Partitioned Scheduling of Multi-Modal Mixed-Criticality Real-Time Systems on Multiprocessor Platforms

Dionisio de Niz
SEI, Carnegie Mellon University

Linh T.X. Phan
University of Pennsylvania

***Abstract*—Real-time systems are becoming increasingly complex. A modern car, for example, requires a multitude of control tasks, such as braking, active suspension, and collision avoidance. These tasks not only exhibit different degrees of safety criticality but also change their criticalities as the driving mode changes. For instance, the suspension task is a critical part of the stability of the car at high speed, but it is only a comfort feature at low speed. Therefore, it is crucial to ensure timing guarantees for the system with respect to the tasks' criticalities, not only within each mode but also during mode changes.**

This paper presents a partitioned multi-processor scheduling scheme for multi-modal mixed-criticality real-time systems. Our scheme consists of a packing algorithm and a scheduling algorithm for each processor that take into account both mode changes and criticalities. The packing algorithm maximizes the schedulable utilization across modes using the *sustained criticality* of each task, which captures the overall criticality of the task across modes. The scheduling algorithm combines Rate-Monotonic scheduling with a mode transition enforcement mechanism that relies on the *transitional zero-slack instants* of tasks to control low-criticality tasks during mode changes, so as to preserve the schedulability of high-criticality tasks. We also present an implementation of our scheduler in the Linux operating system, as well as an experimental evaluation to illustrate its practicality. Our evaluation shows that our scheme can provide close to twice as much tolerance to overloads (ductility) compared to a mode-agnostic scheme.

I. Introduction

Today, it is common to structure the verification of real-time systems based on layers of features of different safety-criticality levels. For instance, the automotive certification standard ISO 26262 [15] identifies four criticality layers (A-D) according to the safety criticality of the features. These layers are characteristics of systems known as *mixed-criticality* systems, in which higher-criticality tasks must be given higher levels of assurance than lower-criticality tasks. At the same time, modern real-time systems are becoming increasingly adaptive. A modern car, for example, needs to adapt its driving *mode* depending on the environmental conditions, and each mode transition may require activations of new tasks, deactivations of existing tasks, or changes of the criticalities of other tasks. This increase in safety requirements and adaptivity in real-time systems introduces a new research question: *How to ensure timing guarantees of tasks with respect to their criticalities – not only in each mode, but also during each mode transition?*

There exists a rich literature on mixed-criticality scheduling (e.g., [3], [4], [8], [18], [19], [23]) and multi-mode analysis (see [32] for a survey) for real-time systems. However, to the best of our knowledge, existing work does not consider mixed-criticality and multi-mode simultaneously. This paper aims to bridge this gap by considering the scheduling of mixed-criticality tasks in multi-modal systems. We use an automotive example that we defined out of features and driving modes present in today's sport vehicles (e.g. active suspension) and real concerns from autonomous driving researchers (e.g. pedestrian detection).

Motivating example. Consider an automotive system that executes four tasks on a two-processor platform: τ_p (pedestrian-detection), τ_s (suspension-control), τ_b (blind-spot warning), and τ_a (air-conditioner control). The system operates in three modes, $\langle \mathsf{Street}, \mathsf{Highway}, \mathsf{IsolatedHOV}^1 \rangle$, and the respective criticalities of each task in these modes are as follows:

$$\tau_p : \langle \text{High, Low, Medium} \rangle; \quad \tau_s : \langle \text{Low, High, High} \rangle;$$
$$\tau_b : \langle \text{High, High, Low} \rangle; \quad \tau_a : \langle \text{Low, Low, Low} \rangle.$$

In the above system, the criticality of a task—and hence the criticality order between tasks—changes when the system changes its mode. This can lead to conflicting schedulability requirements during a mode transition. For instance, τ_p has higher criticality than τ_s in the Street mode but lower criticality than τ_s in the Highway mode; therefore, ensuring the schedulability of τ_p is more important in the Street mode, whereas guaranteeing the schedulability of τ_s is more critical in the Highway mode. This conflict in the scheduling requirements introduces a new challenge: to achieve the schedulability of τ_p in the Street mode, it is necessary to delay or stop τ_s during overloads, but this may make τ_s unschedulable in the Highway mode, if the system enters the Highway mode after τ_s has been delayed for too long.

Another challenge is how to determine a *good* partitioning strategy that can protect high-criticality tasks from missing their deadlines during overloads. One promising approach is to pack tasks that exhibit a mix of high and low criticalities into each processor [19] to enable *overbooking*[2]. However, in multi-modal systems, a packing that is desirable (i.e., that enables overbooking) in one mode may become undesirable in another mode, and this change happens dynamically as the system executes in a sequence of modes.

We demonstrate the above behavior using the two packings in Table I. In the table, tasks within each pair of braces are assigned to the same processor and their criticalities for each mode are shown in the corresponding mode column. It can be observed that P_1 has a good mix of high and low criticalities in the Street mode, but it has a poor criticality mix

[1]In the IsolatedHOV mode, the vehicle travels in an isolated lane, where no passing is possible.

[2]Overbooking is the scheme that uses the scheduled execution budget of a higher-critical task in excess of its nominal execution time (i.e., execution time under overloads) to schedule lower-criticality tasks.

Packing	Mode's Criticality Mix		
	Street	Highway	IsolatedHOV
$P_1 = \langle\{\tau_s, \tau_b\}, \{\tau_t, \tau_a\}\rangle$	{Low, High}, {High, Low}	{High, High}, {Low, Low}	{High, Low}, {Medium, Low}
$P_2 = \langle\{\tau_s, \tau_t\}, \{\tau_b, \tau_a\}\rangle$	{Low, High}, {High, Low}	{High, Low}, {High, Low}	{High, Medium}, {Low, Low}

TABLE I
MULTI-MODE MIXED-CRITICALITY PACKINGS

({High, High}; {Low, Low}) in the Highway mode. Further, although P_1 becomes undesirable when the system switches to the Highway mode, it has a good criticality mix again ({High, Low}; {Medium, Low}) when the system enters the IsolatedHOV mode. In contrast, P_2 has a good criticality mix in both Street and Highway modes, but its criticality mix is worse than that of P_1 in the IsolatedHOV mode ({High, Medium}; {Low, Low}) because the overloading of τ_s makes the medium-criticality task τ_t miss its deadline instead of a low-criticality task, as it happens in P_1.

Contributions. This paper makes the following contributions:

- We present a multi-modal task-to-processor packing algorithm that maximizes the schedulable utilization across all modes. (Section VI)
- We introduce the concept of *transitional zero-slack instant* that enables a mixed-criticality analysis during mode changes, and we describe a method for computing these instants. (Section III-B)
- We present a mode transition enforcement scheme based on transitional zero-slack instants that ensures the absence of criticality violations. (Section IV)
- We present an implementation of our technique on the Linux OS, along with an experiment illustrating its practicality. (Section V)

Our evaluation on synthetic workloads shows that our scheduling scheme can save up to half the number of processors, and provides much better protection to high-criticality tasks during overloads compared to a mode-agnostic scheme. (Section VII)

II. PROBLEM FORMULATION AND APPROACH OVERVIEW

A. System model and problem statement

A multi-modal mixed-criticality system is defined as

$$\mathcal{S} \overset{\text{def}}{=} (\Gamma, \mathcal{P}, \mathcal{M}, \Pi),$$

where Γ is the set of tasks, $\mathcal{P}$ is the set of processors, $\mathcal{M}$ is the set of modes, and $\Pi \subseteq \mathcal{M} \times \mathcal{M}$ is the set of mode transitions of the system. Each mode $m \in \mathcal{M}$ is associated with a set of tasks $\Gamma^m \subseteq \Gamma$ that are active in mode m.

Each task $\tau_i \in \Gamma$ is a periodic task, defined by $\tau_i \overset{\text{def}}{=} (C_i, C_i^o, T_i, D_i, \zeta_i)$, where C_i, C_i^o, T_i, and D_i are the nominal execution time, overload execution time, period, and relative deadline of τ_i, respectively. In addition, $\zeta_i \overset{\text{def}}{=} \{\zeta_i^m \mid m \in \mathcal{M} \wedge \tau_i \in \Gamma^m\}$, where ζ_i^m is the criticality of τ_i in mode m, for all modes m in which τ_i is active. Unlike tasks, jobs have fixed criticalities: the criticality of a job $\tau_{i,k}$ of τ_i is the criticality of τ_i with respect to the mode in which $\tau_{i,k}$ is released.

Categorization of tasks. A task τ_i is called an *old* task of a mode transition $(s,t) \in \Pi$ iff $\tau_i \in \Gamma^s \setminus \Gamma^t$, a *new* task iff $\tau_i \in \Gamma^t \setminus \Gamma^s$, and a *persistent* task iff $\tau_i \in \Gamma^s \cap \Gamma^t$. Since the criticality of a persistent task τ_i may change across the transition, we view it as being made of two separate tasks, τ_i^s and τ_i^t, with the same timing parameters (C_i, C_i^o, D_i, T_i) but different criticality: τ_i^s has criticality ζ_i^s and its jobs are released in s but not in t, and τ_i^t has criticality ζ_i^t and its jobs are released in t but not in s. We call the former an *old-persistent* task and the latter a *new-persistent* task of (s,t).

