducted experiments with the total utilization of a task set varying from 0.05 to 0.95. However, we only show the utilization ranges from 0.55 to 0.88 because the ratio of schedulable tasks is either 1 or 0 outside that range.

As shown in Figure 1, AMC-based analyses performed better in schedulability tests because of the mechanism that lower criticality tasks have to be suspended/aborted when the execution of a higher criticality task exceeds certain execution thresholds. The differences between all three AMC analyses are very small; however, AMC-IA consistently performs better than AMC-MAX and AMC-RT. Beside the marginal schedulability improvement of AMC-IA over AMC-MAX, the AMC-IA analysis can be easily extended beyond two levels of criticality, which is not the case for AMC-MAX.

Figure 2 further demonstrates the slightly better performance for AMC-IA over AMC-MAX and AMC-RT in schedulability tests. In Figure 2, we show the weighted schedulability measure [Bastoni and Brandenburg 2010; Baruah et al. 2011] over varying numbers of tasks and CF ratios, respectively. For each parameter p (number

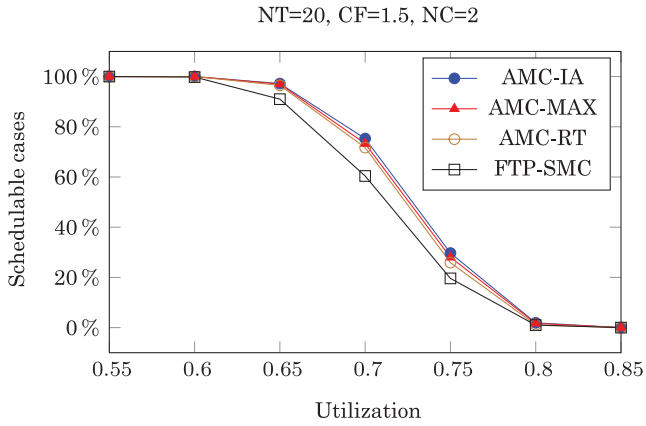


Fig. 1. Schedulability evaluation for FTP SMC/AMC with 20 tasks.

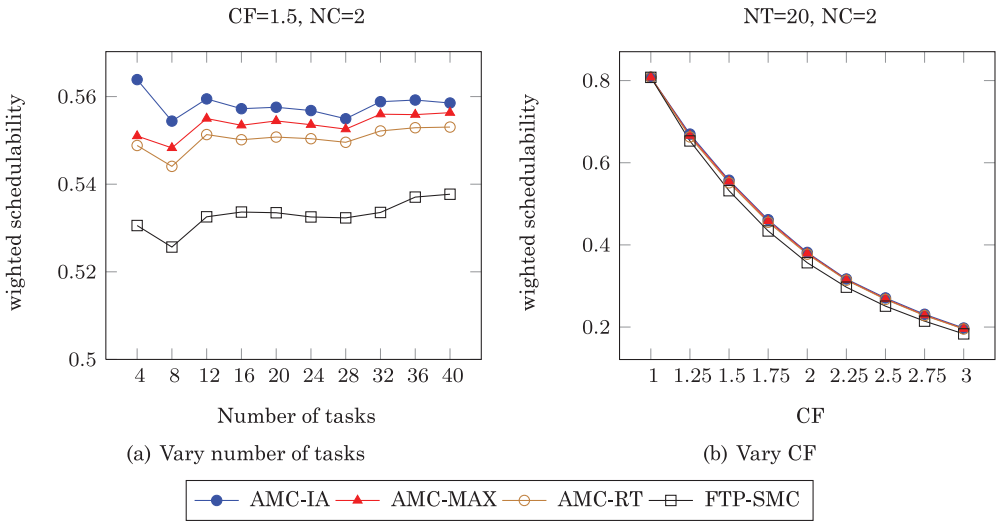


Fig. 2. Schedulability evaluation for FTP SMC/AMC with varying parameters.

of tasks or CF), the weighted schedulability measure combines results for all the task sets Γ generated for all of a set of equally spaced utilization levels (0.05 to 0.95 in steps of 0.05).

Given a total CPU utilization u_i, let $\pi_i(p)$ be the percentage of schedulable cases (out of 1000) with parameter p. The weighted schedulability measure $W(p)$ is defined as

$$W(p) = \left(\sum_{\forall i} u_i \pi_i(p) \right) \Big/ \sum_{\forall i} u_i .$$

The weighted schedulability measure reduces what would otherwise be a 3-dimensional representation to 2 dimensions. Weighting the individual schedulability results by task set utilization reflects the higher value placed on being able to schedule higher utilization task sets [Bastoni and Brandenburg 2010; Baruah et al. 2011].

