Combining Task-level and System-level Scheduling Modes for Mixed Criticality Systems

Boudjadar, Jalil; Ramanathan, Saravanan; Easwaran, Arvind; Nyman, Ulrik

Published in:
Proceedings - 2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2019

DOI (link to publication from Publisher):
10.1109/DS-RT47707.2019.8958666

Publication date:
2020

Document Version
Accepted author manuscript, peer reviewed version

Citation for published version (APA):
Boudjadar, J., Ramanathan, S., Easwaran, A., & Nyman, U. (2020). Combining Task-level and System-level Scheduling Modes for Mixed Criticality Systems. In F. De Rango, C. T. Calafate, M. Voznak, A. Garro, & M. Tropea (Eds.), *Proceedings - 2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2019* (pp. 1-10). Article 8958666 IEEE (Institute of Electrical and Electronics Engineers). https://doi.org/10.1109/DS-RT47707.2019.8958666

Combining Task-level and System-level Scheduling Modes for Mixed Criticality Systems

Jalil Boudjadar[1] Saravanan Ramanathan[2] Arvind Easwaran[2] Ulrik Nyman[3]
[1] Aarhus University Denmark, [2] Nanyang Technological University, [3] Aalborg University Denmark

Abstract*—Different scheduling algorithms for mixed criticality systems have been recently proposed. The common denominator of these algorithms is to discard low critical tasks whenever high critical tasks are in lack of computation resources. This is achieved upon a switch of the scheduling mode from Normal to Critical. We distinguish two main categories of the algorithms: *system-level mode switch* and *task-level mode switch*. System-level mode algorithms allow low criticality (LC) tasks to execute *only* in normal mode. Task-level mode switch algorithms enable to switch the mode of an individual high criticality task (HC), from low (LO) to high (HI), to obtain priority over all LC tasks. This paper investigates an online scheduling algorithm for mixed-criticality systems that supports dynamic mode switches for both task level and system level. When a HC task job overruns its LC budget, then only that particular job is switched to HI mode. If the job cannot be accommodated, then the system switches to Critical mode. To accommodate for resource availability of the HC jobs, the LC tasks are degraded by stretching their periods until the Critical mode exhibiting job complete its execution. The stretching will be carried out until the resource availability is met. We have mechanized and implemented the proposed algorithm using Uppaal. To study the efficiency of our scheduling algorithm, we examine a case study and compare our results to the state of the art algorithms.

I. Introduction

Modern embedded systems are achieved via the integration of different system components having different criticality levels on a single platform. Such systems are known by *mixed criticality systems* (MCS). Examples are safety control systems in avionics [27] and automotive applications [1]. Mixed criticality systems are subjected to certifications dictated by the standards of different application areas, where different criticality levels require different assurance levels [2]. The consequences of missing a deadline vary in severity from task to task, according to the given criticality levels. It is therefore clear that highly critical components require a rigorous analysis to deliver a formal assurance about safety under error-free conditions, and the presence of certain defined errors maintains the behavior predictable [11].

During operation, it is important that critical tasks are supplied with sufficient computation resources to meet their time constraints. Running low critical tasks (**LC**) with the same privilege as high critical tasks (**HC**) enables the system functionality to be fully embraced [22], [44], however this leads to potential violation of the critical tasks safety e.g deadline miss. An intuitive alternative is to prioritize critical tasks eternally over low/non critical ones by the use of *criticality-as-priority*. Prioritizing critical tasks may require to discard low critical tasks. This may degrade the quality of service and functionality of the system [31], [28].

Since Vestal's seminal work [45], different scheduling algorithms for mixed criticality systems have been introduced [29], [42], [19], [4]. Such scheduling protocols rely on the assumption that a task can have different Worst Case Execution Time (WCET) bounds if one considers different confidence levels. This is due to the fact that determining the exact WCET of a task code is very pessimistic [12], [32]. A task's WCET can be bounded according to different confidence levels where the higher the confidence is the larger WCET will be [45].