Mode change semantics. Each transition $(s,t) \in \Pi$ is associated with a mode change request (MCR) that triggers the system to change its current mode from s to t. We assume immediate mode change semantics for new tasks: as soon as the MCR of (s,t) arrives, all new tasks of the transition release their first jobs. In contrast, the first job of each new-persistent task τ_i^t of (s,t) is released exactly T_i time units from the last release of its corresponding old-persistent task τ_i^s prior to the MCR. Further, old and old-persistent tasks have no new releases, and jobs that have been released but not yet completed will continue to be executed after the mode change.

We assume that MCRs may arrive at any time; however, consecutive MCRs are separated enough so that they do not interfere with each other (which is a common assumption in existing multi-mode work [32]). Mode transitions are non-deterministic: if multiple transitions are enabled at the same time, the system non-deterministically selects one.

Criticality violation. A *criticality violation* is said to occur when lower-criticality jobs prevent a higher-criticality job from meeting its deadline[3]; during a mode transition, this applies to only lower-criticality jobs that are (still) active after the MCR. Avoiding criticality violations, both within each mode and during mode changes, is crucial to ensure timing guarantees for higher-criticality jobs.

Objectives. Our goal is to design a scheduling scheme for multi-modal mixed-criticality systems that can maximize the schedulable utilization[4] while ensuring the absence of criticality violations. This goal is driven by the strict ranking requirement of criticality: it is more important to guarantee the schedulability of a higher-criticality job than to guarantee the schedulability of all of its lower-criticality jobs. (As usual, a job is schedulable if its deadline is met, regardless of mode changes.) In this paper, we focus on implicit deadline tasks (i.e., $D_i = T_i$) for simplicity of discussion; however, our method can easily be extended to explicit deadline tasks.

B. Overview of our approach

Our multi-modal mixed-criticality multiprocessor scheduling scheme consists of (i) a mixed-criticality scheduling algorithm that schedules tasks on each processor, and (ii) a multi-modal

[3]This can be caused by a *criticality inversion*, which is said to occur when a lower-criticality job prevents a higher-criticality job from executing.

[4]Schedulable utilization is defined as the maximum utilization under which an algorithm can guarantee the deadlines of any optimally-schedulable task, e.g., the schedulable utilization bound of the Rate-Monotonic algorithm is $\ln 2$ for implicit deadline periodic tasks.

bin-packing algorithm that assigns tasks to processors. We briefly introduce both below.

The mixed-criticality scheduling algorithm schedules tasks on each processor using a multi-mode extension of the zero-slack rate-monotonic (ZSRM) scheduling [8]. Specifically, jobs are scheduled according to the Rate-Monotonic (RM) algorithm; however, lower-criticality jobs will be stopped at absolute *zero-slack instants* of their higher-criticality jobs, i.e., instants from which the higher-criticality jobs must not be preempted by the lower-criticality jobs in order to meet their deadlines under an overload condition. The absolute zero-slack instant of a job is the release time of the job plus the (relative) zero-slack instant of its task, which is computed based on the task's (nominal and overload) execution times and the interference from higher-priority and higher-criticality tasks. In contrast to unimodal systems, the criticality of a task can vary across modes; as a result, its zero-slack instant can be different in each mode and in each mode transition.

Our mixed-criticality scheduling scheme consists of two phases: (1) at design time, it computes the zero-slack instants of each task in each mode (called in-mode zero-slack instant) and in each transition (called *transitional* zero-slack instant); and (2) at runtime, it schedules tasks based on the RM algorithm, while running an enforcement mechanism that stops lower-criticality jobs at the in-mode zero-slack instant (when the system is in a mode) or at the transitional zero-slack instant (during a mode transition) of a higher-criticality job. Sections III and IV discuss these two phases in detail.

The packing algorithm aims to maximize the utilization of the system while enabling overbooking in as many modes as possible. To achieve this, it assigns tasks to processors based on both the tasks' utilizations in each mode and the tasks' *sustained criticalities*. Intuitively, the sustained criticality of a task captures the overall criticality of the task across modes, while ensuring that a higher criticality is more important (i.e., contributes more to the overall criticality) than all lower criticalities. This sustained criticality provides a unified criticality value that can be used to determine how well a combination of tasks on a processor protects high-criticality tasks during overloads across all modes of the system. We present the packing algorithm in Section VI.

III. Multi-Modal Zero-Slack Instants

A. Zero-slack instant within a mode

The zero-slack instant of a task τ_i in a mode s, denoted by Z_i^s, is computed in the same manner as in the single-mode ZSRM [8]. For convenience, we reproduce a summary of the computation here and refer the reader to [8] for the details.

We start by restating the schedulability guarantee under ZSRM, as this influences the amount of preemption suffered by a task. Under ZSRM, a task τ_i is guaranteed to execute for C_i^o within its period T_i if no higher-criticality task τ_j executes for longer than C_j. This guarantee has two implications: First, τ_i is not guaranteed any computation time if any of its higher-criticality tasks τ_j runs for more than C_j; as a result, in the computation of Z_i^s, every higher-criticality task τ_j is considered to execute for only C_j. Second, during the calculation of Z_i^s, every task τ_p with a lower criticality but a higher priority than τ_i must be assumed to run possibly up to its whole overload execution time C_p^o.

Computation of Z_i^s. When the system is in a mode s, each job $\tau_{i,k}$ of τ_i executes in two states: it starts in the nominal state, then switches to the critical state when it reaches its absolute zero-slack instant (which is its release time plus Z_i^s). In the nominal state, $\tau_{i,k}$ is scheduled with all other tasks according to RM, regardless of their criticalities; in the critical state, all tasks of lower criticalities than τ_i are stopped, and $\tau_{i,k}$ is scheduled with only tasks of higher criticalities, also according to RM. As a result, during the nominal state, $\tau_{i,k}$ can suffer preemptions from the *nominal interfering taskset*, denoted by $\Gamma_{i,n}^s$, which consists of all tasks of higher-criticality and all tasks of higher-priority and lower-criticality than τ_i in mode s. In contrast, during the critical state, $\tau_{i,k}$ only suffers preemptions from the *critical interfering taskset*, denoted by $\Gamma_{i,c}^s$, which consists of only higher-criticality tasks of τ_i in mode s. Furthermore, observe that the interference caused by a lower-priority higher-criticality task τ_j can only come from a single job $\tau_{j,h}$ of τ_j (since the tasks are scheduled under RM and thus, τ_j would have a longer period than τ_i); hence, we only need to account for this interference once. Since $\tau_{j,h}$ might arrive during either execution state of $\tau_{i,k}$, the worst case happens when $\tau_{j,h}$ arrives during the critical state of $\tau_{i,k}$, as this would be more likely to make $\tau_{i,k}$ unschedulable.

Based on the above interference on τ_i, we can compute the zero-slack instant Z_i^s by first assuming that τ_i runs completely in the critical state. We then compute the amount of slack (i.e., the sum of the idle time slots) in the active interval – from the release time to τ_i's deadline, assuming that it runs in the nominal state. This slack is used to "move" part of the execution of τ_i from the critical state to the nominal state (thus, deferring the Z_i^s instant). We then repeat this process until no more slack is found in the nominal state, and the final value of Z_i^s is the zero-slack instant of τ_i in mode s.

Preventing back-to-back high-priority preemptions. In [14] the authors pointed out the possibility of back-to-back preemptions in the original ZSRM [8]. This back-to-back preemption can be prevented with a modification to the original enforcement in [8], which is described in Appendix A.

B. Zero-slack instant in a mode transition (s, t)

Let τ_i be a task that is active in mode s or in mode t, i.e., $\tau_i \in \Gamma^s \cup \Gamma^t$. Observe that during a mode transition (s, t), jobs of τ_i may experience interference from both jobs that are released in mode s and jobs that are released in mode t. Therefore, it is not possible to ensure that τ_i does not experience a criticality violation based solely on Z_i^s and Z_i^t. In this section, we propose a new concept, called *transitional zero-slack instant*, that can be used to guarantee the absence of criticality violations for jobs during a mode transition.

First, we define the *transitional interval* of (s, t) as the time interval from the instant the MCR of (s, t) occurs until the earliest instant at which all jobs released in mode s have either completed their executions or missed their deadlines. By definition, the length of the transitional interval of (s,t) is always upper bounded by the maximum relative deadline of all tasks that are active in mode s (i.e., $\max_{\tau_i \in \Gamma^s} D_i$).

Definition 1. *The transitional zero-slack instant of τ_i in a transition (s,t) is the maximum amount of time for which any job $\tau_{i,k}$ of τ_i can be preempted by lower-criticality jobs without making $\tau_{i,k}$ miss its deadline. The (absolute) transitional zero-slack instant of a job $\tau_{i,k}$ is its release time plus the transitional zero-slack instant of τ_i.*

Recall from Section II that, if τ_i is a persistent task of (s,t), its criticality may change across the transition. Therefore, we consider it as two separate tasks: an old-persistent task, which has the old criticality and whose jobs are released in mode s, and a new-persistent task, which has the new criticality and whose jobs are released in mode t. Since the old-persistent task and the new-persistent task may have different sets of lower-criticality tasks, we will compute two different transitional zero-slack instants for each persistent task τ_i of (s,t): the first, denoted by $Z_i^{st,\mathsf{old}}$, will be used to protect jobs of τ_i that are released before the MCR; and the second, denoted by $Z_i^{st,\mathsf{new}}$, will be used to protect jobs of τ_i that are released after the MCR.[5] In contrast, if τ_i is an old task or a new task of (s,t), then it has one transitional zero-slack instant Z_i^{st}, which will be applied to all of its jobs. To compute these transitional zero-slack instants, we first identify the interfering jobsets of τ_i during the transitional interval of (s,t).