Besides the slight performance gain in AMC-IA, Figure 2(a) also shows that all the weighted schedulability curves over the number of tasks are relatively flat which indicates neither the schedulability of AMC nor that of SMC is sensitive to the number

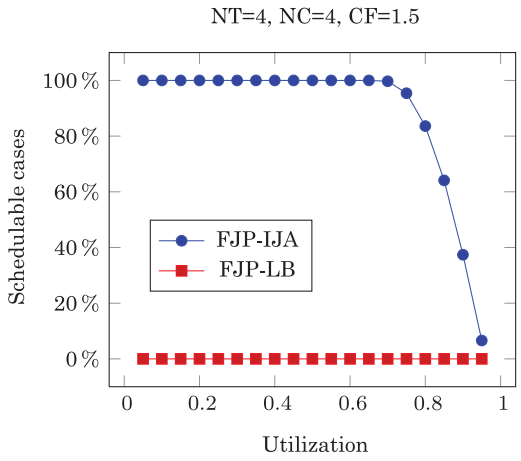


Fig. 3. Schedulability evaluation for fixed-job priority analyses.

of tasks in the system. On the other hand, because the CPU utilization of a generated task τ_i is based on the execution time at the lowest confidence level, $C_i(0)$, it is to be expected that the schedulability decreases as the CF ratio increases as is shown in Figure 2(b).

Comparison of FJP Analyses. As we described in Section 3.2, assigning job priorities to mixed-criticality sporadic tasks was proposed by Li and Baruah [2010a, 2010b]. Before priorities can be assigned to jobs, it is necessary to estimate job arrival patterns (i.e., the number of jobs for each task that can be executed in each criticality level) in a worst-case busy period. Li and Baruah provided an approach for job-arrival pattern estimation based on the *load* of sporadic tasks. However, that approach is difficult to apply in practice because of two reasons. First, the *load* calculation is computationally expensive [Fisher et al. 2006]. In our preliminary tests, we were unable to finish testing schedulability of even 10 randomly generated task sets (with the parameters NT=8, U=0.05, CF=1.5) in a day. Second, the success rate for generating task sets that can pass the schedulability test based on this approach is often very poor. In fact, none of the generated task sets passed the schedulability test with the following parameters: NT=4, NC=4, CF=1.5. Figure 3 illustrates the benefit of using our revised job-arrival pattern (FJP-IJA) calculation described in Section 3.2, in comparson to Li and Baruah's original method [Li and Baruah 2010b].

Comparison of Different MC Scheduling Approaches. Because AMC-IA and FJP-IJA perform best for their respective scheduling approaches, we will use them for the comparison of their representative analyses against other approaches. Figure 4(a) shows the percentage of schedulable task sets where each task set consists of 20 tasks, 4 criticality levels and a CF ratio of 1.5 under different scheduling approaches. Figure 4(b) compares the approaches by varying the number of tasks in each task set. Figure 4(c) compares the approaches by varying CF. Figure 4(d) compares the approaches by varying the number of criticalities in a task set.

The following observations are illustrated by these figures.

—CMS performs very badly in terms of schedulability.
—AMC-IA performs better than ZSS because the time to suspend/abort lower criticality tasks is determined at runtime rather being statically precomputed. This allows AMC-IA to be more adaptive than ZSS based on the workload.

