On the Scheduling of Mixed-Criticality Real-Time Task Sets

Dionisio de Niz, Karthik Lakshmanan, and Ragunathan (Raj) Rajkumar
Carnegie Mellon University, Pittsburgh, PA - 15232

***Abstract*—The functional consolidation induced by the cost-reduction trends in embedded systems can force tasks of different criticality (e.g. ABS Brakes with DVD) to share a processor and interfere with each other. These systems are known as mixed-criticality systems. While traditional temporal isolation techniques prevent all inter-task interference, they waste utilization because they need to reserve for the absolute worst-case execution time (WCET) for all tasks. In many mixed-criticality systems the WCET is not only rare, but at times difficult to calculate, such as the time to localize all possible objects in an obstacle avoidance algorithm. In this situation it is more appropriate to allow the execution time to grow by stealing cycles from lower-criticality tasks. Even more crucial is the fact that temporal isolation techniques can stop a high-criticality task (that was overrunning its nomimal WCET) to allow a low-criticality task to run, making the former miss its deadline. We identify this as the *criticality inversion* problem. In this paper, we characterize the criticality inversion problem and present a new scheduling scheme called *zero-slack scheduling* that implements an alternative protection scheme we refer to as *asymmetric protection*. This protection only prevents interference from lower-criticality to higher-criticality tasks and improves the schedulable utilization. We use an offline algorithm with two parts: a zero-slack calculation algorithm, and a slack analysis algorithm. The zero-slack calculation algorithm minimizes the utilization needed by a task set by reducing the time low-criticality tasks are preempted by high-criticality ones. This algorithm can be used with priority-based preemptive schedulers (e.g. RMS, EDF). The slack analysis algorithm is specific for each priority-based preemptive scheduler and we develop and evaluated the one for RMS. We prove that this algorithm provides the same level of protection against criticality inversion as the best known priority assignment for this purpose, criticality as priority asignment (CAPA). We also prove that zero-slack RM provides the same level of schedulable utilization as RMS when all tasks have equal criticality levels. Finally, we present our implementation of the runtime enforcement mechanisms in Linux/RK to demonstrate its practicality.**

I. Introduction

In modern real-time embedded systems, such as avionics and automotive systems, there is increasing pressure to reduce cost and physical resources such as power and heat. This has resulted in the consolidation of functionality of different criticality into shared hardware resources (e.g. processor and memory). Unfortunately, such sharing can lead to interference across tasks (e.g. one task using the processor longer than expected) of different criticality potentially causing critical failures. For instance, if we deploy an ABS braking system task on the same processor as a navigation system task of the car, if the latter does not release the processor on time it could make the former miss its deadline. This problem is a concern of growing interest. In particular, the US Air Force Research Laboratory has been leading the Mixed-Criticality Architecture Requirements (MCAR) [1] initiative to investigate building blocks to safely construct these mixed-criticality systems.

Resource partitioning is the key mechanism used to prevent interferences due to shared resources. This partitioning incurs its own costs. In particular, we need to provision for the worst-case resource demand (e.g. worst-case execution time – WCET) even though such a demand is only exercised in rare occasions. Obtaining this WCET is a challenging task due to hardware unpredictability (e.g. caches, pipelines, speculative execution) and the dependency of the execution time of some algorithms on the environment (e.g. number of obstacles to avoid). As a result, WCET numbers are often obtained through measurements and supplemented by an added cushion factor as an educated and typically conservative estimate. This leads to a twofold problem: poor average utilization of the processor and occasional enforcement (temporary stopping) of tasks that need to run longer than their WCET. Such enforcement prevents low-criticality tasks from interfering with higher-criticality ones and can make the low-criticality task miss its deadline while preventing the missing of the deadline of the higher-criticality task. Unfortunately, such enforcement can also be suffered by the higher-criticality task to prevent its interference on a low-criticality task. As a result, the enforcement can make the higher-criticality task miss its deadline to allow the low-criticality one to meet its own. We call this type of enforcement *symmetric enforcement* because it acts the same way in both directions: low-to-high and high-to-low criticalities. This enforcement can have the opposite effect to our original intent: a lower-criticality task is favored over a higher-criticality tasks and *criticality inversion* is said to occur.

One way to eliminate the criticality inversion problem is to simply assign the priorities of the tasks in the order of their criticality levels. We call this scheme *Criticality As Priority Assignment* (CAPA). However, this scheme can lead to very poor utilization if the resulting order turns out to be contrary to the best priority assignment to maximize utilization (e.g. rate-monotonic priority, proven to be optimal for fixed-priority scheduling). Such an approach can lead to very low levels of schedulable utilization due to forced priority inversion. In this paper, we describe a scheduling policy called *zero-slack scheduling* to provide a new form of protection we call *asymmetric protection*. This new form of protection eliminates the impact on deadline due to the criticality inversion while minimizing the penalty on schedulable utilization. We also

develop new metrics to evaluate criticality inversion.

The rest of this paper is organized as follows. Section II summarizes the related work in this area. Section III introduces the criticality inversion problem, and develops the blocking terms required to characterize it. Section IV develops the zero-slack scheduling scheme to deal with criticality inversion and presents performance metrics for mixed-criticality scheduling. Section V applies the zero-slack scheme to rate-monotonic (RM) scheduling, and proves useful properties about the integrated zero-slack RM scheduler. Section VI details our implementation of zero-slack RM scheduling on Linux/RK. Finally, we provide our concluding remarks in section VII.

II. Related Work

Multiple papers have been published related to overload scheduling. [2] and [3] use a form of criticality together with a value assigned to task completions. Their approach is then to maximize the accrued value. In our case, we do not require a value for task completion (which can be difficult to obtain) but use an absolute ranking that matches the common semantics of mixed-criticality task sets. Additional studies have also been done on online scheduling of overloads [4–6]. These schemes do not include an explicit notion of criticality and hence cannot take advantage of this notion.

Chi-Sheng et al. present in [7] an approach to map the semantic importance of tasks in a task set into QoS service classes to improve the resource utilization. Their objective is to ensure that the allocated resources are never less for a high-criticality task than for a lower-criticality one. In our case, we focus on deadlines, and guarantee that whenever an overload reduces the resources available we ensure that a high-criticality task will not miss its deadline due to the allocation of more resources to a lower-criticality task.

Various operating systems like Linux/RK [8], QLinux [9], Windows Vista [10], and Rialto [11], provide extensive mechanisms for resource isolation through advance reservations. Although such existing techniques are fundamental to providing hard timing guarantees, they are still agnostic to criticality and enforce isolation symmetrically. In contrast, our scheme uses tasks criticalities and overload budgets to create an *asymmetric enforcement* without incurring criticality inversion.