C. Interfering jobsets of τ_i during a mode transition (s,t)

Let $\tau_{i,k}$ be any job of τ_i that is active during the transitional interval of (s,t). Under our multi-modal mixed-criticality scheduling, $\tau_{i,k}$ operates in two states: it starts in the nominal state and, when its transitional zero-slack instant is reached, it switches to the critical state. In the nominal state, $\tau_{i,k}$ is scheduled with all active jobs; in the critical state, it is scheduled with only higher-criticality jobs. Below, we quantify the jobs that interfere with $\tau_{i,k}$ in each state. Note that we only consider jobs whose release times or deadlines are within $\tau_{i,k}$'s lifetime (with maximum length of T_i).

Interfering jobset of $\tau_{i,k}$ in the critical state: Its interfering jobset effectively consists of only higher-criticality jobs that are (still) active after the MCR. In other words, this set consists of (i) at most one job of each higher-criticality task τ_j of τ_i in mode s (i.e., $\tau_j \in \Gamma_{i,c}^s$), and (ii) all jobs of each higher-criticality task τ_h of τ_i in mode t (i.e., $\tau_h \in \Gamma_{i,c}^t$).

Interfering jobset of $\tau_{i,k}$ in the nominal state: The jobs that interfere with $\tau_{i,k}$ in its nominal state depends on τ_i's type:

- *If τ_i is an old or old-persistent task:* Then $\tau_{i,k}$ is released in mode s, and thus its interfering jobset consists of (i) all jobs of each task τ_j of higher-criticality or higher-priority than τ_i in mode s (i.e., $\tau_j \in \Gamma_{i,n}^s$) that are released before the MCR and (ii) all jobs of each task τ_h of higher-criticality or higher-priority than τ_i in mode t (i.e., $\tau_h \in \Gamma_{i,n}^t$) that are released after the MCR.
- *If τ_i is a new or new-persistent task:* Then $\tau_{i,k}$ is released in mode t, and thus its interfering jobset consists of (i) at most one job of each task $\tau_j \in \Gamma_{i,n}^s$ and (ii) all jobs of each task $\tau_h \in \Gamma_{i,n}^t$.

Based on the above observations, we can derive the maximum critical/nominal interfering jobset of each task τ_i (to be applied to its jobs in their critical/nominal states) by taking the largest set of interfering jobs that any job of τ_i can experience during its lifetime. Algorithm 1 gives the pseudocode for computing the critical/nominal interfering jobset for each task τ_i that is active during the mode transition (i.e., $\tau_i \in \Gamma^s \cup \Gamma^t$) based on its type. Specifically, for each old or new task (Lines 2–4 and 10–12), the algorithm computes one pair of interfering jobsets, $\langle J_{i,c}^{st}, J_{i,n}^{st} \rangle$, to be used for computing Z_i^{st}. In contrast, for each persistent task (Lines 5–9), the algorithm computes two pairs of interfering jobsets, $\langle J_{i,c}^{st,\mathsf{old}}, J_{i,n}^{st,\mathsf{old}} \rangle$ and $\langle J_{i,c}^{st,\mathsf{new}}, J_{i,n}^{st,\mathsf{new}} \rangle$, to be used for computing $Z_i^{st,\mathsf{old}}$ and $Z_n^{st,\mathsf{new}}$, respectively.

Algorithm 1 ComputeInterferingJobSets(Γ^s, Γ^t)

```
1:  for all $\tau_i \in \Gamma^s \cup \Gamma^t$ do
2:      if $\tau_i \in \Gamma^s \setminus \Gamma^t$ then
3:          $J_{i,c}^{st} \leftarrow$ GetOneJobPerTask$(\Gamma_{i,c}^s) \cup$ GetJobsInInterval$(0, T_i, \Gamma_{i,c}^t)$
4:          $J_{i,n}^{st} \leftarrow$ GetMaxHybridJobSet$(i, \Gamma_{i,n}^s, \Gamma_{i,n}^t)$
5:      if $\tau_i \in \Gamma^s \cap \Gamma^t$ then
6:          $J_{i,c}^{st,\mathsf{old}} \leftarrow$ GetOneJobPerTask$(\Gamma_{i,c}^s) \cup$ GetJobsInInterval$(0, T_i, \Gamma_{i,c}^t)$
7:          $J_{i,n}^{st,\mathsf{old}} \leftarrow$ GetMaxHybridJobSet$(i, \Gamma_{i,n}^s, \Gamma_{i,n}^t)$
8:          $J_{i,c}^{st,\mathsf{new}} \leftarrow$ GetOneJobPerTask$(\Gamma_{i,c}^s) \cup$ GetJobsInInterval$(0, T_i, \Gamma_{i,c}^t)$
9:          $J_{i,n}^{st,\mathsf{new}} \leftarrow$ GetOneJobPerTask$(\Gamma_{i,n}^s) \cup$ GetJobsInInterval$(0, T_i, \Gamma_{i,n}^t)$
10:     if $\tau_i \in \Gamma^t \setminus \Gamma^s$ then
11:         $J_{i,c}^{st} \leftarrow$ GetOneJobPerTask$(\Gamma_{i,c}^s) \cup$ GetJobsInInterval$(0, T_i, \Gamma_{i,c}^t)$
12:         $J_{i,n}^{st} \leftarrow$ GetOneJobPerTask$(\Gamma_{i,n}^s) \cup$ GetJobsInInterval$(0, T_i, \Gamma_{i,n}^t)$
```

Algorithm 2 GetMaxHybridJobSet$(i, \Gamma_{i,n}^s, \Gamma_{i,n}^t)$

```
1:  $V_{i,n}^s \leftarrow$ GetSlackVector$(i, \Gamma_{i,n}^s, Z_i^s)$
2:  $V_{i,n}^t \leftarrow$ GetSlackVector$(i, \Gamma_{i,n}^t, Z_i^t)$
3:  $\mathsf{MCRI}_{\max} \leftarrow 0$
4:  minSlack $\leftarrow \infty$
5:  SlackStarts $\leftarrow$ GetSlackStartings$(V_{i,n}^s, V_{i,n}^t)$
6:  for all $p \in$ SlackStarts do
7:      currentSlack $\leftarrow$ GetSlackInInterval$(0, p, V_{i,n}^s)$ +
                        GetSlackInInterval$(p, T_i, V_{i,n}^t)$
8:      if currentSlack $<$ minSlack then
9:          minSlack $\leftarrow$ currentSlack
10:         $\mathsf{MCRI}_{\max} \leftarrow p$
11: return  GetJobsInInterval$(0, \mathsf{MCRI}_{\max}, \Gamma_{i,n}^s) \cup$
            GetJobsInInterval$(\mathsf{MCRI}_{\max}, T_i, \Gamma_{i,n}^t)$
```

In Algorithm 1, the critical interfering jobsets for all tasks and the nominal interfering jobsets of new and new-persistent tasks are computed based directly on the above discussion. However, computing the nominal interfering jobset of an old or old-persistent task (Lines 4 and 7) is more challenging, since this set highly depends on the instant at which the MCR occurs. Since this instant is unknown at design time, we first compute the worst-case MCR instant, $\mathsf{MCRI}_{\max}$, that leads to the maximum interference on a job of τ_i during its nominal and critical states, and then use $\mathsf{MCRI}_{\max}$ to

[5]For simplicity of exposition, we use the same notation τ_i for both of its corresponding old-persistent and new-persistent tasks, but distinguish the two by explicitly referring to their task types.

derive the worst-case interfering jobset. This is done via the function GetMaxHybridJobSet, shown in Algorithm 2. (The key functions in Algorithm 2 are explained in Appendix B.)

Based on the computation of the interfering jobsets of τ_i, we can now derive its transitional zero-slack instant(s).

D. *Computation of transitional zero-slack instants*

Algorithm 3 shows the pseudocode for computing the transitional zero-slack instant Z_i^{st} of each task τ_i, based on its interfering jobsets. Here, taskset (Line 5) is the set of old, old-persistent, new, and new-persistent tasks of (s, t). That is, we maintain in taskset two separate tasks, old-persistent and new-persistent, for each persistent task of (s, t). Thus, the algorithm will compute two separate transitional zero-slack instants for each persistent task of (s, t), as discussed earlier. Note that we use $J_{i,n}^{st}$ and $J_{i,c}^{st}$ to refer to the nominal and critical jobsets of every task τ_i in taskset, including when τ_i is an old-persistent task or a new-persistent task.

Algorithm 3 ComputeTransitionalZeroSlackInstants(s, t)

```
1:  ∀i, $Z_i^{st,1} \leftarrow 0$
2:  ComputeInterferingJobSets($\Gamma^s, \Gamma^t$)
3:  repeat
4:      ∀i, $Z_i^{st,0} \leftarrow Z_i^{st,1}$
5:      for all $\tau_i$ in taskset do
6:          $V_{i,n}^{st} \leftarrow$ GetSlackVector($i, J_{i,n}^{st}, Z_i^{st,0}$)
7:          $V_{i,c}^{st} \leftarrow$ GetSlackVector($i, J_{i,c}^{st}, Z_i^{st,0}$)
8:          $Z_i^{st,1} \leftarrow$ GetSlackZeroInstant($i, V_{i,n}^{st}, V_{i,c}^{st}, Z_i^{st,0}$)
9:  until ∀i, $Z_i^{st,0} = Z_i^{st,1}$
10: ∀i, $Z_i^{st} \leftarrow Z_i^{st,1}$
```

At a high level, Algorithm 3 works as follows. It begins with the worst-case assumption that there is no slack in the nominal state of each task, and each task always needs to run in the critical state. Then, it starts moving computation time from the critical state to the nominal state as we discover slack in the nominal state (using Algorithm 4). It then repeats the process with the newly-computed zero-slack instant. Note that, even though we only calculate the interfering jobset at the beginning of the algorithm, the proportions of computation of the interfering jobs that occur during the nominal state (C^n) and during the critical state (C^c) change as we recalculate the zero-slack instants (as we will show in Algorithm 4). The algorithm converges when it cannot move the zero-slack instant of any task towards its deadline.