Mixed criticality scheduling algorithms commonly use *scheduling modes* to decide which tasks to consider for scheduling at any point in time [10]. In essence, a scheduling mode dictates the tasks that can be prioritized/ignored according to the actual workload, so that tasks of a given criticality level obtain privilege over the rest of the tasks regardless of the actual priorities. Within a given scheduling mode, tasks are scheduled according to the adopted scheduling policy.

Scheduling algorithms for mixed criticality systems can be categorized, based on the type of mode switch scenario, in two groups: *system-level mode* and *task-level mode*. System-level mode scheduling algorithms [33], [29], [13] employ two scheduling modes **Normal** and **Critical**. HC and LC tasks are equally scheduled under Normal mode. A mode switch from Normal to Critical happens whenever there is a potential insufficiency of computation resources due to one or more HC tasks exhibiting high confidence behavior, i.e., tasks run for more than their low confidence WCET. In Critical scheduling mode, LC tasks are either entirely dropped [4], [16], or run with a degraded service [33], [42], [19] to accommodate HC tasks. The system-level algorithms commonly penalize LC tasks [33], [29], [13] as the system mode switch can be decided when a single HC task overruns its low confidence WCET.

Task-level mode switch [29], [26] is motivated by the fact that not necessarily all **HC** tasks exhibit high criticality behavior (largest WCET) at the same time. Thus, only the HC tasks running high confidence WCET obtain priority over the rest of tasks. Each HC task runs in **LO** mode and switches to **HI** mode whenever it overruns its low confidence WCET. Such overruns can lead to insufficiency of computation resources where HC tasks running LO mode miss their deadlines if their priorities are lower than those of LC tasks.

In this paper, we introduce a new elastic control-based scheduling algorithm by combining the aforementioned cat-

978-1-7281-2923-5/19/$31.00 ©2019 IEEE

egories. The resulting algorithm relies on a *job-level* mode switch technique, where the system mode switch occurs only when there is a **HC** task job, running **LO** mode, in risk to miss its deadline due to a low priority. We restrict HC behavior to only the job that either exceeds its low confidence WCET or triggers a systems mode switch. On Critical mode, we run LC tasks under a degraded mode (periods stretching) rather than completely discarded. When the workload permits, LC tasks are compensated by shrinking subsequent periods to amortize the degradation. Our scheduling algorithm enables runtime resilience and recovery from overload transient scenarios.

The rest of the paper is organized as follows: Section II cites the relevant related work. Section III presents our multimode scheduling setting for MCS. In Section. IV, we show how to analyze the schedulability. Section V is a case study. Finally, Section VI concludes the paper.

II. Related Work

Since Vestal's [45] seminal work on *mixed-criticality* (MC) systems, several studies have been carried out in the recent past for MC scheduling. Most existing works on MC scheduling [15], [4], [16], [13], [35] rely on system-level mode switch i.e., when a HC task executes more than its low confidence WCET the remaining HC tasks also simultaneously exhibit HC behavior. In order to guarantee resources for the HC tasks, many solutions employ a very pessimistic approach that completely discards all the LC tasks upon mode transition [15], [4], [16]. There are some works to delay the dropping of LC tasks by postponing the mode switch instant [38], [20], [23], [33]. Santy et al. [39] and Bate et al. [7] proposed some techniques to minimize the duration for which the system is in mode HI so that to reduce the non-service duration of LC tasks.