In the specific context of overload scheduling, scheduling algorithms based on the elastic task model have been proposed [12]. Tasks with higher elasticity are allowed to run at higher rates when required, whereas tasks with lesser elasticity are restricted to a more steady rate. This elasticity does not explicitly capture the criticality of a task, and it requires all remaining tasks to be compressed under an overload scenario. Our solution explicitly supports overload scenarios and criticality levels through a critical mode (C) of operation in addition to the normal mode (N) of operation. During this critical mode, we block the lower-criticality tasks, thereby transferring utilization only from the lower-criticality tasks under overloaded conditions to benefit higher-criticality tasks.

Closely related to our proposal for dual-execution modes viz. normal mode (N) and critical mode (C) is the work on dual-priority scheduling [13]. Dual-priority scheduling is an effective technique for responsively scheduling soft tasks without compromising the deadlines of crucial hard real-time tasks. While their notion of crucial tasks is somewhat similar to criticality there are two stark differences. First, non-crucial tasks are not given any scheduling guarantees in their work, whereas, in our scheme all criticality levels have a guarantee on their execution times. Secondly, their limitation to two levels simplifies the analysis but also limits the utility. In contrast, we allow an unlimited number of criticality levels with an implicit graceful degradation on overload that follows the criticality order.

Another related work to dual-priority scheduling is the earliest deadline zero laxity (EDZL) algorithm [14, 15]. In EDZL, tasks are scheduled based on earliest deadline first (EDF) until some task reaches zero laxity (or zero slack), in which case its priority is elevated to the highest level. While the notion of zero slack is used in our solution, the existing EDZL results do not consider the notion of task criticalities and are not directly applicable to the mixed-criticality scheduling problem.

Period transformation has been an important solution studied in the context of mixed-criticality systems [16, 17]. This technique has been subsequently extended and generalized in [18]. For any given task set, we can transform the tasks by scaling both their computation time and period equally. In the context of mixed criticality, we can perform period transformation to ensure that the task priorities match the required criticality. The problem in using this solution for dealing with overload scenarios is that the period transformation needs to be performed assuming the task will overrun its absolute WCET, i.e., with an overloaded computation-time. Performing such a period transform using the overloaded computation time results in early and pessimistic preemptions by the period-transformed higher-criticality tasks over the lower-criticality tasks. The proposed zero-slack scheduling algorithm switches tasks to their critical mode only when there would be zero slack remaining to meet their corresponding overloaded computation time requirements. Therefore, the zero-slack scheduling algorithm does not impose more interference than necessary to provide scheduling guarantees in mixed-criticality systems.

III. The Criticality Inversion Problem

Priority-based preemptive scheduling policies assign priorities to tasks, and at run-time, attempt to schedule the highest-priority ready task. The priority assignment can be fixed across task instances (fixed-priority schedulers) or it can change across task instances (dynamic priority schedulers). In traditional real-time scheduling, priorities are assigned with the purpose of maximizing schedulable utilization while respecting the deadlines of all the tasks in the set.

The utilization maximization approach of traditional real-time schedulers makes two important assumptions: (1) all tasks are equally important, and (2) the utilization never goes beyond the allowable threshold(s). These two assumptions seldom hold in mixed-criticality systems. In particular, tasks

can have different criticality levels. Hence, if there is ever a situation where we can only satisfy the deadline of one task, then we should choose to meet the one with higher criticality.

The second assumption does not hold for a more subtle reason. Specifically, the fact that enforcement is even considered as a protection mechanism implies that there is a possibility that some tasks can go beyond their specified worst-case execution time. While this is, in general, considered a fault, an explicit protection against it is needed because it can create (temporal) overload situations. These overload situations are precisely when criticality inversion can occur. Since the traditional priority assignment made by the scheduler was tailored to increase schedulable utilization, it is agnostic to criticality. In particular, under priority-based preemptive scheduling, a low criticality task can have a higher scheduling priority than a higher-criticality task. Such priority assignment would schedule the low criticality task earlier than the higher-criticality task, potentially making the latter miss its deadline.

On the other hand, if we assign priorities based on criticality (as introduced in Section I), then we eliminate criticality inversion. However, this assignment can potentially create significant priority inversion [19] from the perspective of priority-based preemptive schedulers.

To consider both scheduling priorities and task criticalities, and explicitly capture the overload execution requirements, let us define a task τ_i as:

$$\tau_i = (C_i, C_i^o, T_i, D_i, \zeta_i)$$

where:

- C_i is its worst-case execution time under non-overloaded conditions,
- C_i^o is the overload execution budget,
- T_i is the period of the task,
- D_i is the deadline of the task (with $D_i \leq T_i$),
- ζ_i is the criticality of the task. We follow the same convention as with priorities: lower its value, higher the criticality.

It is worth noting that the overload budget C_i^o is used to create a precise definition our guarantee, i.e., how much overload is guaranteed. As we will show later, it is possible to reserve a conservative (large) C_i^o since it does not add up across criticality levels.

The priority blocking density PB_i for task τ_i is then defined as:

$$PB_i = \sum_{\tau_j | \zeta_j < \zeta_i} \frac{pb_i^j}{D_i} \tag{1}$$

where:

- pb_i^j is the maximum time that a higher-criticality task τ_j can block the execution of an instance of τ_i during which the scheduling priority of τ_i is higher than that of τ_j.

Similarly, the criticality blocking density CB_i for task τ_i is calculated as:

$$CB_i = \sum_{\tau_j | \zeta_i < \zeta_j} \frac{cb_i^j}{D_i} \tag{2}$$

where:

- cb_i^j is the maximum time that a lower-criticality task τ_j can block the execution of an instance of τ_i during which the scheduling priority of τ_j is higher than that of τ_i.

PB_i and CB_i become blocking utilization when $D_i = T_i$ quantifying the exact schedulable utilization loss. However, we present our general discussion on utilization loss using the blocking density figures to allow the more general case where $Di \leq T_i$. Notwithstanding we will use the blocking utilization to evaluate our *Zero-Slack Rate-Monotonic* algorithm in Section V-E since $D_i = T_i$ in this case.

Both pb_i^j and cb_i^j vary depending on how and when the scheduler assigns priorities and the strategies used to tradeoff priority and criticality blocking. In the next section we discuss the strategies taken in our new scheduling scheme.

IV. Zero-Slack Scheduling

Our scheduling policy works on top of traditional priority-based preemptive real-time schedulers. It is based on the observation that criticality inversion only matters under overload conditions. We use this observation to create two execution zones for each task τ_i. In the first zone, every task is included while in the second zone, every task $\tau_j | \zeta_j > \zeta_i$ is suspended. This suspension effectively blocks the interference of lower-criticality tasks in the case of an overload condition, up to the completion of the task activation. It must be noted that τ_i itself can also be suspended by a task $\tau_c | \zeta_c < \zeta_i$ in τ_c's second zone. The execution zones partition the execution of each task into two modes: the normal mode (*N mode*) and the critical mode (*C mode*). Our scheduling algorithm then calculates the execution time for each mode.

We now define our scheduling guarantee.

A. Scheduling Guarantee

Our scheduler performs admission control, and if admitted[1], a task τ_i is guaranteed to run up to C_i^o if no higher criticality tasks exceeds its C_i.