To achieve the above, the algorithm maintains for each task τ_i two variables, $Z_i^{st,0}$ and $Z_i^{st,1}$, which store the zero-slack instant values before and after an iteration, respectively. Both $Z_i^{st,0}$ and $Z_i^{st,1}$ are initialized to zero (to indicate that there is no slack in the nominal state). In each iteration (Lines 3–9), the algorithm first calculates the slack vectors for τ_i in the nominal and critical states ($V_{i,n}^{st}$ and $V_{i,c}^{st}$), based on τ_i's nominal and critical interfering jobsets (Lines 6–7). Each slack vector contains a sequence of slack regions ordered by time; each slack region consists of a starting instant and a duration. Based on the obtained slack vectors, the algorithm then computes a new zero-slack instant ($Z_i^{st,1}$) for τ_i using the function GetSlackZeroInstant$(i, V_{i,n}, V_{i,c}, Z_i^{st,0})$. With the new zero-slack instant, the algorithm proceeds to the next iteration and continues in the same manner until it converges ($\forall i,\ Z_i^{st,0} = Z_i^{st,1}$).

Algorithm 4 GetSlackZeroInstant(i, V_n, V_c, t_0)

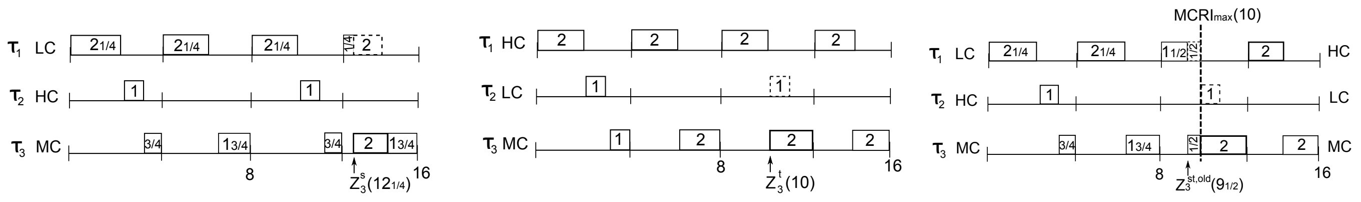
```
1:  $C_i^c \leftarrow C_i^o$ ; $C_i^n \leftarrow 0$
2:  repeat
3:      $t_1 \leftarrow$ StartOfTrailingSlack($i, C_i^c, V_c$)
4:      if $t_1 \geq 0$ and $t_1 \leq t_0$ then
5:          $k_u \leftarrow$ GetSlackInInterval($0, t_1, V_n$) $- C_i^n$
6:          $k_u \leftarrow \max(\min(k_u, C_i^c), 0)$
7:          $C_i^c \leftarrow C_i^c - k_u$
8:          $C_i^n \leftarrow C_i^n + k_u$
9:      else
10:         $k_u \leftarrow 0$
11: until $k_u = 0$
12: return $t_1$
```

Algorithm 4 shows the pseudocode for the function GetSlackZeroInstant(i, V_n, V_c, t_0). Intuitively, this function divides the overload execution time C_i^o of τ_i into two parts: C_i^c, the execution time in the critical state, and C_i^n, the execution time in the nominal state. Initially, all execution time is in the critical state ($C_i^c = C_i^o$) and the critical state starts at time t_0. The function then repeatedly (i) computes a new starting time t_1 for the critical state (Line 3) using StartOfTrailingSlack(i, C_i^c, V_c), which gives the latest instant where there are at least C_i^c units of slack within V_c (the slack vector of τ_i in the critical state) that is before T_i, and then (ii) moves the corresponding amount of execution time (k_u in Lines 5–6) from the critical state to the nominal state (Lines 7–8). The algorithm terminates when no more computation can be moved ($k_u = 0$), and it returns the final value of t_1 as the new value for $Z_i^{st,1}$ in Line 8, Algorithm 3.

Example 1. To illustrate the above computation, consider a system with two modes, s and t, that executes three (persistent) tasks: $\tau_1 = \big(T_1 = 4, C_1 = 2, C_1^o = 2.25, \zeta_1^s = 1, \zeta_1^t = 3\big)$; $\tau_2 = \big(T_2 = 8, C_2 = 1, C_2^o = 1, \zeta_2^s = 3, \zeta_2^t = 1\big)$; and $\tau_3 = \big(T_3 = 16, C_3 = 1, C_3^o = 7, \zeta_3^s = 2, \zeta_3^t = 2\big)$. Recall that in the transition (s, t), each task τ_i has two transitional zero-slack instants, $\tau_i^{st,\mathsf{old}}$ and $\tau_i^{st,\mathsf{new}}$. We will illustrate the computation of the transitional zero-slack instant $Z_3^{st,\mathsf{old}}$ of τ_3.

Figures 1(a), 1(b) and 1(c) depict the timeline of τ_3 with its zero-slack instants in mode s, in mode t, and in the transition (s, t), respectively. A solid box represents an actual execution, whereas a dashed box represents an execution that would have happened if the task had not been preempted or stopped.

To compute the interfering jobsets of the old-persistent task τ_3 (c.f. Algorithm 1), we first derive the worst-case MCR instant $\mathsf{MCRI}_{\max}$ that leads to the maximum interference on τ_3 during its nominal state and its critical state (c.f. Algorithm 2). It can be observed from Figures 1(a) and 1(b) that, during the nominal state, τ_3 suffers more interference from τ_1 in mode s than in mode t. This is because τ_1 has lower criticality than τ_3 in mode s and thus, its interference includes all of its overload execution time ($C_1^o = 2.25$); however, it has higher criticality than τ_3 in mode t and hence, its interference includes only its nominal execution time ($C_1 = 2$). Observe also that the interference from τ_2 during τ_3's nominal state is the same in



(a) Source Mode Timeline (b) Target Mode Timeline (c) Mode Transition Timeline

Fig. 1. Transitional Zero-Slack Computation Example

both modes (since $C_i^o = C_i$). Therefore, the interference from τ_1 and τ_2 in mode s is larger than their interference in mode t. On the contrary, in the critical state, τ_3 experiences more interference in mode t than in mode s, because τ_2 is stopped in mode t (which frees only 1 unit of execution) whereas τ_1 is stopped in mode s (which frees 2 units of execution).

The above interference patterns lead to the worst-case MCR instant at the zero-slack instant of τ_3 in mode t ($\mathsf{MCRI}_{\max} = 10$), as shown in Figure 1(c). Based on the obtained $\mathsf{MCRI}_{\max}$ and the corresponding interfering jobset of τ_3 in the nominal state, using Algorithm 3 we can derive the transitional zero-slack instant of τ_3 ($Z_3^{st,\mathsf{old}} = 9.5$).

IV. Run-time scheduling with enforcement

On each processor, our multi-modal mixed-criticality scheduler schedules tasks according to RM, and it uses an enforcement mechanism that stops (or drops) jobs based on the in-mode and transitional zero-slack instants of tasks. (The enforcement stops or drops a job following the strategy in Appendix A, but we simply use the term 'stop' for convenience.)

Recall from the definition of the transitional interval of a transition (s,t) that, from the end of the transitional interval onward, the system contains only jobs of new or new-persistent tasks (released in mode t). It can easily be shown that, if a job of τ_i is released after or at the end of the transitional interval of (s,t), the maximum interference it experiences over any interval is the same as the maximum interference a job of τ_i would have experienced in an interval of the same length if the system had operated in a single mode t. Therefore, we can safely use the zero-slack instants of tasks in mode t for the enforcement after the transitional interval; consequently, the transitional zero-slack instant is only applicable to jobs that are released before the end of the transitional interval of (s,t).

Enforcement in a mode. After the system has transitioned to a mode s and the transitional interval has ended, enforcement works as follows: whenever an active job of each task τ_i reaches its in-mode zero-slack instant Z_i^s, all released jobs of tasks with lower-criticality than τ_i in mode s are stopped.

Enforcement upon an MCR. When the system is in mode s and the MCR associated with an outgoing transition (s,t) arrives, the system will enter the new mode (c.f. Section II) and perform the enforcement instantaneously. The enforcement is only performed on jobs that are active during the transitional interval of (s,t), and the specific action depends on whether the job arrives before or after the MCR. Specifically, let MCRI be the arrival time of the MCR with respect to the release time of an active job $\tau_{i,k}$ during the transitional interval. Then, the enforcement on $\tau_{i,k}$ works based on the transitional zero-slack instant Z_i^{st} of $\tau_{i,k}$ as follows:

- If $\mathsf{MCRI} \geq Z_i^{st}$, then the system immediately stops all the jobs with lower-criticality than $\tau_{i,k}$.[6]
- If $\mathsf{MCRI} < Z_i^{st}$, then the system waits until the transitional zero-slack instant Z_i^{st} to stop all jobs with lower-criticality than $\tau_{i,k}$.