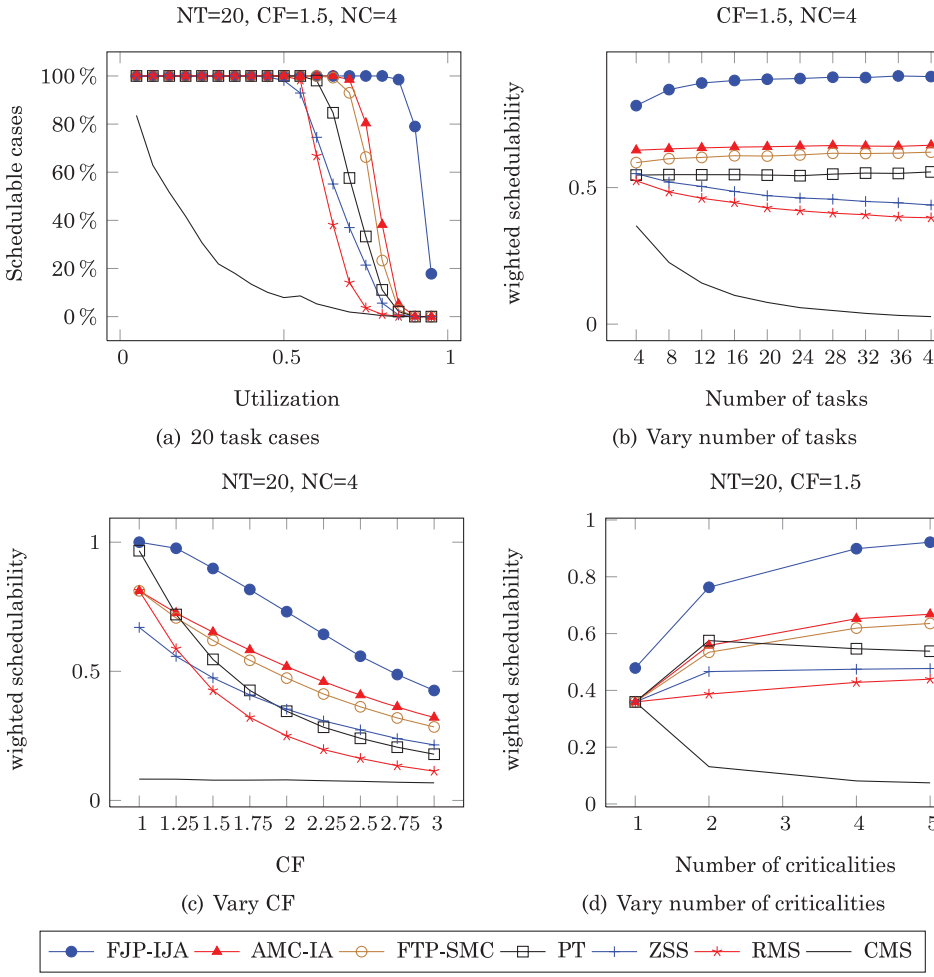


Fig. 4. Schedulability evaluation for MC scheduling approaches.

—For the approaches based on Audsley's priority assignment algorithm (AMC-IA or FTP-SMC), the rate of successful schedulable tasks increases as the number of tasks increases. This is largely due to the finer granularity for schedulable job/task selection given a fixed CPU utilization. This factor is more prominent when the number of tasks is small.

—Except for CMS and PT, the number of successfully schedulable tasks increases as the number of criticality levels increases. This is because the finer the granularity of the criticality levels, the more tasks a high-criticality task can suspend or abort, which thus increases its schedulability. However, we also observed an increase in the number of criticality levels also significantly increases the computation time for the schedulability tests, especially for AMC-IA and FJP-IJA. This is because the computational complexity of the AMC-IA and FJP-IJA schedulability tests depends on the length of longest busy period, and the busy period increases as the number of criticality levels increases.

Overall, the key observation is that FJP-IJA performs best in all scenarios. Given the fact that FJP-IJA selects schedulable jobs instead of tasks in each step of Audsley's priority assignment algorithm, it is not surprising that FJP-IJA would outperform AMC-IA because of the finer granularity. In theory, FJP dominates FTP because any FTP priority assignment is also a valid FJP assignment [Baruah and Vestal 2008]. However, this does not translate to the necessity that FJP-IJA dominates FTP-IJA. This discrepancy comes from the fact that the job-arrival patterns used by FJP-IJA are different from those of FTP-IJA. If an FJP algorithm chooses to test all possible job-arrival patterns, it would dominate all possible FTP algorithms; however, the complexity of doing that would likely be too high in practice.

5. MIXED-CRITICALITY RUNTIME ENFORCEMENT MECHANISM IMPLEMENTATION AND EVALUATION

Among the scheduling approaches, we have considered for mixed-criticality real-time systems, FTP-SMC, rate-monotonic and criticality-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 [Liu 2000] are needed to ensure a task will not execute beyond its budget within a transformed period. Enforcement of zero-slack and AMC 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 FTP-SMC approaches are thus of practical interest. In this section, we present a user space implementation of deferrable server, zero-slack scheduling and AMC mechanisms atop Linux, and evaluate the runtime overhead of the different scheduling policies for mixed-criticality systems.

5.1. 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 a 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 nonincreasing 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.

5.2. Period Transformation Implementation

To support period transformation, we also implemented a *deferrable server* enforcement mechanism. 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 4. 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 scheduling quantum expiration. In our platform, the quantum duration is 1 ms. That is, if a thread CPU timer expires 100 μs after the quantum expiration time, the expiration event of the thread CPU 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 quantum expiration period.

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 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.

5.3. AMC Implementation

Our AMC scheduling policy enforcement implementation also relies on mechanisms used to enforce the zero-slack scheduling and period transformation approaches. In the implementations of all these scheduling approaches, a manager thread running at

highest priority performs common scheduling control actions. Like period transformation, AMC requires both wall clock and thread CPU timers to monitor the progress of a task. Similar to the expiration of a ZSI timer, the AMC implementation must suspend lower criticality tasks once the execution of a task exceeds the execution specification for a particular certain confidence level.

However, the availability of only 99 priority levels in Linux limits the usefulness of this implementation for use with FJP-AMC scheduling without any modification. FJP-AMC assigns a unique priority level to every single job in the job-arrival pattern. The highest priority is reserved for the manager thread and the lowest priority is reserved for suspended tasks, so that only 97 priority levels are left for job assignments. In the 1000 task sets we generated based on the method in Section 4 (with parameters CF=1.5, NC=4, NT=20, U=0.8), on average FJP-IJA had around 115 distinct jobs in the job-arrival patterns, making the availability of only 97 distinct priority levels inadequate.