This guarantee follows the separation of the overloaded from the non-overloaded situation. This separation allows us to make two strategic decisions. First, when no overload condition is present, we should schedule the task with the objective of maximizing utilization. And secondly, we also avoid introducing any interference prevention until the last instant necessary to satisfy our guarantee.

The last instant to block interference for a task τ_i is the latest time at which the interference prevention mechanism can free enough cycles for τ_i to complete its C_i^o before its deadline. This instant is identified as *zero-slack* instant because it leaves no slack in C mode after the completion of C_i^o. This is the instant at which the execution mode of the task switches from N mode into C mode. Avoiding interference prevention till this last instant enables us to maximize the total schedulable utilization, while still meeting our scheduling guarantee.

[1]In the rest of this paper, when we use the term schedulable, we refer to the property of satisfying this scheduling guarantee.

Our scheme is divided into two parts. The first part is an admission test that determines the zero-slack instant for each of the tasks in a task set. The second is a runtime enforcement mechanism that prevents interference based on the criticality and the zero-slack instant of each task.

B. Admission Control

Our admission control scheme takes the form of a slack-discovery algorithm. We start with the worst-case assumption that there is no slack in the N mode of the tasks and they always need to run in C mode. This is equivalent to CAPA. Then, we start moving computation time from the C mode (in C_i^c) to the N mode (in C_i^n) of the tasks as we discover slack in the N mode. This scheme is presented in Algorithm 1. The algorithm converges when it cannot move the zero-slack instant of any task towards their deadline. This is codified with two variables per task Z_i^0 (zero-slack before cycle) and Z_i^1 (zero-slack after cycle). Inside the cycle, the algorithm calculates the slack vector in for task τ_i in N mode (with a set of all the original tasks - Γ_i^n) and C mode (with a set of higher-criticality tasks - Γ_i^c). The slack vector contains a sequence of slack regions ordered by time, where each slack region contains a starting instant and duration. With these slack vectors, we obtain a new zero-slack instant (Z_i^1) with the function *GetSlackZeroInstant* that takes C_i^o and accommodates the most computation available in the N mode slack vector, with the rest in the C mode slack vector. After the convergence of this algorithm a taskset Γ is schedulable if $\forall \tau_i \in \Gamma\ TotalSlack(V_i^c) \geq C_i^o$.

Algorithm 1 Compute Final Zero-Slack Instants ($t = D_i$)

```
1: ∀i Z_i^1 ⇐ 0
2: repeat
3:    ∀i Z_i^0 ⇐ Z_i^1
4:    for all i in taskset do
5:       V_i^n ⇐ GetSlackVector(i, Γ_i^n)
6:       V_i^c ⇐ GetSlackVector(i, Γ_i^c)
7:       Z_i^1 ⇐ GetSlackZeroInstant(i, V_i^c, V_i^n, t)
8:    end for
9: until ∀i Z_i^0 = Z_i^1
10: return Z_i^1
```

The function *GetSlackVector* used in Algorithm 1 is specific to each priority-based preemptive scheduling algorithm and we will discuss it later in the context of rate-monotonic scheduling. However, the function *GetSlackZeroInstant* is generic. This function is presented in Algorithm 2. At the beginning of this algorithm, we initialize the C_i^c and C_i^n of the task to C_i^o and 0 respectively. The cycle in this algorithm first finds the last time instant at which C_i^c units of slack are available in the slack vector V^c before $t = D_i$ (at line 3). Then, we calculate how much slack is available from zero to this instant in the vector V^n (at line 5). This available slack is then transferred from C_i^c to C_i^n (lines 7 and 8). The next iteration of the cycle then finds the next instant for a smaller C_i^c (due to the transfer in lines 7 and 8), moving this instant closer to $t = D_i$. The loop ends when we are not able to transfer any more computation from C_i^c to C_i^n. The function *StartOfTrailingSlack* returns the instant at which the requested slack (C_i^o) for task τ_i starts such that it is available before deadline $t = D_i$. The function *SlackUpToInstant*, on the other hand, returns the amount of slack available up to the specified instant. It is to be noted here that for simplicity of presentation, we assume that the origin of time coincides with the release of task τ_i.

TABLE I
SAMPLE CRITICAL TASK SET

Task	C	C^o	T	D	Criticality	Priority
τ_1	2	2	4	4	1	0
τ_2	2.5	5	10	8	0	1

Algorithm 2 GetSlackZeroInstant(i,V^c, V^n, t): Calculate Instant of Slack = 0 before time t

```
1: C_i^c ⇐ C_i^o ; C_i^n ⇐ 0
2: repeat
3:    t_1 ⇐ StartOfTrailingSlack(i, C_i^c, V^c)
4:    if t_1 ≥ 0 and t_1 ≤ t then
5:       k_u ⇐ SlackUpToInstant(V^n, t_1) − C_i^n
6:       k_u = max(min(k_u, C_i^c), 0)
7:       C_i^c ⇐ C_i^c − k_u
8:       C_i^n ⇐ C_i^n + k_u
9:    else
10:      k_u ⇐ 0
11:   end if
12: until k_u = 0
13: return t_1
```

C. A Two-Task Example

We now illustrate the admission control algorithm using a two-task example and deadline-monotonic scheduling.

Consider the task set in Table I. In this table, we present the parameters of two tasks, with an additional column for their deadline-monotonic priority. Note that the priority and the criticality of the tasks in this table are the inverse of each other. With these parameters we proceed to calculate the C_i^c and C_i^n for each task τ_i in the set.

For our task set in Table I, C_2^n and C_2^c are obtained by first calculating the slack for τ_2 in both the N mode and the C mode. In N mode, the slack up to the end of its deadline (8) is 4 which is not enough to schedule its $C_2^o = 5$ (this is, in fact, the problem when using pure deadline-monotonic scheduling). This means that we need to switch this task to C mode at some point to compensate. To do this, we start by executing its $C_2^c = 5$ time in C mode. Then, we evaluate the last instant Z_2 that C_2^c time units are available in C mode. Since τ_2 has the highest criticality, it is the only task running in its C mode, and hence this last instant is its deadline $(8) - C_2^c(5) = Z_2 = 3$. Then, we check how much time is available in N mode up to

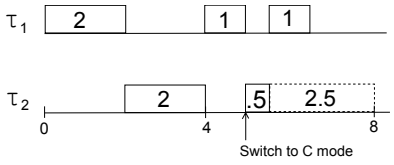

Fig. 1. Two-Task Example of the Zero-Slack Schedule

time 3, which in this case is 1 (i.e., $3 - C_1^o = 1$). We then make this the initial time for C_2^n and subtract it from C_2^c. Next, we repeat the operation to find the last instant to schedule $C_2^c = 4$ units of time in C mode ($Z_2 = 4$). This, in turn, increments the available slack in N mode (up to instant 4) to 2, moving another unit to C_2^n (i.e., $C_2^n = 2$ and $C_2^c = 3$). The last increment then leads us to the last instant when $C_2^c = 3$ is available in C mode ($Z_2 = 5$). This new instant does not increment the available slack in N mode but defines the final zero-slack instant Z_2 for task τ_2 to switch to C mode. The final schedule can be seen in Figure 1. In this figure, we can see τ_1 running first for its whole execution time $C_1^o = 2$. Then, τ_2 runs for 2 units and then gets preempted by the second arrival of τ_1 at time 4. Then, τ_1 runs for one more unit. At this point we reach the zero-slack instant Z_2 of τ_2 and suspend τ_1. When there is no overload in the system, τ_2 will run only for 0.5 units more and resume τ_1. However, in an overload situation, τ_2 will have enough cycles to complete its $C_2^o = 5$ before the end of its deadline (at 8).