The next lemma states the correctness of the enforcement; its proof is available in Appendix C.

Lemma IV.1. *The above run-time enforcement guarantees that no job experiences criticality violations.*

V. Implementation

We implemented our multi-modal zero-slack scheduler in a Linux kernel module along with a user-level library to support the development of multi-modal mixed-criticality systems. An empirical evaluation of the run-time overhead of our implementation is available in Appendix D.

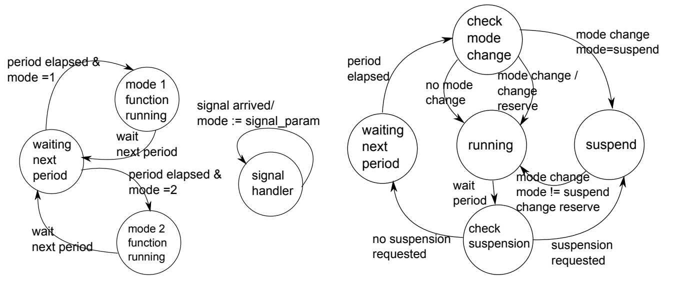
A. Modal system

Our user-level library provides abstractions to build modal systems that consist of a set of modal tasks and a system mode transition manager. A modal task, in turn, is composed of a collection of *modal job functions* (one for each mode), a set of mode transitions, and an initialization function (`init()`). A modal task is created by calling the function `start_modal_task()` with a modal task structure that contains pointers to the mode job functions, the `init()` function, and the mode transitions (to be discussed shortly). This library function forks a process and returns the process ID of the newly-created process.

The main body of a modal task first calls the `init()` function and then installs the mode transition signal handler. After that, it enters an infinite loop that executes the job function repeatedly. In each iteration, the loop first calls the job function and then waits for the arrival of the next period (`wait_for_next_period()`) before calling the job function again[7]. The signal handler function is hidden

[6]Note that if $\mathsf{MCRI} > Z_i^{st}$ and the criticality of $\tau_{i,k}$ is lower than that of some new job(s) released after the MCR, then $\tau_{i,k}$ may not have enough time to finish by its deadline if it executes for all of its C_i^o, due to interference of these higher-criticality new job(s). However, since all active lower-criticality jobs of $\tau_{i,k}$ are stopped, $\tau_{i,k}$ will not experience a criticality violation.

[7]While our current implementation waits for the next period, this can be easily changed to a generic arrival triggering function that waits for external events, such as interrupts.



(a) Library (b) Scheduler

Fig. 2. Mode Change State Machines

inside the runtime library; it only changes the `mode` variable to the mode number that is sent by the mode transition manager.

We make two important observations about the mode change signal handling: First, the signal handler runs inside the modal task process under the scheduling parameters of that process; hence, it does not interfere with other tasks. Second, for tasks that are currently active, the new mode is not effective until the next arrival of the job; that is, the lifetime of the current job (from the current time to the next arrival) is not modified. Instead, the new mode is registered within the signal handler, which will be used upon the next arrival (within the `wait_for_next_period()` function). At the next arrival, the new job function is executed, and the kernel scheduler then waits for the new period of the task.[8] This behavior is illustrated by the finite state machines in Figure 2(a).

Mode transitions. Task mode transitions are part of the modal task structure, which contains, for each transition, the source mode number, the target mode number, and the computed transitional zero-slack instant(s) (c.f. Section III-B).

The final modal system is composed of system modes and system mode transitions controlled by the system mode manager. System modes are composed of a collection of task modes that each contain a process ID and a mode number. System mode transitions bundle tasks' mode transitions. The system mode manager triggers system mode transitions by sending mode switch signals to each of the tasks, and then instructs the kernel scheduler to start the transition, which activates the corresponding transitional zero-slack instants. The system mode manager runs as a persistent task across all modes, with the highest criticality and highest priority.

B. Modal kernel scheduler

The kernel scheduler is a reservation-based scheduler that assigns modal zero-slack reserves to each task. A modal reserve is a collection of regular zero-slack reserves along with transitions. A modal reserve is first created as an empty reserve, and then zero-slack reserves are created and added to the modal reserve. Once all the modal reserves have been created, the mode transitions are added with their corresponding transitional zero-slack instant(s). Finally, a modal reserve is attached to a task upon its creation.

A special `SUSPENSION` mode is used to represent that a task is inactive. The kernel scheduler uses this mode to suspend the task when it needs to be disabled in a particular system mode. A task transitioning to the suspension mode is disabled at the next period arrival (as expected). However, when a task transitions from the suspension mode to an active mode, the task is activated immediately, as required by our mode change semantics (c.f. Section II). This is illustrated by the finite state machine in Figure 2(b).

[8]Our implementation allows tasks to change their periods across modes.

Our implementation uses a kernel scheduler thread that changes the priorities and the scheduling policy of the tasks in response to scheduling events and the current mode, following the strategy of the non-intrusive kernel module in [9]. This kernel thread is necessary to avoid changing the priority inside the timer handler (triggered by an interrupt) in the kernel module to avoid deadlocks, since changing the priority modifies the priority queues. This technique is also used in Linux/RK [26] and a similar kernel module scheduler in [17].

C. Mode change experiment timeline

To show the effect of mode changes on our system, we created an experiment with three tasks (see Table II): a Suspension task (τ_s), a Pedestrian Detection task (τ_p) and a (Coordinated) Cruise Control task (τ_c). Each task can run in two modes, Low Speed and High Speed. We designed the experiment to focus our attention on the zero-slack instant of the task τ_s, which was calculated at 13s for both the mode transition and the target mode (High Speed).

Task	T(s)	C(s)	C^o(s)	Criticality	
				Low Speed	High Speed
Suspension (τ_s)	16	4	7	0	2
Pedestrian Detect. (τ_p)	4	2	2	2	0
Coord. Cruise Ctl. (τ_c)	8	1	1	1	1

TABLE II
SAMPLE EXPERIMENTAL CAR TASKSET

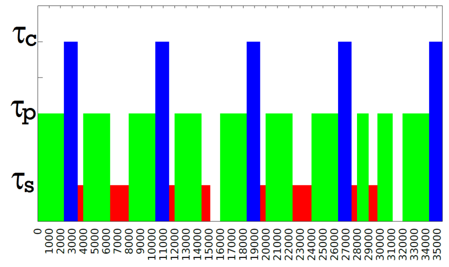


Fig. 3. Mode Change Timeline (ms)

Figure 3 shows the timeline of a change from the Low Speed mode to the High Speed mode in the experimental task. The X-axis shows time (in msec), and the Y-axis shows the task that is active at the corresponding time (e.g., τ_p is active from 0 to 2000, as is shown by the solid bar that rises to τ_p's marker). The mode change was requested roughly at time t = 15s, which is close to the end of the first hyperperiod (at t = 16s). We observe the following effect of this mode change request from the timeline: Even though all old jobs have finished, the new jobs are only released at the next periods of the tasks (at t = 16s); this is consistent with our mode change semantics for persistent tasks. In addition, when task τ_s overruns its zero-slack instant at t = 29s (= 16s +13s), task τ_p is stopped to let

τ_s complete; this illustratres the effect of the enforcement. It is worth highlighting that, around the same relative time from the start of the first hyperperiod in the source mode (i.e., at t = 13s), τ_p was not preempted by τ_s, because τ_p has higher criticality and higher priority than τ_s in the source mode.

VI. Multi-modal Mixed-Criticality Bin-Packing

As discussed in Section II-B, the main objective of our packing algorithm is to maximize the utilization of the system while enabling overbooking in as many modes as possible. Towards this, we first define the *sustained criticality* of a task τ_i as the overall criticality of the task across all modes, given by:

$$\zeta_i^* \overset{\text{def}}{=} \sum_{m \in \mathcal{M}} \left(|\mathcal{M}| + 1\right)^{\zeta_i^m}, \tag{1}$$

where $\mathcal{M}$ is the set of modes of the system. Informally, each summand in Eq. (1) captures the degree of importance (in terms of schedulability requirements) of the task's criticality in each mode m. It is defined as the number of modes plus one to the power of the task's criticality in mode m, so as to preserve the strict ranking of criticality (i.e., a higher-criticality task is more important than any number of lower-criticality tasks).

In addition, we define the *dominant criticality* of τ_i as its highest criticality in all modes, i.e., $\widehat{\zeta_i} = \max_{m \in \mathcal{M}} \zeta_i^m$.