In this context, a plethora of studies has been carried out to improve the service offered to the LC tasks [3], [20], [28], [41], [40], [43], [5], [31], [36], [30], [18], [19], [25]. These approaches can be classified into four major categories:

1) *Elastic Scheduling*. The dispatch frequency of LC tasks is reduced (extending their periods) in the HI mode [3], [28], [41], [40], [43], [33].
2) *Imprecise Computation/Reduced Execution*. LC tasks are executed with reduced execution budget when the system is in mode HI [3], [5], [31], [20], [36], [30].
3) *Selective Degradation*. Depending on the budget availability in the HI mode, only a certain subset of LC jobs/tasks are executed [20], [18], [19].
4) *Processor speedup*. Huang et al. [24], [25], [9] proposed a dynamic processor speedup technique to guarantee resources for HC tasks instead of degrading the service to the LC tasks in the HI mode.

However, all the above works employ an impractical assumption that all the HC tasks in the system simultaneously exhibit HC behavior. On the contrary, there are very few works that relax the system-level mode switch assumption and employ task-level mode switch [26], [37], [21], [29]. Task-level mode switch algorithms restrict the impact of HC tasks exceeding their low confidence WCET and limit the service degradation of LC tasks.

Huang et al. [26] proposed a constraint graph to map the execution dependencies between HC tasks and LC tasks: when a HC task exhibits HC behavior only the LC tasks connected to it are dropped. However, in their analysis they consider all HC tasks utilize their high confidence WCET. Ren et al [37] proposed a similar technique in which each HC task is grouped with some LC tasks and only these tasks are affected if that particular HC task exhibits HC behavior.

Gu et al [21] presented a hierarchical component-based scheduling technique that allows multiple HC tasks to be grouped within a component. If any HC task in a component switches to HI mode, all the HC tasks in the component are run with their high confidence WCET and the LC tasks within that component are discarded. The authors also limit the number of components that can safely switch to HI mode using a tolerance parameter to trigger the system mode switch.

Erickson et al. [17] proposed a scheduling framework for multicore mixed criticality systems to recover from transient overload scenarios. The recovery relies on scaling the task inter-release times to reduce the jobs frequency. The underlying schedulability analysis requires that all tasks must run the WCETs of the same confidence level, which implies to rerun the analysis for each criticality level separately. Compared to that, our schedulability analysis is performed across different criticality levels at once.

Lee et al. [29] proposed an online schedulability test for task-level mode switch and an adaptive runtime task dropping strategy that minimizes LC task dropping. However, they consider all the jobs of a HC task exhibit HI mode behavior which may be a pessimistic assumption. Recently, Papadopoulos et al. [34] presented a control approach to achieve resilience in MC systems. HC tasks and LC tasks are executed using a server-based approach and based on the runtime property of the tasks the budget allocated to these servers is dynamically varied. When a HC server exhibits HC behavior, the LC servers are under-scheduled to meet the demand of HC servers. We rely on the same control-based mechanism to achieve LC task periods stretching, however we compensate such a degradation by shrinking LC task periods whenever the HC tasks workload permits.

In contrast to the above studies, we propose a dynamic mode switching algorithm that allows both task-level and system-level mode transitions. In particular, we restrict the HC behavior to only the job that either exceeds its low confidence WCET or triggers a systems mode switch. At the same time, we offer a minimum service to all LC tasks in the Critical mode using elastic scheduling instead of dropping them.

III. Multimode Scheduling of MCS

In this section, we combine system-level and task-level scheduling modes to produce a multimode scheduling algorithm for MCS. Our mixed criticality scheduling algorithm enables efficient mode switches for HC tasks, by predicting the workload causing HC tasks to fail.

A. System model

We consider deadline-implicit periodic task systems with two distinct criticality levels: high (HC) and low (LC), so that each mixed criticality (MC) task can be a LC or HC. By *default criticality*, we refer to the criticality level assigned to a given task at the design stage (constant). The *runtime criticality* of a task is in fact the (dynamic) criticality level assigned to the task according to the scheduling mode and/or task behavior.

a) Assumptions: We consider the following assumptions:

- Tasks are preemptible.
- All tasks are assigned a static criticality level (LC or HC) by design, called default criticality.
- The execution of a HC task must not be discarded under any runtime circumstances.
- The runtime criticality of a LC task can never be upgraded to HC.
- LC tasks stick always to their low confidence WCET.
- There is no dependency between LC and HC tasks.