It is worth noting that our computation of the zero-slack instant in this example starts by assuming that there is no slack in the N mode and, as it discovers more slack, keeps moving computation to N mode. The key property of this technique is that the computation executed in C mode decreases monotonically. A second important observation is that the initial execution distribution (between C_i^n and C_i^c) may not be schedulable. In this example, the initial starting point is, in fact, not schedulable, i.e., if τ_2 runs always in C mode and only uses up to C_2 (2.5) units of time starting at zero, τ_1 misses its deadline because only 1.5 units are left before its deadline (4) and our guarantee is not honored (this is how CAPA would schedule the task set). However, with the final scheduling parameters the task set honors our guarantee. That is, if τ_2 only uses C_2, then it means that at its zero-slack instant (5), it will only run for $\frac{1}{2}$ a time unit more leaving $2\frac{1}{2}$ time units for τ_1 (that only needs 1 more) to complete its C_2^o.

D. Proof of Convergence of the Zero-Slack Computation

Because our scheme relies on the convergence of the zero-slack computation algorithm, we state the assumptions of the algorithm and prove its convergence.

The zero-slack computation algorithm assumes the use of a *non-anomalous*[2] fixed-priority scheduling algorithm. These algorithms ensure that reducing any C_j of $\tau_j \in \Gamma$ never increases the response time W_i $\forall \tau_i \in \Gamma$. Uniprocessor scheduling algorithms like RMS and CAPA are known to be non-anomalous with respect to computation time [20]. Assuming the use of this algorithm, we can prove convergence.

Lemma 1: When non-anomalous scheduling algorithms are used to calculate the slack vectors, the Compute Final Zero-Slack Instants algorithm (Algorithm 1) converges.

Proof: The proof follows from the following three loop invariants for Algorithm 1:

1) $Z_i^1 \geq 0$ $\forall \tau_i \in \Gamma$
2) $Z_i^1 \geq Z_i^0$ $\forall \tau_i \in \Gamma$
3) $Z_i^1 \leq D_i$ $\forall \tau_i \in \Gamma$

Given these three loop invariants, we can observe that Z_i^1 starts at 0, never decreases since $Z_i^1 \geq Z_i^0$, and is bounded to be less than or equal to D_i. Therefore, Algorithm 1 converges.

We now prove that these three loop invariants hold true for Algorithm 1.

$Z_i^1 \geq Z_i^0$ $\forall \tau_i \in \Gamma$: At the beginning of the iteration, task τ_i is split into C_i^c units of execution in the critical mode (C), and $C_i^o - C_i^c$ units of execution in the normal mode (N). During the iteration, the execution time in the critical mode C_i^c never increases. For any *non-anomalous* scheduling algorithm, the slack available in the critical mode (C) can never decrease since C_i^c never increases. Therefore, the zero slack instant Z_i^1 calculated using the slack available in the critical mode (C) is never pushed earlier than Z_i^0 i.e. decreased in value.

$Z_i^1 \geq 0$ $\forall \tau_i \in \Gamma$: this follows from the initialization condition $Z_i^1 = 0$ and the previously established invariant that $Z_i^1 \geq Z_i^0$ $\forall \tau_i \in \Gamma$.

$Z_i^1 \leq D_i$ $\forall \tau_i \in \Gamma$: Consider a task τ_i with a zero-slack scheduling instant of D_i. The amount of time spent in the critical mode (C) is 0. Therefore, there is no additional slack available in the critical mode (C). This implies that the zero-slack scheduling instant can be moved no further than D_i. ■

E. Runtime Behavior of The Zero-Slack Scheduling

The central piece of the runtime mechanism of our scheme is the dual-mode execution implementation. It is important to keep the overhead of such a mechanism as lean as possible. We can achieve this by using the zero-slack instants calculated offline as timers to switch to C mode. These timers then mark the time when the lower-criticality tasks are suspended. For this reason, we need to keep track of the criticality level of the tasks to be able to distinguish which tasks need to be suspended. At the same time, we need to keep track of which tasks are in C mode in order to know which tasks to resume when a task finishes its C mode. In a resource reservation framework, the admission test can be performed at the time of the resource reservation and the corresponding zero-slack instants can be computed.

F. Effectiveness of Scheme

To evaluate the effectiveness of our scheduling algorithm, we defined a metric to measure a reduction of the blocking density factors (PB_i and CB_i) defined in Section III. For

[2]Some multiprocessor scheduling schemes are known to be anomalous, where this property does not hold.

CB_i, we know that a task $\tau_j|\zeta_i < \zeta_j$ imposes a blocking term $cb_i^j = C_j^o$ in the worst case. For PB_i, a task $\tau_j|\zeta_j < \zeta_i$ imposes a blocking term $pb_i^j = C_j$ in the worst case.

Because the blocking factors at each task level may involve parts of the blocking factors of other tasks, the final taskset-wise blocking, is calculated by taking the maximum blocking among all the tasks:

$$TB_\Gamma = \max_{\tau_i \in \Gamma} (CB_i + PB_i)$$

This total blocking is the maximum blocking present in the task set Γ whose criticality-awareness is improved by zero-slack scheduling. We call this combined blocking *mixed-criticality blocking* (MC blocking). It is worth noting that this value depends on the level of inversion between priorities and criticalities, as well as their budgets and periods.

For instance, for the task set in Table I, the blocking factors are defined as:

- $PB_1 = \frac{C_2}{D_1} = \frac{2.5}{4} = 0.625$
- $PB_2 = 0$ (because it is the lowest priority)
- $CB_1 = 0$ (because it is the lowest criticality)
- $CB_2 = \frac{C_1^o}{D_1} = \frac{2}{4} = 0.5$

Hence, the MC blocking of the task set is the maximum of the sum of the factors that is equal to 0.625.