Algorithm 5 Multi-modal mixed-criticality packing algorithm.

```
1:  $\mathcal{P} \leftarrow \{p_1, p_2, \ldots, p_k\}$, where $k = \max_{m \in \mathcal{M}} \{ \lceil \sum_{\tau_i \in \Gamma^m} \frac{C_i}{T_i} \rceil \}$
2:  $\Gamma_{\text{rest}} \leftarrow \emptyset$
3:  for all $\tau_i \in \Gamma$ in non-increasing $\zeta_i^*$ and non-increasing $\frac{C_i^o}{T_i}$ do
4:      deployed $\leftarrow$ false
5:      for all $p_j \in \mathcal{P}$ in non-increasing fullness level of $p_j^m$ where $\zeta_i^m = \widehat{\zeta_i}$ do
6:          if overloaded($\tau_i$) schedulable in overloaded($p_j$) then
7:              deployed $\leftarrow$ true
8:              break
9:      if ¬deployed then
10:         $\Gamma_{\text{rest}} \leftarrow \Gamma_{\text{rest}} \cup \tau_i$
11: for all $\tau_i \in \Gamma_{\text{rest}}$ do
12:     deployed $\leftarrow$ false
13:     while ¬deployed do
14:         for all $p_j \in \mathcal{P}$ in non-increasing fullness level of packed($p_j^m$)
            where $\zeta_i^m = \widehat{\zeta_i}$ do
15:             if ¬overloaded($\tau_i$) is schedulable in packed($p_j$) then
16:                 deployed $\leftarrow$ true
17:                 break
18:         while ¬deployed do
19:             $\mathcal{P} \leftarrow \mathcal{P} \cup$ new($p_{\text{new}}$)
20:             if overloaded($\tau_i$) schedulable in overloaded($p_{\text{new}}$) then
21:                 deployed $\leftarrow$ true
```

At a high level, the packing is performed by verifying that the total requirements of the tasks allocated to a processor do not exceed the capacity of that processor. For this, we describe the *fullness level* of a processor j as a vector $p_j = \langle p_j^1, p_j^2, \ldots, p_j^{|\mathcal{M}|} \rangle$, where p_j^m is the fullness level of the processor in mode m, all for $m \in \mathcal{M}$. The fullness vectors of the processors are then used together with the sustained and dominant criticalities of the tasks to order processors and tasks in our allocation algorithm. We refer to this algorithm as *vector Mixed-Criticality Packing* (vMCP), since it is based on vector packing, with processors modeled as vectors of capacities (one for each mode) and tasks modeled as vectors of requirements. The pseudocode for vMCP is shown in Algorithm 5.

Algorithm 5 starts by creating an initial set of processors $\mathcal{P}$ whose combined capacity is equal to the largest taskset utilization across all modes (Line 1). It then proceeds to the first stage (Lines 2–10), in which it attempts to assign tasks to the processors in a non-increasing order of sustained criticality and utilization, assuming that each task τ_i runs for its overload execution time C_i^o. In this order, each τ_i is tried on each of the processors in $\mathcal{P}$, ordered in non-increasing levels of fullness in the mode m with the dominant criticality of τ_i (i.e., $\zeta_i^m = \widehat{\zeta_i}$). The tasks that cannot be fitted on these processors are then set aside in a set Γ_{rest} to be deployed in the second stage.

In the second stage (Lines 11–21), the algorithm deploys the tasks in Γ_{rest} to the processors in $\mathcal{P}$ using the same orders of tasks and processors as in the first stage, but assuming that the tasks that have already been deployed in the first stage only run for their nominal execution times (referred to as packed() in Lines 14–15). If a task cannot be fitted onto any processor, a new processor is added (Line 19), and the algorithm then tries to deploy this task on the new processor, assuming that the task overruns (Lines 20–21).

It can be observed that the resulting packing always ensures all tasks are schedulable if the system does not overload.

Note that Algorithm 5 follows a best-fit strategy when assigning a task to a processor. However, it can easily be modified to use other variants, such as first-fit, worst-fit, etc. Due to space constraints, we omit them here.

VII. Evaluation

In this section, we present an evaluation of the proposed vMCP algorithm using synthetic real-time workload. Our goal is to evaluate how well the vMCP performs across different criticality variations of the system compared to a baseline mode-agnostic vector packing algorithm [7]. (To the best of our knowledge, there exists no mode-aware packing algorithm for multi-modal mixed-criticality systems).

A. Evaluation metrics: ductility and resource use

Observe that the conventional schedulability metric cannot fully characterize the scheduling performance in a mixed-criticality system, since two schedulers may both feasibly schedule the system under nominal execution but one may guarantee more schedulable higher-criticality tasks than the other under an overload condition. To address this, we adapt the *ductility* metric [19] to evaluate the capacity of a scheduler in protecting high-criticality tasks during overloads, while extending this metric to the multi-modal setting.

Formally, let k be the number of distinct criticality levels of the tasks in the system, and assume without loss of generality that all tasks have criticalities between 1 and k. To define the ductility of a schedule in a mode, we consider all possible overloading scenarios of tasks in each criticality level j. Each scenario O_l is a k-tuple $O_l = \langle O_{l,k}, \ldots, O_{l,1} \rangle$, where $O_{l,j} = 0$ if no tasks with criticality j overload, and $O_{l,j} = 1$ otherwise. (Since there are k criticality levels, there are 2^k scenarios in total.) Corresponding to each overloading scenario O_l, we define for every criticality level j a ductility value $d_{l,j}$, which characterizes the possibility that all tasks with criticality j meet their deadlines under the overloading scenario O_l. That is, for all $1 \leq j \leq k$, $d_{l,j} = 1$ if all tasks with criticality j are guaranteed to meet their deadlines under the overloading scenario O_l, and $d_{l,j} = 0$ otherwise. The ductility of a schedule in a mode m can then be computed by aggregating all of its ductility values $d_{l,j}$ in mode m, scaled by an order of magnitude between criticality levels:

$$\mathsf{ductility}(m) = \sum_{j=1}^{k} \Big(\frac{\sum_{l=1}^{2^k} d_{l,j}}{2^{kj} + 1} \Big).$$

The overall ductility of a schedule of a multi-modal system is then given as the sum of all the ductilities $\mathsf{ductility}(m)$ for all modes m. It can be observed that this ductility metric preserves the strict ranking semantics of criticality, while enabling comparison between different schedules during overloads.

B. Workload

In our experiments, we first generated five different types of harmonic periods[9], sampled from a uniform distribution between 100 and 1000, which were used for all the tasks in our experiments. For each generated period, we selected the overloaded execution time uniformly at random between 10% and 30% of the period. Finally, the nominal execution time was selected uniformly at random between 50% and 100% of the overloaded execution time. This enables us to explore tasksets that do not suffer the internal fragmentation produced by heavy tasks (e.g., a task that can consume close to, or over 50% of the processor and ends up being packed on a processor by itself) but still allows for a reasonable amount of overbooking due to the difference between the nominal and the overload execution time.

Next, we generated all possible variations of criticality-to-priority relationships. These relationships were encoded in a criticality vector, whose first, second, and third entry represent the criticality assigned to the task with the highest, second-highest, and lowest priority, respectively. In our experiments, we used three modes and generated one criticality vector per mode. These vectors are shown on the X-axis of Figure 4. (In the figure, we only show a subset of the generated criticality vectors to highlight interesting cases where tasks with different priorities have different criticalities.)

C. Evaluation results

Resource savings. Figure 4 shows the number of processors needed under the vMCP algorithm and under the baseline vector packing with respect to different criticality vectors. The data presented in the figure is the average of 100 experiments.

We observe the following from the results: First, vMCP is able to reduce the number of required processors, independent of the criticality vector. This is because vMCP is able to "follow" the size of the tasks and the fullness of the processors according to the criticality, whereas the baseline algorithm frequently selects the wrong criticality (as it is agnostic to

[9]Note that it is common for a system to have harmonic periods, such as Boeing avionics [11] and ARINC 653 [1] systems.

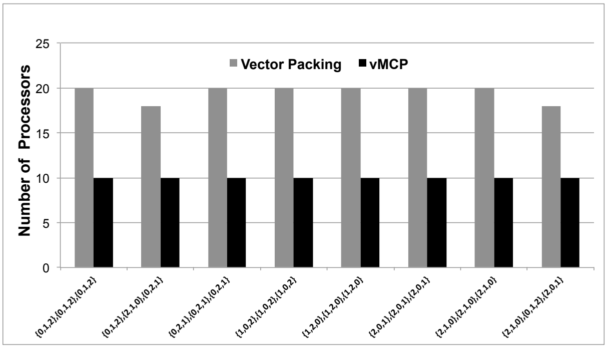


Fig. 4. vMCP vs Vector Packing Across Criticality Variations

criticality changes across modes). Second, unlike vMCP, the baseline packing uses more or fewer processors depending on the alignment of the overall processor fullness across modes and the criticality of the task being deployed; this is the case for the second and the last columns. Finally, in most criticality variations, vMCP is able to reduce the number of processors by half, except for the two of the cases shown in the figure (corresponding to the second and the last criticality vectors).
Ductility. In the second experiment, we reduced the number of processors that the two packing algorithms were allowed to allocate, to compare how well they can preserve ductility. Figure 5 shows the ductility values (normalized to the maximum possible for the taskset) that vMCP and the baseline vector packing algorithm can achieve for the same taskset if the number of processors that they are able to allocate is reduced. This figure presents the average of 100 data points with tasksets generated in the same fashion as in the previous experiment, with the criticality vector $\{\{2, 1, 0\}, \{0, 1, 2\}, \{2, 0, 1\}\}$.

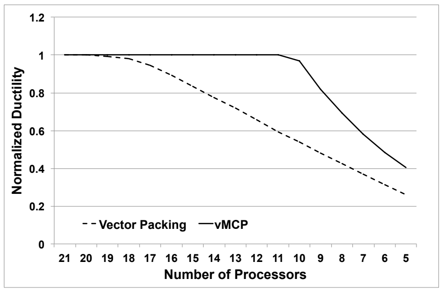


Fig. 5. Ductility vs. Number of Processors

We can make three important observations from Figure 5: First, vMCP keeps a higher ductility in all data points. Second, the gap between vMCP and the normal vector packing increases at first but later on becomes smaller, as the number of processors becomes too small to produce a significant difference. Finally, vMCP can achieve up to nearly twice the ductility obtained by the baseline vector packing algorithm.