b) Notations:

- We use π_i to refer to a single task, and Π to refer to the set of tasks.
- $Mode(t) \in \{Normal, Critical\}$ states the system scheduling mode at time point t.
- To track the mode of individual HC tasks over runtime, we introduce a function $\Omega : \{\pi_i \mid \chi_i = \mathbf{HC}\} \times \mathbb{R}_{\geq 0} \rightarrow \{\mathbf{HI}, \mathbf{LO}\}$. For the sake of notation, we write $\Omega(\pi_i, t)$ for the mode of task π_i at time point t.

Definition III.1 (Tasks)**.** *A task π_i is given by $\langle T_i, C_i^l, C_i^h, \chi_i, \rho\rangle$ where:*

- *T_i is the task period.*
- *$C_i^l \in \mathbb{R}_{\geq 0}$ and $C_i^h \in \mathbb{R}_{\geq 0}$ are the worst case execution time for low and high confidence levels respectively. We assume that $C_i^h \geq C_i^l$ for HC tasks, and $C_i^h = C_i^l$ for LC tasks.*
- *$\chi_i \in \{\mathbf{LC}, \mathbf{HC}\}$ is the default (constant) criticality of the task.*
- *ρ is the task priority.*

The task runtime mode $\Omega()$ will be updated on the fly according to the actual task execution budget.

We distinguish between the task mode $\Omega(\pi_i, t)$, which is individual for each task, and the system scheduling mode $Mode(t)$. A task scheduling mode is driven by its execution time, so that whenever the execution violates the low confidence WCET C_i^l the task mode is elevated to **HI**. The individual mode of a HC task switches independently. The overrun of C_i^l, by a HC task, is considered to be non-deterministic.

The system scheduling mode is common for all tasks. It determines the tasks that are allowed to execute, and the main scheduling criterion (criticality, priority or both). Under **Normal** mode, all ready tasks are equally scheduled according to the adopted scheduling policy. However, when the system mode is **Critical** criticality levels are used as the main scheduling criterion to arbitrate tasks. If two tasks have the same criticality level, then we refer to their actual priorities. In such a scheduling mode, **LC** tasks may not be scheduled given their low criticality level. A stretching of the LC task periods is applied while the system runs in mode Critical. Thus, reducing the utilization of LC tasks to accommodate HC tasks. Whenever the system scheduling mode returns to Normal, the periods of LC tasks are then shrunk to amortize the delays created by the stretching. The shrinking can start only after LC tasks complete the jobs of the periods experienced a stretching.

Taskset Π will be scheduled by the real-time operating system according to a scheduling function $Sched$. In fact, $Sched()$ implements an actual static priority-based scheduling policy such as Fixed Priority scheduling (FP).

$$Sched : 2^{\Pi} \times \mathbb{R}_{\geq 0} \rightarrow \Pi$$

In a similar way, we define a (*Intermediate*) scheduling function $Sched_I(\Pi, t)$ which employs both task mode and priority. Thus, a task gets scheduled at a given time point t if it has either a higher task mode[1] compared to any ready task, or the same task mode but a higher priority.

$$Sched_I(\Pi, t) = \pi_i \mid Ready(\pi_i, t) \wedge \forall \pi_j \in \Pi \; Ready(\pi_j, t) \Rightarrow \begin{cases} \Omega(\pi_j, t) < \Omega(\pi_i, t) \\ \vee \\ \Omega(\pi_j, t) = \Omega(\pi_i, t) \wedge Sched(\{\pi_i, \pi_j\}, t) = \pi_i \end{cases}$$

where $Ready(\pi_i, t)$ is a predicate stating whether a given task is ready at a given time point. As a third stage, we define a more restrictive scheduling function $Sched_C()$ which employs *Criticality* level, task mode and priority to decide which task to be scheduled at any point in time.