To overcome this limitation, we implemented an alternate way to enforce job priorities when the number of distinct priority levels provided by the operating system is not sufficient. Instead of mapping job priorities directly to native OS priorities, our alternative implementation only uses three native OS priority levels (namely, `MGR_PRIO`, `RUN_PRIO`, and `SUSPENDED_PRIO` in decreasing priority order). The manager thread (running at the `MGR_PRIO` priority) works as a job dispatcher to control when jobs are released, preempted, and suspended. At any given moment, only the thread of the current running job with highest FJP priority is assigned the `RUN_PRIO` priority; the other threads are set to the `SUSPENDED_PRIO` priority.

Besides job-priority enforcement, FJP-AMC also requires a runtime mechanism to compute the priorities of jobs. To avoid runtime job-priority computation of pseudopolynomial complexity described in Li and Baruah [2010a], we implemented PLRS [Guan et al. 2011] which off-loads the pseudopolynomial complexity to an offline stage and employs an online algorithm of quadratic complexity.

5.4. 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 hyperthreading enabled, running Ubuntu Linux 10.04 with the 2.6.33-29-realtime kernel with the Linux RT-Preempt patch [Rostedt and Hart 2007]. To avoid task migration among cores, CPU affinity was assigned so the 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.

ZSS Measurements. Our first experiment used the task set in Table I, where τ_h and τ_ℓ ran workloads of 4 and 2 ms within their periods, respectively. We found that approximately every 1000 μs (1 ms), there was a 3μs interval that was not used by the task set or by the enforcement thread, which we attribute to the invocation of the Linux scheduler in every scheduling quantum (set to 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

Table I. A Two-Task Example (in *ms*)

Task	$C_i(0)$	$C(\zeta_i)$	T_i	ζ_i	π_i	Z_i
τ_h	4	6	10	1	Low	6
τ_ℓ	2	3	5	0	High	0

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 of Scheduling Policies. To evaluate the cost of criticality enforcement through different scheduling approaches, we generated a random task set and benchmarked subsets of it with between 4 to 32 tasks (in increments of 4 tasks). The generated task set had 4 criticality levels, a hyper period of 1000 ms, and was schedulable under each scheduling policy (i.e., FTP-SMC, PT, ZSS, FTP-AMC, and FJP-AMC). All subsets had equal numbers of tasks at each criticality level.

We configured the system to release all tasks' first jobs at once and measured all instances of busy periods starting from the job release time of a task set up until the CPU again became idle. Each task τ_i executed for $C_i(0)$ time units and each experiment ran for 100 seconds for each scheduling approach, so that at least 100 busy periods were observed. We chose the execution time $C_i(0)$ for every task so that the busy intervals would not vary for different scheduling approaches.

Table II shows the durations of busy periods in μs, and the overheads and number of busy period thread switches of the generated task sets. Because FTP-SMC does not require any other runtime mechanism besides the priority-based thread dispatching provided by the operating system, it serves as a baseline in our comparison. The overhead number indicates the percentage of increase in busy period duration, compared to the baseline, for the other scheduling policies of the same task set.

Based on the results shown in Table II and Figure 5, the scheduling approaches can be ordered by increasing levels of overhead as follows: FTP-SMC, zero-slack scheduling, FTP-AMC, FJP-AMC, and period transformation. We also observed that the overheads of the scheduling approaches correlate strongly with the number of thread switches they experience during the busy period.

Period transformation imposes the most overhead compared to ZSS- and AMC-based scheduling policies because it usually needs to partition the period into small pieces in order to increase schedulability, especially when the lower criticality tasks have much higher frequency than those of the higher criticality tasks.

In our zero-slack scheduling implementation, periodic timers are used for implementing the expiration of ZSI. That is, the manager thread will be triggered periodically to check if a job is finished or needs to be suspended. This explains the increased number of thread switches over FTP-SMC.

In our AMC implementations, job release and completion events must be handled by the manager thread in order to monitor the progress of jobs. Therefore, the number of thread switches is even higher than for our zero-slack scheduling implementation. In all cases, the number of thread switches for FTP-AMC and FJP-AMC are very similar, and the overheads of those two approaches also are very similar. This indicates that the overhead of FJP-AMC was dominated by the number of thread switches, and that