1) Residual MC blocking: The effectiveness of an algorithm in reducing MC blocking is calculated by the amount of residual MC blocking that the algorithm is not able to remove. For instance, in the task set of Table I, when criticality is used as the scheduling priority (CAPA), both CB_1 and CB_2 are zero. However, $PB_1 = \frac{C_2}{D_1} = \frac{2.5}{4} = 0.625$ (and $PB_2 = 0$). Taking the maximum of the factors the residual MC blocking left by the algorithm is 0.625. On the other hand, when RMS is used, then $PB_1 = PB_2 = 0$. But $CB_2 = \frac{C_1^o}{D_1} = \frac{2}{4} = 0.5$ (and $CB_1 = 0$), is the residual MC blocking left by the algorithm. Finally, in the case of the zero-slack scheduler, the factors are reduced to $PB_1 = \frac{C_2^c}{D_1} = \frac{0.5}{4} = 0.125$, $PB_2 = 0$, $CB_1 = 0$, and $CB_2 = \frac{C_1}{D_2} = \frac{C_1}{D_2} + \frac{1}{D_2} = \frac{2}{8} + \frac{1}{8} = 0.375$ ($\frac{1}{D_2}$ is the fraction of C_1 that executes in the second arrival of τ_1 before the zero-slack instant). Hence, the residual MC blocking is 0.375. It is worth noting that the zero-slack scheduler reduces both PB and CB. PB is reduced because only a fraction of the C_2^o (0.5) is run with a priority greater than the scheduling priority (deadline-monotonic priority). CB is reduced because the switching to critical mode at the zero-slack instant of τ_2 reduces this blocking from $\frac{2}{4}$ to $\frac{3}{8}$ (2 in the first period of τ_1 and 1 in its second period). The reduction in criticality blocking is just enough to enable the completion of C_2^o. Furthermore, the fact that PB_1 was reduced from 0.625 to 0.125, when compared to CAPA, indicates that the zero-slack algorithm has further capacity to tolerate a larger priority blocking without breaking the scheduling guarantee. We capture this tolerance with a metric we call *laxity density*.

2) Laxity Density: The laxity density of a task τ_i measures the available density for τ_i after we discount the preemptions of higher-priority tasks and the blocking factors left by the scheduling algorithm of interest for this task. The laxity density for a task τ_i is then defined as:

$$LaxU_i = A(i) - \frac{C_i^o}{D_i} - (PB_i + CB_i) - \sum_{\tau_j \in H_i^{hc}} \frac{C_j}{D_j} \quad (3)$$

where:

- A(i) is the available density for task τ_i for a specific priority-based preemptive scheduler. For instance, for rate-monotonic scheduling with n implicit-deadline tasks, this is $n(2^{\frac{1}{n}} - 1)$ (see [21])
- $\frac{C_i^o}{D_i}$ is the overload density of τ_i
- $PB_i + CB_i$ is the blocking factor for the scheduling algorithm of interest
- $\sum_{\tau_j \in H_i^{hc}} \frac{C_j}{D_j}$ is the density consumed by the higher-priority and higher-criticality tasks (H_i^{hc})

The laxity density allows us to see how an algorithm spreads the laxity of the system across the different tasks. Hence a successful algorithm will keep the minimum laxity across the tasks of the task set always greater than or equal to zero. This means that, an algorithm X is better than an algorithm Y, if there exist task sets where the minimum laxity under Y goes below zero while under X is greater than or equal to zero, but if there exists a task set with a minimum laxity under X below zero, its minimum laxity under Y is also below zero. We will use this metric to evaluate the performance of our zero-slack RM once we discuss its details in the next section.

V. The Zero-Slack Rate-Monotonic Scheduling (Zero-Slack-RM)

The zero-slack scheduling policy works on top of a priority-based preemptive scheduler and relies on the calculation of slack vectors provided by such scheduler. In this section, we present the slack vector calculation for the rate-monotonic scheduler. We chose the RM scheduler to simplify the explanation of the zero-slack scheme. Hence, in this section we will assume implicit deadlines ($D_i = T_i$).

A. Worst-Case Phasing of Dual-Mode Tasks

Key to the calculation of the slack vectors in the rate-monotonic scheduling is the phasing of the tasks. For a single-mode execution, Liu and Layland [21] proved that the phasing that creates the maximum preemption for a task τ_i happens when every task $\tau_j|priority(\tau_j) < priority(\tau_i)$ arrives at the same time as τ_i. However, in a dual-mode task, this worst-case phasing does not hold. This is because, when tasks reach their zero-slack instants, they will suspend lower-criticality tasks. On the one hand, this suspension acts, as intended, to avoid preemptions suffered by task τ_i from lower-criticality tasks. However, it also acts as a preemption when higher-criticality tasks suspend τ_i. Hence, to calculate the worst-case delay imposed by this type of preemption, we need to align all the suspensions in the same way as the period arrivals. Unfortunately, if we align the zero-slack instants of the higher-criticality tasks, we may misalign the arrival of higher-priority tasks. In other words, it is not always possible to align both

TABLE II
ZERO-SLACK-RM SCHEDULED TASK SET

Task	C	C^o	T	D	Criticality	Priority	ZS Instant
τ_0	10	50	100	100	2	0	80
τ_1	20	100	200	200	1	1	60
τ_2	40	200	400	400	0	2	200

the worst-case arrival of the tasks and the zero-slack instants. The implication of this misalignment is that we cannot create a single integrated critical zone based on the alignment of both types of preemptions. As a result, we take a pessimistic approach by assuming that the effects of both alignments always happen.

Although the worst-case phasing may not exist, it provides an upper bound on the total interference imposed on task τ_i. This can be shown as follows. Before the zero-slack instant, the maximum interference from higher-priority tasks happens when they are released simultaneously with τ_i (from [21]). After the zero-slack instant, τ_i effectively blocks all the lower - criticality tasks. Therefore, the interference can only arise from higher-criticality tasks. By switching all the higher-criticality tasks to their critical mode (C) along with τ_i, the interference suffered by τ_i in the critical mode (C) is also maximized.

B. A Zero-Slack-RM Scheduling Example

Let us use an example to illustrate the characteristics of the zero-slack-RM scheduler. Table II presents a task set with the priorities assigned by the rate-monotonic scheduler and the zero-slack instants calculated by our algorithm.

Due to space limitations, we will focus our discussions on τ_1. Figure 2 presents the critical zone of this task. In this figure, we can see the preemption from τ_0 in the N mode of τ_1 for 50 units of time. After this, τ_1 runs for 10 units and then reaches its zero-slack instant at time 60, switching to C mode. In C mode, it suspends the lower-criticality task τ_0, but at the same time it is suspended by the higher-criticality task τ_2. This suspension is the pessimistic approach we use due to the absence of an exact worst-case phasing (as discussed in Subsection V-A). τ_2 then runs for C_2 (40) units and resumes the lower-criticality tasks. However, in order to maintain the criticality order, this resumption is implemented as a stack, meaning that it only returns to the previous criticality level (leaving τ_0 suspended). Then, τ_1 can continue executing completing its C_1^o (100) at time 190.

Each task in the task set has its own (pessimistic) critical zone similar to the one presented in Figure 2, but they are unfortunately not necessarily aligned with each other.