$$Sched_C(\Pi, t) = \pi_i \mid Ready(\pi_i, t) \wedge \forall \pi_j \in \Pi \; Ready(\pi_j, t) \Rightarrow \begin{cases} \chi_j < \chi_i \\ \vee \\ (\chi_j = \chi_i) \wedge \Omega(\pi_j, t) < \Omega(\pi_i, t) \\ \vee \\ (\chi_j = \chi_i) \wedge (\Omega(\pi_j, t) = \Omega(\pi_i, t)) \\ \qquad \wedge \; Sched(\{\pi_i, \pi_j\}, t) = \pi_i \end{cases}$$

The utilization of $Sched_I()$, $Sched_C()$ and $Sched()$ is described in the next sections. In the rest of this section, we present our task-level and system-level mode switches and how to combine both modes to achieve a more flexible scheduling.

B. Task-level mode switch

a) Low criticality tasks behavior: Low criticality tasks are not concerned by the task mode switch because they are not concerned by rigorous certification as high criticality tasks. They are also assumed to run always the same WCET, i.e. $C^l = C^h$. Figure 1 illustrates the LC tasks behavior. In fact, LC tasks execute regularly next to HC tasks as long as the system scheduling mode is Normal. Under that context LC tasks are equally scheduled, using $Sched()$, as HC tasks running in mode LO.

Upon a switch of the system mode to Critical, the current job periods of LC tasks are stretched to reduce their utilization and the frequency of releasing new jobs. The system is then declared to be performing a stretching pattern. We introduce a variable $\mathcal{P} \in \{Stretching, Shrinking, Regular\}$ to store the current system pattern.

[1] We consider that $HI > LO$, but HC tasks running in mode LO are comparable to LC tasks.

Figure 1: Low criticality task behavior

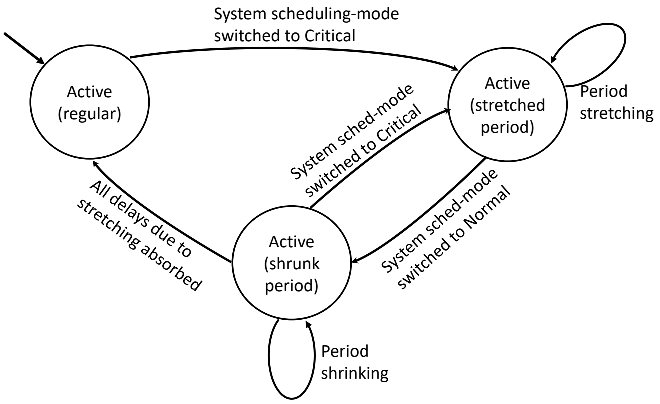
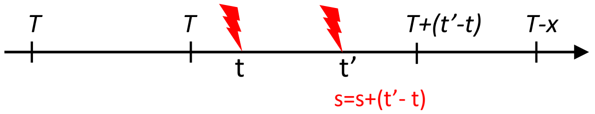


Figure 2: Stretching/shrinking of LC task periods



To track the stretching duration, we use a variable s which indicates how much an LC task needs to be compensated in order to absorb the delays caused by the stretching. The stretching of LC tasks is a degraded operation mode.

Whenever the system scheduling mode is back to Normal and the current stretched periods expire, the stretching is interrupted and the LC tasks can then execute regularly. To amortize the slack time created by stretching, the scheduler applies a shrinking to LC task periods [2]. The shrinking pace depends on the system workload and the LC task periods length. The fewer HC tasks run C^h the larger the shrinking will be. Once all the delays introduced due to stretching are amortized, LC tasks run regular periods[3].