Table II. The Duration of Busy Periods in μs, Overheads and Number of Thread Switches in Busy Periods for Subsets of the Generated Task Set

num tasks	FTP-SMC	PT	ZSS	FTP-AMC	PLRS
4	11154 0% 2	11212 0.515% 11	11161 0.057% 3	11203 0.433% 7	11210 0.495% 7
8	14037 0% 6	14210 1.231% 37	14044 0.047% 6	14139 0.725% 15	14141 0.741% 15
12	56963 0% 28	57809 1.487% 177	56994 0.054% 33	57272 0.544% 59	57189 0.398% 59
16	77699 0% 44	78915 1.565% 294	77719 0.025% 52	78138 0.564% 91	78185 0.626% 91
20	89484 0% 58	91552 2.310% 420	89522 0.042% 67	90431 1.058% 123	90496 1.130% 123
24	97758 0% 75	112596 15.178% 578	97793 0.035% 87	98599 0.860% 153	98662 0.925% 153
28	149124 0% 156	164433 10.266% 1470	149225 0.068% 180	155069 3.987% 331	155395 4.205% 331
32	177341 0% 194	199403 12.441% 1951	178451 0.626% 286	183478 3.460% 407	183625 3.544% 405

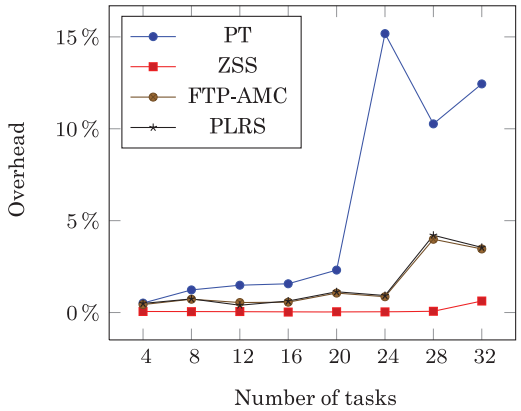
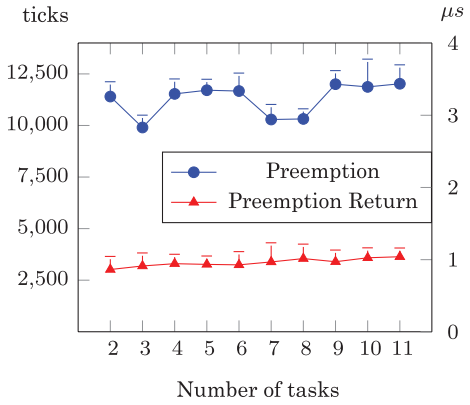


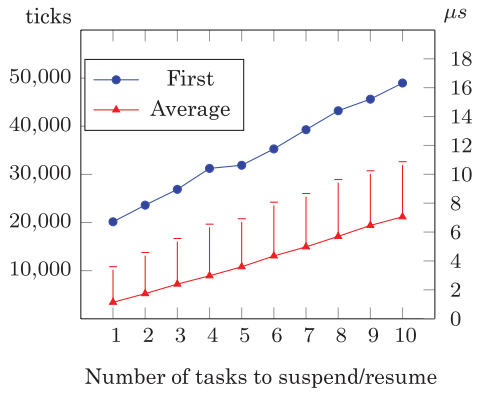
Fig. 5. Overhead comparison for different scheduling approaches.

the effect of online job priority calculation was insignificant compared to the effect of thread switches, at least for the task sets we evaluated.

Preemption Overhead Microbenchmarks. To better understand the overheads of each segment of manager thread invocation and execution, we developed another test case to measure the preemption overhead in the system. The rationale is that those overheads are related to the number of threads and the operation performed during mode switching, rather than the instants when they take place. All tasks were set to



(a) Preemption overhead, error bar represents the maximum



(b) The cost of priority adjustment, error bar represents the second largest

Fig. 6. Micro benchmark for Linux RT kernel.

have the same execution time (2 ms) and 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 6(a) and 6(b).

As is shown in Figure 6(a), 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 6(b), we can clearly see that the cost of priority adjustment is linear with regard to the number of 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 labeled "Average" and its error bars show the mean and maximum (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 the Linux scheduler 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 eight tasks to suspend is about $4 + 5 + 1 = 10$ μs.

Similar to our zero-slack scheduling implementation, the overhead incurred by our period transformation and AMC implementations 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 and AMC implementations were very close to what is shown in Figure 6(a); therefore, we omit those details for brevity.

Table III shows the average and maximum cost of manager thread invocations for our implementations of various mixed-criticality scheduling enforcing mechanisms. As was mentioned previously, the PT manager thread is responsible for budget replenishment and exhaustion and for adjusting server thread priorities, as well as for canceling the budget exhaustion timer when a job finishes. Regardless of the functionalities involved, the average and maximum response times of each manager thread invocation were about 19435 and 410068 cycles, or about 5.75 and 121.39 μs, respectively.