VIII. Related Work

Timing analysis of multi-mode systems has been extensively studied within the context of uniprocessors (see e.g., [10], [12], [13], [27]–[30], [32], [33], [35], [36]) and it has recently extended to multiprocessor platforms [20], [21], [24], [25]. The primary focus of this line of work has been on designing mode change protocols that ensure schedulability in each mode and during mode transitions (see [32] and references therein). Formalization of mode change semantics for uniprocessor setting has also been considered in [31]. However, none of the existing solutions consider mixed-criticality constraints.

Several existing techniques also consider overload scheduling. However, they either focus on online overload management [6], [22] and proportions of resource allocation (instead of deadlines met) [34] or provide a tradeoff encoding, such as the elasticity concept in [5], which cannot encode the mixed-criticality semantics. In contrast, our approach provides a graceful degradation mechanism that guarantees that, in an overload situation, deadlines can only be missed in reverse order of criticality.

In the mixed-criticality domain, Baruah et al. [2]–[4] proposed an alternative mixed-criticality model where tasks have different execution times at different criticality levels. The authors proposed a schedulability test to find the appropriate priority ordering for mixed-criticality schedulability [3] and a response-time test [4] for this model, and they also considered sporadic tasks [4]. In contrast, our model includes only two execution times that correspond to whether a task has overloaded or not, and we focus on graceful degradation guarantee on overloads through the use of a zero-slack enforcement that stops lower-criticality tasks. The schedulability of mixed-criticality systems in multi-processors has also been studied before. Mollison et al. [23] presented a method for distributing tasks to processors based on the slack generated by the difference between the worst- and best-case execution times. Kelly et al. [18] presented a partitioned approach to scheduling mixed-criticality tasks in multiprocessors that follows Baruah's task model. Our vMCP packing algorithm is based on vector packing [7], [19], but considers the effect of mode changes on criticality, which the existing approaches did not. It is worth noting that none of the existing solutions have task models that consider modes, which is the focus of the present paper.

IX. Conclusions

We have presented a partitioned scheduling scheme for multi-modal mixed-criticality real-time systems on multiprocessor platforms. Our scheme consists of a scheduling algorithm for scheduling tasks on each processor and a bin-packing algorithm for assigning tasks to processors that take into account both mode and criticality changes. The scheduling algorithm extends the existing unimodal zero-slack rate monotonic algorithm [8] with a mode transition enforcement mechanism, which relies on transitional zero-slack instants to preserve the schedulability of high-criticality tasks and to ensure the absence of criticality violations during mode transitions. The packing algorithm uses the sustained criticality of tasks to optimize the schedulable utilization of the systems across modes. Our evaluation on synthetic tasksets shows that our packing algorithm can save up to half the processors and can preserve almost twice as much ductility as a baseline vector packing scheme. Finally, we presented an implementation of our scheduler in the Linux operating system, along with an experimental evaluation to illustrate its practicality.

Acknowledgments

This research was supported in part by the NSF grants CNS 1117185, ECCS 1135630 and CNS 1329984, the ARO grant W911NF1110403, the ONR grant N00014-13-1-0802, and the MKE (The Ministry of Knowledge Economy), Korea, under the Global Collaborative R&D program supervised by the KIAT (M002300089).

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
DM-0000674

References

[1] ARINC Standards. http://www.aviation-ia.com/standards/index.html.
[2] S. Baruah, V. Bonifaci, G. DAngelo, A. Marchetti-Spaccamela, S. Ster, and L. Stougie. Mixed-criticality scheduling of sporadic task systems. In *ESA*, 2011.
[3] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. *RTAS*, 2010.
[4] S. K. Baruah, A. Burns, and R. I. Davis. Response-time analysis for mixed criticality systems. In *RTSS*, 2011.
[5] G. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. *RTSS*, 1998.
[6] G. Buttazzo, M. Spuri, and F. Sensini. Value vs deadline scheduling in overload conditions. In *RTSS*, 1995.
[7] A. Caprara, H. Kellerer, and U. Pferschy. Approximation schemes for ordered vector packing problems. *Naval Research Logistics*, 50(1), 2002.
[8] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *RTSS*, 2009.
[9] D. de Niz, L. Wrage, A. Rowe, and R. R. Rajkumar. Utility-based resource overbooking for cyber-physical systems. In *RTCSA 2013*, 2013.
[10] N. Fisher and M. Ahmed. Tractable real-time schedulability analysis for mode changes under temporal isolation. In *ESTIMedia*, 2011.
[11] C. D. Gill, J. M. Gossett, D. Corman, J. P. Loyall, R. E. Schantz, M. Atighetchi, and D. C. Schmidt. Integrated adaptive qos management in middleware: A case study. *Real-Time Systems*, 2005.
[12] Q. Guangming. An earlier time for inserting and/or accelerating tasks. *Real-Time Systems*, 41(3):181–194, 2009.
[13] Y. Hang and H. Hansson. Timing analysis for mode switch in component-based multi-mode systems. In *ECRTS*, 2012.
[14] H.-M. Huan, C. Gill, and C. Lu. Implementation and Evaluation of Mixed-Criticality Scheduling Approaches for Periodic Tasks. In *RTAS*, 2012.
[15] ISO/DIS 26262 road vehicles - functional safety. http://www.iso.org.
[16] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.
[17] S. Kato, R. Rajkumar, and Y. Ishikawa. A loadable real-time scheduler suite for multicore platforms. Technical report, Department of Electrical and Comptuter Engineering, Carnegie Mellon University, 2009.
[18] O. Kelly, H. Aydin, and B. Zhao. On partitioned scheduling of fixed-priority mixed-criticality task sets. In *TrustCom*, 2011.
[19] K. Lakshmanan, D. de Niz, R. R. Rajkumar, and G. Moreno. Resource allocation in distributed mixed-criticality cyber-physical systems. In *ICDCS*, 2010.
[20] J. M. Lopez, J. L. Diaz, and D. F. Garcia. Utilization bounds for edf scheduling on real-time multiprocessor systems. *RTS*, 28, 2004.
[21] J. Marinho, G. Ravai, V. Nelis, and S. M. Petters. Partitioned scheduling of multimode systems on multiprocessor platforms: when to do the mode transition. In *RTSOPS*, 2011.
[22] P. Mejia-Alvarez, R. Melhem, and D. Mosse. An incremental approach to scheduling during overloads in real-time systems. In *RTSS*, 2000.
[23] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *CIT*, 2010.
[24] V. Nelis, J. Goossens, and B. Andersson. Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms. In *ECRTS*, 2009.
[25] V. Nelis, J. Marinho, B. Andersson, and S. M. Petters. Global-edf scheduling of multimode real-time systems considering mode independent tasks. In *ECRTS*, 2011.
[26] S. Oikawa and R. Rajkumar. Portable rk: a portable resource kernel for guaranteed and enforced timing behavior. In *RTAS 1999*, 1999.
[27] P. Pedro and A. Burns. Schedulability analysis for mode changes in flexible real-time systems. In *ECRTS*, 1998.

[28] L. T. X. Phan, S. Chakraborty, and I. Lee. Timing analysis of mixed time/event-triggered multi-mode systems. In *RTSS*, 2009.
[29] L. T. X. Phan, S. Chakraborty, and P. S. Thiagarajan. A multi-mode real-time calculus. In *RTSS*, 2008.
[30] L. T. X. Phan, I. Lee, and O. Sokolsky. Compositional analysis of multi-mode systems. In *ECRTS*, 2010.
[31] L. T. X. Phan, I. Lee, and O. Sokolsky. A formal semantic framework for mode change protocol. In *RTAS*, 2011.
[32] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26:161–197, 2004.
[33] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *RTS*, 1(3), 1989.
[34] C.-S. Shih, P. Ganti, and L. Sha. Schedulability and fairness for computation tasks in surveillance radar systems. In *RTAS*, 2004.
[35] N. Stoimenov, S. Perathoner, and L. Thiele. Reliable mode changes in real-time systems with fixed priority or edf scheduling. In *DATE*, 2009.
[36] K. Tindell, A. Burns, and A. Wellings. Mode changes in priority preemptively scheduled systems. In *RTSS*, 1992.

Appendix A: Preventing back-to-back preemptions

We observe that the back-to-back effect comes from a *soft* enforcement at the zero-slack instants: the enforcement postpones the execution of lower-criticality, higher-priority tasks τ_j, and it reactivates them after the zero-slack enforcement, which results in a shorter inter-arrival time between τ_j's jobs[10] and consequently more preemptions from τ_j. This issue can be prevented with the following modification: *At the zero-slack instant of τ_i, if τ_i has already executed for more than C_i, then all the active jobs of the tasks with lower-criticality than τ_i are dropped; otherwise, these jobs are only temporarily stopped, but they will be dropped later if τ_i executes beyond C_i.*

With the above modification, we can show that, if the portion of C_i that τ_i executes after its zero-slack instant (denoted by δ_i) is no more than $C_i^o - C_i$, then all the tasks with lower-criticality and higher-priority than τ_i (denoted by the set H_i^{lc}) will have enough cycles to complete before T_i elapses, despite any delay imposed on them.

Theorem IX.1. *Every task $\tau_j \in H_i^{lc}$ that is blocked by τ_i in τ_i's critical state finishes before τ_i's deadline if τ_i does not overload.*

Proof: First, note that δ_i is calculated as part of the calculation of the zero-slack instant Z_i^s of τ_i in the current mode s. Specifically, Z_i^s is calculated such that the processing time used by the higher-priority lower-criticality tasks (H_i^{lc}) during the interval $[Z_i^s, T_i]$ is enough to fit $\delta_i + (C_i^o - C_i)$ execution units. This processing time is the amount of time that τ_i steals from the taskset H_i^{lc} during τ_i's execution in its critical state, considering that all tasks $\tau_j \in H_i^{lc}$ execute for their overload execution times C_j^o.