C. Calculating The Slack Vectors in Zero-Slack-RM

In order to calculate the slack vectors in zero-slack-RM, we first define an interfering taskset for each task τ_i under zero-slack-RM. For each of the two execution modes (C and N=RM), there could be different tasksets that could interfere with (i.e. preempt) τ_i. We use C_j^E to represent the *effective* execution time that will be considered for each interfering task

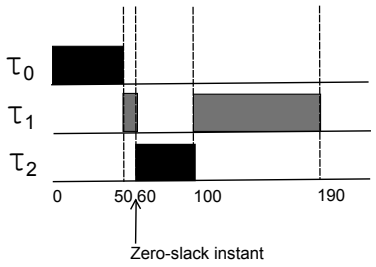

Fig. 2. Critical Zone of *Task 1*

τ_j. In particular, in the RM mode, the task set that would interfere with task τ_i is defined as:

$$\Gamma_i^n = \Gamma_i^{rm} = H_i^{lc} \cup H_i^{hc} \cup L_i^{hc} \cup S_i$$

where,

* H_i^{lc} is the set of tasks with higher rate-monotonic priorities but lower criticality. These tasks attempt to run for C^o units. That is, $\forall \tau_j \in H_i^{lc}$, $C_j^E \Leftarrow C_j^o$.

* H_i^{hc} is the set of tasks with higher rate-monotonic priorities and higher criticality. These tasks run for their respective non-overloaded computation time. If a higher-criticality task executes beyond its non-overloaded computation time, then the scheduling guarantee of lower-criticality task τ_i need not be honored. Hence, $\forall \tau_j \in H_i^{hc}$, $C_j^E \Leftarrow C_j$.

* L_i^{hc} is the set of tasks with lower rate-monotonic priorities and higher criticality. These tasks are considered to be running in their C mode. In fact, we only need to consider the part of computation that runs in its C mode. That is, assuming that $a_j^{rm,i}$ contains the slack available to task $\tau_j \mid \tau_j \in L_i^{hc}$ in RM mode, while honoring the guarantee of task τ_i, then $C_j^E \Leftarrow (C_j - a_j^{rm,i})$. Initially, we assume that there is no slack available in RM mode ($a_j^{rm,i} = 0$), but as we move the zero-slack instants of the tasks towards the end of their period we will discover additional slack in RM mode reducing C_j^E further, in turn the available slack in RM mode increases.

* S_i is the set of tasks with the same level of criticality but higher priority. These tasks are running at their C^o, i.e., $\forall \tau_j \in S_i$, $C_j^E \Leftarrow C_j^o$.

The interfering task set for C mode of task τ_i is defined as:

$$\Gamma_i^c = H_i^{hc} \cup L_i^{hc} \cup S_i$$

with the same set of definitions as before. Note that the low criticality tasks are excluded since the lower-criticality tasks are blocked during τ_i C mode of execution.

With these task sets, we create a slack vector for each execution zone using Algorithm 3. This algorithm receives as parameters the index of the task we are evaluating and the interfering task set, ordered by increasing order of priority.

Algorithm 3 calculates the slack vector given a task τ_i and its interfering task set Γ, for the execution mode under consideration. The algorithm uses a parameter C_i^v to accumulate the total slack available to τ_i over the entire period of duration t. The outer loop of Algorithm 3, adds new entries to the slack

vector. The inner loop computes the time instant $R_{current}$ by which C_i^v units of slack are available, and the time instant b at which the next interfering task I_m arrives. Difference between b and $R_{current}$ is the next available slack region for τ_i.

Algorithm 3 GetSlackVector(i, Γ, $t = T_i$): Slack Vector Calculation

```
index ⇐ 0 ; C_i^v ⇐ 0
repeat
    R_current ⇐ C_i^v 1 b ⇐ 0
    repeat
        R_previous ⇐ R_current
        R_current ⇐ C_i^v + Σ_{j∈Γ} ⌈R_previous / T_j⌉ C_j^e
        b ⇐ t ; I_m ⇐ i
        for (j ∈ Γ) do
            A ⇐ ⌈R_previous / T_j⌉ T_j
            if (A < b) then
                b ⇐ A ; I_m ⇐ j
            end if
        end for
        if (R_previous = R_current ) then
            R_current ⇐ R_current + C_{I_m}^e
        end if
    until (R_previous = R_current or R_current ≤ t)
    V_i[index].slack ⇐ min(b, t) − R_current
    V_i[index].time ⇐ R_current
    C_i^v ⇐ C_i^v + (min(b, t) − R_current)
    index + +
until (R_current ≥ t)
return V_i
```

D. Properties of The Zero-Slack-RM Scheduler

We now present the properties of our zero-slack RM scheduler.

1) Subsumes CAPA: Any task set schedulable under Criticality-As-Priority Assignment(CAPA) is also schedulable under the zero-slack scheduling scheme.

Proof: Algorithm 1 starts with assigning $Z_i^0 = Z_i^1 = 0$ for all tasks τ_i. Under this assignment of zero-slack instants, the zero-slack scheduler behaves essentially like a CAPA scheme, since whenever τ_i is released all the lower criticality tasks are immediately blocked due to τ_i switching to its critical mode ($Z_i^1 = 0$). Therefore, if the task set is schedulable under CAPA, it should be schedulable at the first iteration of Algorithm 1. In this scenario, we now inductively prove that each task τ_i remains schedulable over subsequent iterations.

During subsequent iterations of the Compute Final zero-slack Instant algorithm, the Algorithm 2 is used to transfer additional computation of up to k_u from the critical mode (C) to the normal mode (N). This transfer is performed only to use up the slack of k_u available in the normal mode (N) as calculated using *SlackUpToInstant* till the zero-slack instant t_1. Considering any task τ_i, this transfer of computation does not increase the blocking terms suffered by τ_i from higher criticality tasks executing in their C mode. The normal mode N of τ_i remains unaffected, since the additional computation is transferred to only fill up the available slack. Therefore, the response time of task τ_i only reduces with subsequent iterations. Hence, if task τ_i was schedulable in the previous iteration, it continues to remain schedulable. This completes the proof by induction. ■

2) Corollary: Graceful Degradation: When a task τ_j executes for a C such that $C_j < C \leq C_j^o$, then only tasks τ_i with $\zeta_i > \zeta_j$ can miss their deadline. This follows from the scheduling guarantee provided by zero-slack scheduling, which was described in subsection IV-A, and the property that rate-monotonic scheduling is non-anomalous.

3) Subsumes RM Scheduler: Any task set schedulable under rate-monotonic scheduling is also schedulable under the zero-slack scheduling scheme.