Compared to our PT implementation, where job release and job termination are handled in the respective task thread, the manager thread of our AMC implementation

Table III. The Average and Maximum Cost in Cycles of Manager Thread Invocation

	PT	ZSS	FTP-AMC	PLRS
Avg	19435 (5.75 μs)	6578 (1.95 μs)	21326 (6.31 μs)	24854 (7.35 μs)
Max	410068 (121.39 μs)	49844 (14.75 μs)	479408 (141.92 μs)	419732 (124.25 μs)

is also involved whenever a job is released or terminated because the AMC manager thread needs to identify the current running job and sets the wall clock timer to trigger the event for the system criticality level indicator change. As a result, AMC approaches incur more overhead than PT as is shown in Table III. For our PLRS implementation, the manager thread also needs to adjust thread priorities to maintain the correct job execution ordering, which increases the cost of manager thread processing. PLRS also requires additional job priority computations on some manager thread invocations.

In general, Table III shows that the PLRS manager thread overhead is highest among the scheduling approaches we have implemented because it involves more operations than others. On the other hand, the runtime overhead of a scheduling approach is strongly influenced by the number of preemption and task suspension and resumption events in a schedule. Because different scheduling approaches likely will produce different runtime schedules, there is no absolute guarantee which scheduling approach would result in the fewest manager thread invocations. Therefore, it is hard to know a priori which scheduling approach will impose the highest runtime overhead even though the average overhead of PLRS-MW is the highest per manager thread invocation.

Nevertheless, FTP-SMC relies only on priority, and no extra runtime mechanism for task suspension/resumption is required; thus, it is expected to have the lightest overhead in general. Therefore, we recommend that system developers consider adopting FTP-SMC if the target task set can be schedulable with that approach. For the cases where more stringent CPU utilization bounds are required, PLRS with our new worst-case arrival pattern calculation also could be a good solution.

6. SURVEY OF RELATED WORK

In recent years, multiple papers have been published related to mixed-criticality scheduling. Vestal [2007] 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. DO-178B is a software development process standard, which assigns criticality levels to tasks categorized by effects on commercial aircraft, as a means of certifying software in avionics applications. Vestal [2007] used fixed-priority scheduling and provided a preliminary evaluation using three real-world mixed-criticality workloads which showed that priority assignment [Audsley 1991] and period transformation [Sha et al. 1986] improved the utilization of the system, in comparison to deadline monotonic analysis.

Baruah and Vestal [2008] then studied fundamental scheduling-theoretic issues with fixed task-priority, fixed job-priority and earliest deadline first (EDF) scheduling policies, under Vestal's mixed-criticality model. In contrast to the traditional single-criticality task set where EDF is known to be optimal for uniprocessor scheduling in the sense that EDF always meets all deadlines for all feasible traditional systems, they showed that EDF was not optimal for mixed-criticality tasks in the uniprocessor environment. They also showed that a mixed-criticality task set is schedulable under an EDF scheduling policy if and only if it is feasible under traditional EDF scheduling

analysis where the multiple specifications of worst-case execution times are ignored and each mixed-criticality task is treated as a traditional task with the worst-case execution time of the highest criticality. In fact, fixed task-priority and EDF scheduling are incomparable because neither dominates the other; in other words, a mixed-criticality task set that is schedulable under fixed task-priority scheduling may not be schedulable under EDF scheduling, and vice-versa. To achieve better task schedulability, they proposed a hybrid-priority scheduling algorithm which dominated both fixed task-priority scheduling and EDF scheduling. This algorithm generalizes both EDF and Audsley's method such that each task is assigned a priority that is not necessarily unique. Tasks of different priorities are scheduled according their priority settings; tasks of the same priority are scheduled in EDF. The priority assignment algorithm extends Audsley's algorithm to select tasks in increasing priority order. Instead of using the modified Joseph-Pandya worst-case response time analysis [Joseph and Pandya 1986] in each task selection step, Baruah and Vestal [2008] adopted a simulation-based approach to test whether a task would fail to be schedulable at a given priority level.

Baruah et al. [2010] investigated the schedulability of mixed-criticality systems consisting of a finite number of (nonrecurring) jobs. Their analysis demonstrates that complexity of mixed-criticality schedulability testing is NP-hard even when all jobs are released at the same time and the total number of criticality levels in the system is 2. It also proves that priority-based scheduling offers a processor speedup factor between 1.236 and 2 compared to EDF when the number of criticality levels is 2 or approaches infinity, respectively.

Li and Baruah [2010a] further developed an online algorithm for scheduling sporadic tasks on a per-job basis. The algorithm is also based on Audsley's approach but with a different function to test the feasibility of a job running at a particular priority. The new function allows Li and Baruah's algorithm to be more adaptive on the arrival pattern of jobs rather than relying on the static periodic model. However, this algorithm depends on an online priority assignment recomputation with psudopolynomial complexity in the worst case. This may make the algorithm unacceptable for systems that require stringent worst-case bounds on timing behavior.