Now, during the calculation of the zero-slack instant of each task $\tau_j \in H_i^{lc}$, δ_i is taken into account as a blocking term. In other words, the blocking term δ_i limits the processing time available to H_i^{lc}, and all tasks $\tau_j \in H_i^{lc}$ are able to tolerate this blocking and still finish executing for C_j^o time units by their deadlines. Furthermore, the total processing time available to H_i^{lc} cannot exceed the processing time available to τ_i in its critical mode after it has completed its nominal execution C_i (i.e., it is bounded by $C_i^o - C_i$), because it is the amount of time τ_i steals from the taskset H_i^{lc} during the calculation of Z_i^s. Thus, all pending executions of the tasks $\tau_j \in H_i^{lc}$ can be completed with the budget $C_i^o - C_i$, if τ_i does not overload and neither does any other task of higher-criticality than τ_j in mode s (otherwise τ_j is dropped).

Note further that the tasks $\tau_j \in H_i^{lc}$ are considered to be running for C_j^o and hence, if they do not overload, $C_j^o - C_j$ execution time units will be available for H_j^{lc}. If this property is applied recursively, the theorem follows. ■

By the same argument, we can also show that the theorem holds for transitional zero-slack instants. The absence of back-to-back preemptions directly follows.

[10]The distance between the restarting time of the postponed job of τ_j and the arrival time of the next job of τ_j.

Appendix B: Key functions in Algorithm 2

The function $\mathsf{GetSlackVector}(i, \Gamma_{i,n}^s, Z_i^s)$ (resp. $\mathsf{GetSlackVector}(i, \Gamma_{i,n}^t, Z_i^t)$) computes a sequence of slack intervals, each with a starting time and a duration, for τ_i from the simulated fixed-priority response-time timeline [16], taking into account the preemptions from the interfering taskset $\Gamma_{i,n}^s$ (resp. $\Gamma_{i,n}^t$). Specifically, the amount of interference caused by an interfering job $\tau_{j,k}$ that is taken into account (effective execution time C_j^E) to calculate the slack vectors for τ_i depends on the criticality and the priority of $\tau_{j,k}$ with respect to τ_i as follows: (1) If $\tau_{j,k}$ has higher priority but lower criticality, then its effective computation time is C_j^o when we calculate the slack in the nominal state, because $\tau_{j,k}$ can overload before the zero-slack instant of τ_i. (2) If $\tau_{j,k}$ has higher criticality, then its effective execution time is C_j when we calculate the slack in the nominal state, because τ_i is given a guarantee only if τ_j does not overload. (3) Finally, if $\tau_{j,k}$ has higher priority and higher criticality, then the effective execution time is $C_j - C_j^n$ when we calculate the slack in the critical state, because we do not need to consider the computation that already happens in the nominal state of $\tau_{j,k}$ (see [8] for more details).

The function $\mathsf{GetSlackStartings}(V_{i,n}^s, V_{i,n}^t)$ simulates two response-time timelines – with the source interfering taskset ($V_{i,n}^s$) and with the target interfering taskset ($V_{i,n}^t$) – and it returns all the starting instants of the slack intervals. Finally, $\mathsf{GetSlackInInterval}(s, e, V)$ returns all the slack available from the instant s to the instant e from the slack vector V.

Appendix C: Proof of Lemma IV.1

Proof of Lemma IV.1: We will show, for every job $\tau_{i,k}$ of any task τ_i in the system, that $\tau_{i,k}$ can never experience a criticality violation during its lifetime, based on the characteristics of the transitional zero-slack instants and zero-slack instants. Indeed, if $\tau_{i,k}$'s release time and deadline are outside the transitional interval of a mode transition, then the enforcement based on its zero-slack instant in the mode it is released always ensures that $\tau_{i,k}$ does not experience a criticality violation. If $\tau_{i,k}$'s lifetime spans across a mode transition (s, t), then (1) $\tau_{i,k}$ does not experience a criticality violation before the MCRI of (s, t), since the enforcement based on its zero-slack instant in mode s would stop all the lower-criticality tasks before they can make $\tau_{i,k}$ miss its deadline, and (2) $\tau_{i,k}$ does not experience a criticality violation after the MCRI, since the enforcement based on $\tau_{i,k}$'s transitional zero-slack instant would stop all lower-criticality jobs that are active after the MCRI before they can make $\tau_{i,k}$ miss its deadline. Similarly, we can also show that $\tau_{i,k}$ does not experience a criticality violation when it is released after the MCRI but within the transitional interval of a mode transition. Thus, $\tau_{i,k}$ never experiences a criticality violation. ■

We also observe that, since the transitional zero-slack instants are computed offline, the run-time enforcement involves only monitoring MCR arrivals and stopping jobs accordingly, which can be done efficiently.

Appendix D: Mode Transitions Run-time Overhead

To evaluate the scalability of our implementation, we conducted an empirical analysis of its run-time overhead. We focused on the overhead due to mode change and enforcement.
Experimental setup. Our experiments were performed on a Lenovo Yoga 13 laptop with 8 GB of RAM and an i7-3537U Intel processor with 4 MB of cache. The frequency of the processor was fixed to 2 GHz. Our kernel module scheduler was implemented and run in the Linux kernel 3.8.0-30 64 bits. For the mode change overhead measurements, we used a single task that (i) switched between two active modes (with period of 200 ms and 400 ms, respectively), or (ii) switched from an active mode (with a period of 400 ms) to an inactive mode, and vice versa. Finally, to measure the enforcement overhead, we ran a single task with a period of 200 ms and used the response-time timer to enforce it.
Measurement results. Table III presents the average and the worst-case values of the different types of overhead measured in our experiments. Each presented overhead values were computed from more than 1000 data points.

Action	Average (ns)	Worst Case (ns)
New job arrival without mode change	17,034	32,204
New job arrival with mode change	18,820	36,555
Transition into suspension	1,097	11,225
Resume from suspension	12,167	24,335
Mode change request	9,984	31,508
End-to-end mode change	11,770	35,859
Enforcement	8,281	21,735

TABLE III
Multi-modal mixed-criticality system run-time overhead

The first two lines of Table III give the extra overhead a task experiences in the presence of mode change. Specifically, the first shows the cost of a job arrival (job dispatching) when no mode change occurs (i.e., a regular job activation), whereas the second shows the overhead of the same action when a mode change happens. The mode change overhead is the difference of these two values: in our measurements, it takes 1,786 ns additional time on average to perform a mode change during a job dispatching and 4,351 ns in the worst case.

The third and fourth lines show the overhead of a task transitioning into and out of the suspension mode (the task is in the suspension mode if it is not active), respectively. It is worth noting that in the transition into the suspension mode, we avoid reprogramming the new timers for the next period; hence, a transition into the suspension mode is on average faster than both the job dispatching of an active task and a transition out of the suspension. In particular, during a new job activation (including when a task transitions out of the suspension mode), the timers for the new arrival and for the zero-slack instant, as well as the response-time enforcement timer, need to be programmed. In contrast, when a task goes into the suspension mode, we only need to check that the timers are inactive and do not need to reactive the thread (that was waiting for the arrival of the next period).

The fifth line shows the overhead of processing a mode change request, which was measured from the instant an MCR is initiated until the instant all the tasks receive the mode switch signals from the system mode manager. Note that, after receiving the mode switch signal, each mode-switching task executes its own signal handler with its own priority. This signal handling consists of only a couple of instructions to save the signal parameter with the number of the next mode into the `next_mode` variable; in our experiments, we avoided measuring this overhead, since the measurement itself would take much longer than the execution of the instructions.

The sixth line of the table gives the end-to-end mode change overhead. This is obtained by adding up the overhead of the mode-change request processing and the extra overhead of job activation due to mode change. While the two instructions executed in the signal handler are part of the end-to-end overhead, we did not include them (due to the reason mentioned above).

The final line in the table shows the overhead of the enforcement for the case when the enforced tasks resume in the next mode. In our implementation, a task is enforced (stopped) by changing its priority to a non-real-time priority, and this is done by the kernel scheduler task. Therefore, the enforcement overhead was measured as the time starting when the enforcement timer is triggered and ending when the scheduler thread has finished changing the priority of the task. Note that the worst-case enforcement overhead can include preemptions to the kernel thread by interrupts or other kernel threads, but these preemptions would also be experienced by the enforcing task anyway. For the case when the enforced tasks become inactive in the next mode, the enforcement is equivalent to the transition into suspension, except that it happens at the enforcement timer instead of at the period timer.

One final comment on the difference between the average and the worst-case overheads is in order. This difference is due to two effects: First, our kernel module takes advantage of high-resolution timers that program the timer chip as needed to be able to achieve nanosecond resolution. These timers use a tree data structure inside the kernel itself to decide when to program the timer chip and when not to. As a result, the resulting overhead is higher when the timer chip is programmed and lower when the timer chip was already previously programmed or when we do not setup or cancel any new timer[11]. These overhead values could be reduced by using periodic timers instead of using one-shot timers for the period. Due to timing constraints, we could not investigate this option; however, we plan to explore more efficient implementations based on periodic timers in our future work.

[11]Our implementation is an extension of the ZS-QRAM scheduler implementation, which uses these one shot-timers since the task periods change constantly.