Proof: For any task set Γ consisting of tasks $\tau_i = (C_i, T_i)$ schedulable under the RM scheduling scheme, consider an equivalent Γ_z with tasks $\tau_i^z = (C_i, C_i^o, T_i, \zeta_i)$ with $C_i = C_i^o = C_i$ and $\zeta_i = \pi_i$, where π_i is the priority assigned to task τ_i under RM scheduling. Scheduling the task set Γ_z using CAPA produces the same schedule as the RM scheduler, since the priorities are completely aligned with the criticality under the chosen ζ_i values. Hence, Γ_z is also schedulable under CAPA, since it is schedulable under RM scheduling. Using the property that zero-slack scheduling subsumes CAPA, it follows that Γ_z is also schedulable under zero-slack scheduling. ■

4) Utilization Properties: A task set Γ_z with with tasks $\tau_i^z = (C_i, C_i^o, T_i, \zeta_i)$ is schedulable under zero-slack-RM if:

$$U_{rm} \geq U_z = \sum_{\zeta_k = \zeta_i and T_k < T_i} \frac{C_k^o}{T_k} + \sum_{\zeta_j < \zeta_i and T_j < T_i} \frac{C_j}{T_j} + \frac{C_i^o + \sum_{\zeta_p < \zeta_i and T_p \geq T_i} C_p}{T_i} \quad (4)$$

From the perspective of τ_i, under overloaded conditions, the utilization is $\frac{C_i^o}{T_i}$. The worst-case interfering utilization arising from the equal-criticality higher-priority tasks τ_k is given by $\frac{C_k^o}{T_k}$, since they could potentially misbehave. The worst-case interfering utilization arising from higher-criticality higher-priority tasks τ_i is given by $\frac{C_j}{T_j}$, and the blocking utilization from higher-criticality lower-priority tasks is given by $\frac{C_p}{T_i}$. We do not have to consider the lower-criticality tasks since they cannot affect the schedulability of task τ_i. If the lower-criticality tasks could impose any interference on τ_i that causes it to potentially become unschedulable, then the zero-slack instant would have been moved earlier to ensure the schedulability of τ_i.

It is important to note here that Equation (4) is only used to present a property of the Zero-Slack RM Scheduler. For the purposes the admission control, we use the response-time test based slack discovery algorithm described in Algorithm 3.

E. Laxity Utilization Comparison

The properties of the zero-slack-RM algorithm can be depicted graphically in a comparison of its laxity utilization

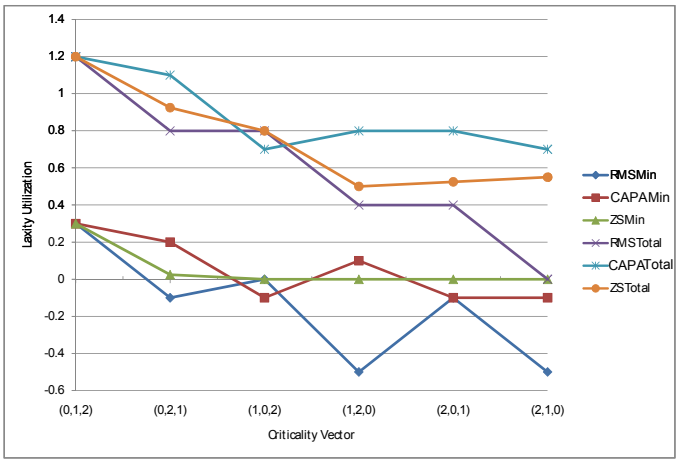

Fig. 3. Laxity Utilization vs Criticality-to-Priority Inversion

against that of both RMS and CAPA. Figure 3 depicts this comparison. We decided not to include period transformation in the comparison given that in the end it will behave as RMS with overloaded computation time, as described in section II.

For the comparison, we use the task set presented in Table II. We then vary the criticality of the task set (as a criticality vector $[\zeta_0, \zeta_1, \zeta_2]$) to explore all the different degrees of priority-to-criticality inversion. Because this task set is harmonic, we use 1 as the available utilization $A(i)$ for each task τ_i, the maximum for a harmonic task set under rate-monotonic scheduling. The assigned criticality vector is shown in the X-axis, and the minimum and total laxity utilization values are plotted. It is to be noted here that the total laxity utilization can be greater than 1 because the laxity available to a task could also be available to other tasks in the system.

Figure 3 shows that there are criticality assignments for which either RMS or CAPA or both algorithms are unable to achieve a minimum laxity greater than or equal to 0 (to make it schedulable), whereas zero-slack scheduling ensures a minimum laxity greater than or equal to zero in all cases. In particular, this figure shows that, for a criticality assignment $[1, 0, 2]$, both RMS and zero-slack scheduling algorithms are feasible, whereas, CAPA renders the task set unschedulable. However, for the criticality assignment $[1, 2, 0]$, RMS is unschedulable, whereas, CAPA and zero-slack scheduling continue to remain feasible. This illustrates that zero-slack scheduling subsumes both RMS and CAPA.

We observe that the total laxity available under zero-slack scheduling is bounded by the maximum total laxity available under RMS and CAPA. Specifically, Figure 3 shows that zero-slack scheduling always has a total laxity that is either upper bounded by the laxity of RMS or by the laxity of CAPA. However, there are criticality assignments like $[2, 1, 0]$, where both RMS and CAPA have negative minimum laxity but zero-slack scheduling remains feasible. The reason for this behavior is that the key factor determining feasibility is the distribution of available laxity, as opposed to the total available laxity itself. For the criticality assignment $[2, 1, 0]$, while CAPA has more laxity than zero-slack scheduling, the latter distributes the laxity more evenly among the tasks by controlling the switching between different execution modes.

F. Relaxing Constraints

For simplicity of presentation, the discussion so far has assumed implicit-deadline constraints and accommodated rate-monotonic scheduling. It is important to note here that our slack vector calculation in Algorithm 3 uses generic static-priority preemptive scheduling results [22], therefore enabling it to be readily extended to other priority assignments. For instance, it can be extended to deadline-monotonic scheduling, which is an optimal scheduling algorithm for systems with constrained deadlines ($D_i \leq T_i$). In the context of deadline-monotonic scheduling, it is straight forward to use $t = D_i$ instead of T_i in Algorithm 3, after assigning deadline-monotonic priorities to tasks. Due to space constraints, we do not discuss the density-based properties of such a Zero-Slack Deadline-Monotonic (ZSDM) scheduler.

Subsequent support for release jitter also follows from existing results for fixed-priority scheduling based on those presented in [22]. This is due to the fact that we employ their basic analysis in step 6 of Algorithm 3. Replacing this response-time computation with the results in [22] will enable such extensions. In order to simplify the presentation of our results, we avoid a detailed discussion of such extensions.

VI. Implementation

The zero-slack-RM scheduler and its runtime enforcement mechanism have been implemented in Linux/RK [8]. Linux/RK is a resource kernel that creates time (or space) partitions of resources, e.g. CPU, providing temporal isolation between these partitions. These partitions are called resource reserves. Tasks are then associated with these reserves. Time reserves are specified as a consumption time (C) of the resource over a period of time (T). With this specification, Linux/RK ensures that the tasks do not consume more time than the allocated one over the specified period. These reserves are implemented as periodic servers [23] where their budget is replenished periodically and if this budget is exhausted before the end of the period, the associated tasks are stopped. These tasks are resumed once the reserve is replenished.