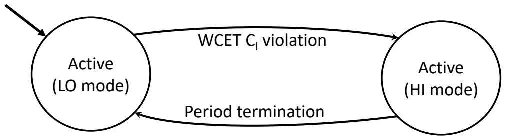
Figure 2 depicts an example of stretching and shrinking operations for an LC task period. Within the initial period, the task executes normally. After releasing the second period, a system mode switch (from Normal to Critical) happens at time t causing the period to be stretched until time instant t' where another system mode switch (Critical to Normal) occurs. The stretching duration $t' - t$ is accumulated in s. The third period will then be shrunk with $0 \leq x \leq s$ to absorb the delay s. If the delay s is not completely absorbed in one period, subsequent periods will be shortened accordingly. Formal calculation of the stretching/shrinking durations is provided in Section III-C

Given that C^l and C^h are equal for each LC task, we simply write C. The utilization of a LC task is defined as follows:

- Regular activation: $U_{L_i} = \frac{C_i}{T_i}$
- During shrinking with a duration δ: $U^{\delta}_{L_i} = \frac{C_i}{T_i - \delta}$ such that $C_i \leq (T_i - \delta)$.

b) High criticality tasks behavior: Each individual HC task starts at mode LO and can change its mode independently from the rest of tasks. By default, on the release of a new period the HC task runs LO mode and whenever C^l overrun happens the task mode switches to HI [29]. Such a task mode is maintained until the expiry of the given period. The budget overrun is *non-deterministic*. Figure 3 illustrates the mode switches of HC tasks.

Figure 3: High criticality task behavior



Whenever a HC task switches to mode HI, $\Omega(\pi_i, t) = \mathbf{HI}$, it obtains the scheduling privilege over all LC tasks. Besides, a HC task running in HI mode has priority over all HC tasks running in LO mode. Among the HC tasks running HI mode, the task having the highest priority is scheduled first. Function $Sched_I()$ is used to schedule tasks according to these criteria.

However, given that HC tasks running LO mode do not have privilege over LC tasks, a HC task can miss its deadline under LO mode in case there is a lack of computation resources to execute both HC and LC tasks. This can be considered to be the major drawback of both task-level and system-level scheduling algorithms of mixed criticality systems. To circumvent this issue, our scheduling algorithm can assign a HC task running in LO mode the privilege over LC tasks even though it does not overrun its low confidence WCET C^l.

We define the utilization of a HC task π_i running mode HI, respectively mode LO, by:

$$U_{H_i} = \frac{C_i^H}{T_i}, \text{ respectively } \; U_{L_i} = \frac{C_i^L}{T_i}$$

We also use U_L to refer to the utilization of LC tasks. To specify the task mode switches, we introduce the following functions:

- $Status(\pi_i, t) \in \{Ready, Running, Done\}$ returns the status of any task π_i at any point in time t.
- $\Lambda(\pi_i, t)$ returns the budget consumed at time t by the current release of a task π_i. $\Lambda(\pi_i, t)$ is not accumulative, i.e., it resets to zero upon each period release.

Formally, the runtime mode of a high criticality task switches from LO to HI as follows:

$$\boxed{\dfrac{\begin{array}{c}\forall\ \pi_i \in \Pi \mid \chi_i = HC,\ \forall t \mid \\ Status(\pi_i, t) \neq Done \wedge \Lambda(\pi_i, t) \geq C^l_i \wedge \Omega(\pi_i, t) = LO\end{array}}{\Omega(\pi_i, t) \mapsto HI}}$$

Accordingly, the runtime criticality of a HC task returns to LO mode whenever its period expires as shown below.

$$\boxed{\dfrac{\begin{array}{c}\forall\ \pi_i \in \Pi \mid \chi_i = HC,\ \forall t \mid \Omega(\pi_i, t) = HI \ \wedge \\ Status(\pi_i, t) = Done \wedge t \ \% \ T_i = 0\end{array}}{\Omega(\pi_i, t) \mapsto LO}}$$

$\%$ is the arithmetic modulo operator. One can see that the task-level mode switch relies on the violation of C^l and does not guarantee the feasibility of HC tasks running LO mode.