To overcome the limitations of Li and Baruah's algorithm, Guan et al. [2011] presented an algorithm called PLRS to address this issue. Like Li's algorithm, PLRS is also a fixed job-priority scheduling algorithm. However, instead of solely relying on high-complexity online priority recomputation for job scheduling, PLRS separates the algorithm into two stages: an offline priority computation and an online priority assignment stage. The PLRS offline priority computation is essentially the same as the online priority recomputation of Li and Baruah's method, but PLRS only computes jobs of the worst-case busy period when all tasks are released at the same time. The resulting information is then used by a lighter-weight online algorithm for priority assignment. The complexity of the offline stage is still psudopolynomial but that of the online stage is reduced to being quadratic in the number of tasks.

In Baruah et al. [2011] and Burns [2011], Baruah et al. developed an adaptive approach for scheduling mixed-criticality sporadic task sets on uniprocessor systems. Unlike Li's algorithm and PLRS where released jobs won't change priorities, this algorithm requires the scheduler to demote the priorities of lower criticality jobs once it detects that the execution time of the current running job J_i has exceeded its respective WCET $C_i(\zeta_i)$ for the given criticality level. Notice that this scheduler is similar to the zero-slack scheduler in that they both involve mode change behavior. However, this scheduler requires monitoring the runtime progress of jobs in order to determine the exact time instant to change mode whereas the zero-slack instant used by a zero-slack scheduler is a fixed time duration relative to the release time of a job. This approach thus requires more runtime support than zero-slack scheduling. Priority assignment

in this approach is based on Audsley's approach with an alternative task feasibility testing function which is tailored to the mode change behavior.

Anderson et al. [2009] developed an extension of Linux to support mixed-criticality scheduling on multicore 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. Intracontainer task scheduling for high-criticality tasks uses a cyclic executive approach where scheduling decisions are statically predetermined offline and specified in a dispatching table, whereas EDF can be used for low-criticality containers.

Pellizzoni et al. [2009] also used a reservation-based approach to ensure strong isolation guarantees for applications with different criticalities. This work focused on a methodology and tool for generating software wrappers for hardware components that enforce (at runtime) the required behavior rather than focusing on the CPU scheduling policies.

7. CONCLUSIONS

In previous work [Huang et al. 2012], we provided what is, to our knowledge, the first side-by-side implementation and overhead and schedulability evaluations of multiple mixed-criticality approaches, specifically those based on fixed task priority static mixed criticality (FTP-SMC), period transformation, and zero-slack scheduling, on uniprocessor systems. In this article, we have presented a more comprehensive comparison of known mixed-criticality scheduling approaches, including new implementations and evaluations of fixed task priority adaptive mixed criticality (FTP-AMC) and fixed job priority adaptive mixed criticality (FJP-AMC) scheduling approaches, under a mixed-criticality scheduling model that is common to all of the evaluated approaches.

For the fixed task priority adaptive mixed criticality approach, we present refinements to the original analysis [Baruah et al. 2011] and extended it to more than two criticality levels. Our simulation results demonstrate that our new analysis can provide tighter response-time bounds and thus may achieve schedulability for task sets not deemed schedulable by the original analysis.

We also present a new method to calculate the worst-case job arrival pattern to be used in conjunction with fixed job priority adaptive mixed-criticality scheduling approaches such as Li and Baruah's algorithm [Li and Baruah 2010b] or PLRS [Guan et al. 2011]. The original worst-case job-arrival pattern calculation method published by Li and Baruah [2010b] was based on the *load* of a task set. However, as our experiments showed, that method may perform very poorly in terms of schedulability.

Our schedulability evaluations showed that the FJP-AMC approach together with our worst-case job-arrival pattern calculation performed best overall in terms of schedulability among the mixed-criticality scheduling approaches we evaluated. However, FJP-AMC did not always dominate in terms of schedulability, that is, there were some task sets that were schedulable under other approaches but not under FJP-AMC. The evaluation of why different approaches offer better schedulability than others is also an important contrbution of this work. For example, AMC-IA often performed better than ZSS in terms of schedulability because the times at which to suspend/abort lower-criticality tasks are determined dynamically instead of statically.

To offer a more complete evaluation of mixed-criticality scheduling, this article also presents scalability and overhead comparisons of all of the approaches considered. Overall, our results demonstrate the viability of mixed-criticality scheduling enforcement via user-space mechanisms, even as the number of tasks increases. Our micro-benchmarks also showed that the runtime overhead of those mixed-criticality scheduling approaches was strongly correlated with the number of preemptions and the number of tasks to suspend or resume in a schedule.

Because different scheduling approaches likely will produce different runtime schedules, there is no absolute guarantee which scheduling approach would result in the fewest preemptions, task suspensions, or resumptions. Therefore, the design, implementation, and evaluation methods presented in this article offer exemplars for system developers to use in conducting similar comparisons for their task sets.

Rather than necessarily distinguishing or excluding any approach (except for perhaps excluding criticality-monotonic which performed very poorly in terms of schedulability), the results presented in this work thus offer relative rankings of the approaches in two dimensions that should be considered together in determining which approach to try for a given task set: (1) the overhead for runtime enforcement of an approach, and then (2) the likelihood a task set may be schedulable under that approach.