Our runtime enforcement is implemented as a new type of reserve in Linux/RK with two execution modes. This reserved type is called a *criticality* reserve. A criticality reserve adds two additional parameters to the regular CPU reserve: the criticality (ζ) and the zero-slack instant (ZS), and uses C_i^o as the C consumption time.

A. Zero-Slack Enforcement

The dual execution mode is implemented as a zero-slack enforcement. Specifically, a timer is setup at the start of the period of every task τ_i to expire according at the zero-slack instant. If, when the timer expires, τ_i has not finished its activation, it suspends all the tasks with a criticality lower that its own, thereby entering its C mode. It also sets a global

criticality-level variable to equal its own criticality indicating that no task with lower criticality should be running (storing the previous value as set by any previous task in C mode in a stack). In addition, it puts the suspended tasks in a list that is ordered by criticality. Once τ_i finishes its execution, then it resumes the tasks in C mode for the next criticality level, or all the tasks if no other task is in C mode. Finally, when a reserve is replenished, the criticality level of the associated task is compared against the current criticality level and it is added to the suspended list if it is lower.

VII. Conclusions

In this paper, we studied the problems introduced by mixed-criticality task sets. We also identified the shortcomings of traditional temporal isolation schemes to prevent inter-task interference in mixed-criticality systems. In particular, we characterized and quantified the *criticality inversion* problem that arises when a higher-criticality tasks misses its deadline because it is forced to wait for a lower-criticality task. We then presented a new scheduling scheme with a new protection policy, which we call as *asymmetric protection* that only prevents interference to higher-criticality tasks from lower-criticality tasks. Our scheme has a generic offline zero-slack computation algorithm and runtime enforcement mechanism that can be used with any priority-based preemptive scheduler that has *non-anomalous* scheduling behavior. The other part of our scheme is a slack analysis algorithm that is specific to each priority-based preemptive scheduler. We developed the slack analysis algorithm for RMS and proved multiple properties of the integrated algorithm (*zero-slack RM*). This algorithm provides the same level of protection against criticality inversion as the best known priority assignment for this purpose, criticality as priority (CAPA). Zero-slack RM provides the same level of schedulable utilization as RMS if criticality levels are equal to priorities, and better if they are different and we use our scheduling guaranteed. We also developed two metrics to evaluate such algorithms. First, a *mixed-criticality blocking* (MC blocking) metric measures how much blocking reduction is possible in a specific task set. Secondly, *laxity utilization* measures how well a scheduling algorithm reduces the MC blocking of the task set to make the task set schedulable. These metrics are then used to show that zero-slack RM always outperforms or equals CAPA and RMS to spread the MC blocking reduction of a task set to the tasks that need it the most. Finally, we demonstrate the practicality of our runtime enforcement with an implementation in Linux/RK.

References

[1] D. Homan, "Designing future systems for airworthiness certification," http://www.cse.wustl.edu/~cdgill/CPSWEEK09_MCAR/MCAR%20overview%20BoF%20CPS%20PA%20approved.pdf, 2009.

[2] P. Mejia-Alvarez, R. Melhem, and D. Mosse, "An incremental approach to scheduling during overloads in real-time systems," in *In Proceedings of the 21st, IEEE RTSS*, 2000.

[3] G. Buttazzo, M. Spuri, and F. Sensini, "Value vs deadline scheduling in overload conditions," in *In Proceedings of the 16st, IEEE Real-Time Systems Symposium*, 1995.

[4] G. Koren and D. Shasha, "Dover; an optimal on-line scheduling algorithm for overloaded real-time systems," in *In Proceedings of the IEEE Real-Time Systems Symposium*, 1992.

[5] S. Baruah and J. Haritsa, "Scheduling for overload in real-time systems," *IEEE Transactions on Computers*, vol. 46, no. 9, pp. 1034–1039, 1997.

[6] M. Gardner and L. J.W.S., "Performance algorithms for scheduling real-time systems with overrun and overload," in *In Proceedings of the 11th Euromicro Conference on Real-Time Systems*, 1999.

[7] C.-S. Shih, P. Ganti, and L. Sha, "Schedulability and fairness for computation tasks in surveillance radar systems," in *In Proceedings of the 10th RealTime and Embedded Computing Systems and Applications Conference*, 2004.

[8] S. Oikawa and R. Rajkumar, "Linux/rk: A portable resource kernel in linux," *In Proceedings of the 19th, IEEE Real-Time Systems Symposium*, 1998.

[9] A. Sundaram, P. Chandra, P. Goyal, Shenoy, J. Sahni, and H. Vin, "Application performance in the qlinux multimedia operating system," *In Proceedings of the Eighth ACM Conference on Multimedia*, November 2000.

[10] X. Feng, "Towards real-time enabled microsoft windows," *Proceedings of the 5th International Conference On Embedded Software*, pp. 142–146, 2005.

[11] M. J. Paul, M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera, "Modular real-time resource management in the rialto operating system," *Microsoft Research, Advanced Technology Division*, pp. 12–17, 1995.

[12] G. Buttazzo, G. Lipari, and L. Abeni, "Elastic task model for adaptive rate control," *IEEE Real-Time Systems Symposium*, pp. 286–295, 1998.

[13] R. Davis and A. Wellings, "Dual priority scheduling," *In Proceedings of the 16th IEEE RTSS*, pp. 100–109, Dec 1995.

[14] S. Cho, S.-K. Lee, A. Han, and K.-J. Lin, "Efficient real-time scheduling algorithms for multiprocessor systems," *IEICE Trans. Communications, E85-B(12)*, pp. 2859–2867, Dec 2002.

[15] M. Cirinei and T. Baker, "Edzl scheduling analysis," *In Proceedings of ECRTS*, pp. 9–18, July 2007.

[16] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," *In Proceedings of the Real-Time Systems Symposium (Tucson, AZ, December 2007)*, pp. 239–243, 2007.

[17] L. Sha, J. P. Lehoczky, and R. Rajkumar, "Task scheduling in distributed real-time systems," in *Proceedings of the IEEE Industrial Electronics Conference*. IEEE, 1987.

[18] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications," *Euromicro Conference on Real-Time Systems*, pp. 147–155, July 2008.

[19] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *Computers, IEEE Transactions on*, vol. 39, no. 9, pp. 1175–1185, Sep 1990.

[20] S. Baruah and A. Burns, "Sustainable scheduling analysis," in *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, 2006, pp. 159–168.

[21] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[22] K. Tindell, A. Burns, and A. Wellings, "An extendible approach for analysing fixed priority hard real-time tasks," *The Journal of Real-Time Systems, 6(2)*, pp. 133–152, March 1994.

[23] T. M. Ghazalie and T. P. Baker, "Aperiodic servers in a deadline scheduling environment," *Real-Time Syst.*, vol. 9, no. 1, pp. 31–67, 1995.