For example, because FTP-SMC relies only on priority, no other mechanisms for task suspension or resumption are required and thus it is expected to have the lightest overhead in general. Therefore, we recommend that system developers first consider adopting FTP-SMC if the target task set is schedulable with that approach. However, for cases where more stringent CPU utilization bounds are required, PLRS with our new worst-case arrival pattern calculation also could be a good solution since (even though our evaluations showed that PLRS imposed the heaviest overhead among those mechanisms we implemented) compared to fixed-priority scheduling PLRS imposed only around 4% additional overhead, which shows its potential viability.

The current PLRS scheduling approach does not offer a means to deal with dependent tasks with mutually exclusive synchronization constraints. As a future work, we are considering how to extend the worst-case arrival pattern calculation as well as the PLRS runtime scheduling algorithm to accommodate this sort of constraint. Incorporating the overhead results directly into the schedulability analyses is also a natural next step as future work.

REFERENCES

James H. Anderson, Sanjoy Baruah, and Björn B. Brandenburg. 2009. Multicore operating-system support for mixed criticality. In *Proceedings of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*.

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

Sanjoy Baruah and Alan Burns. 2011. Implementing mixed criticality systems in Ada. In *Proceedings of the Reliable Software Technologies-Ada-Europe 2011*.

Sanjoy Baruah, Alan Burns, and R. I. Davis. 2011. Response-time analysis for mixed criticality systems. In *Proceedings of the Real-Time Systems Symposium*. 34–43.

Sanjoy Baruah, Haohan Li, and Chapel Hill. 2010. Towards the design of certifiable mixed-criticality systems. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*. IEEE Press, 13–22.

Sanjoy Baruah and Steve Vestal. 2008. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Proceedings of the Euromicro Conference on Real-Time Systems*. IEEE Press, 147–155.

Andrea Bastoni and B. Brandenburg. 2010. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 33–44.

Enrico Bini and Giorgio C. Buttazzo. 2005. Measuring the performance of schedulability tests. *Real-Time Syst.* 30, 1–2, 129–154.

Alan Burns. 2011. Timing faults and mixed criticality systems. In *Dependable and Historic Computing*, Lloyd and Jones, Eds., Lecture Notes in Computer Science, vol. 6875, Springer, 147–166.

Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. 2009. On the scheduling of mixed-criticality real-time task sets. In *Proceedings of the Real-Time Systems Symposium*. IEEE Press, 291–300.

N. Fisher, T. P. Baker, and S. Baruah. 2006. Algorithms for determining the demand-based load of a sporadic task system. In *Proceedings of RTCSA*, 135–146.

Nan Guan, Pontus Ekberg, Martin Stigge, and Wang Yi. 2011. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *Proceedings of the Real-Time Systems Symposium*, 13–23.

Huang-Ming Huang, Christopher Gill, and Chenyang Lu. 2012. Implementation and evaluation of mixed-criticality scheduling approaches for periodic tasks. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, 23–32.

M. Joseph and P. Pandya. 1986. Finding response times in a real-time system. *Comput. J.* 29, 5, 390–395.

Karthik Lakshmanan, Dionisio de Niz, Ragunathan Rajkumar, and Gabriel Moreno. 2010. Resource allocation in distributed mixed-criticality cyber-physical systems. In *Proceedings of the International Conference on Distributed Computing Systems*. IEEE Press, 169–178.

Haohan Li and Sanjoy Baruah. 2010a. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *Proceedings of the Real-Time Systems Symposium*. IEEE Press, 183–192.

Haohan Li and Sanjoy Baruah. 2010b. Load-based schedulability analysis of certifiable mixed-criticality systems. In *Proceedings of the 10th ACM International Conference on Embedded Software*. ACM, New York, 99–108.

Jane W. S. W. Liu. 2000. *Real-Time Systems* 1st Ed. Prentice-Hall, Upper Saddle River, N.J.

Rodolfo Pellizzoni, Patrick Meredith, Min-Young Nam, Mu Sun, Marco Caccamo, and Lui Sha. 2009. Handling mixed-criticality in SoC-based real-time embedded systems. In *Proceedings of the 7th ACM International Conference on Embedded Software*. ACM, Press, New York, 235.

Steven Rostedt and Darren V. Hart. 2007. Internals of the RT patch. In *Proceedings of the Linux Symposium*.

Lui Sha, John P. Lehoczky, and Ragunathan Rajkumar. 1986. Solutions for some practical problems. In *Proceedings of the Real-Time Systems Symposium*. IEEE Press, 181–191.

Steve Vestal. 2007. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the Real-Time Systems Symposium*. IEEE Press, 239–243.

Received July 2012; revised February 2013; accepted March 2013