[2] The system pattern is then updated accordingly, $\mathcal{P} = Shrinking$.

[3] $\mathcal{P} = Regular$.

Figure 4: Runtime example for the system in Table. I

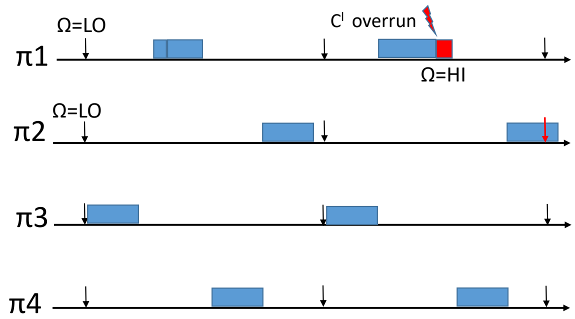
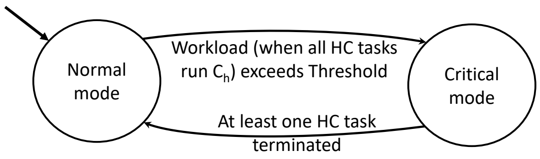


Figure 5: System scheduling mode behavior


Normal mode
Workload (when all HC tasks run C_h) exceeds Threshold
Critical mode
At least one HC task terminated

C. System-level mode switch

As stated earlier, the task level mode can be used to prioritize HC tasks running in HI mode. The drawback of the task level scheduling mode is then *how to prioritize a HC task running a LO mode when the system workload lacks computation resources*. To circumvent this drawback, our system level mode complements the task level mode and constrains the classic system level mode switches with the workload of HC tasks running both **LO** and **HI** modes equally. Let us illustrate the aforementioned drawback scenario for the system of Table I.

Table I: Example of a failure case for both system and task level scheduling modes

Task	T	C^l	C^h	χ	ρ
π_1	20	5	7	HC	2
π_2	20	5	6	HC	4
π_3	20	5	-	LC	1
π_4	20	4	-	LC	3

Figure 4 depicts a runtime example. On the first period, tasks execute according to the order of their priorities. On the second period, π_1 violates its $C^l = 5$ and runs for two extra time units. This delays π_4, which in turn delays π_2 due to its lower priority. In the end, π_2 misses its deadline with one time unit. This scenario could be avoided if one would account for the feasibility of π_2, at the time point when π_1 violates C^l, and elevate its priority immediately. Thus, π_2 would execute before π_4 and meets its deadline.

To summarize, our system level scheduling mode monitors the workload, for both LC and HC tasks, online and decides when to prioritize HC tasks over all LC tasks regardless of the HI/LO task modes. The system scheduling mode is effectively switched from Normal to Critical if the actual workload of LC tasks and HC tasks exceeds the resource supply for a time interval starting at the actual time point.

Figure 5 shows the system mode behavior. The system is initially at Normal mode, and transits to Critical mode when the resource demand exceeds the resource supply. LC task periods are stretched accordingly, thus reducing their utilization, to make room in the schedule for HC tasks at least for their low confidence WCET C^l. Whenever the workload of HC tasks is relaxed, the system switches back to Normal and LC tasks can then be compensated to absorb the delay caused by stretching.

We define the workload function $\Psi(\pi_i, [a, b])$ of a task π_i over a time interval $[a, b]$ to be the amount of resource that can be requested by π_i. Such a workload includes the remaining execution time at time point a for the current job plus the jobs to be potentially released until time instant b. We distinguish between $\Psi^H()$ and $\Psi^L()$ according to the task criticality and modes.

$$\Psi^H(\pi_i, [a, b]) = \begin{cases} C_i^h - \Lambda(\pi_i, a) + U_{H_i} \cdot T_i \cdot \lceil \frac{b-a}{T_i} \rceil & \text{If} \\ & (b-a)\% T_i \geq C_i^h \\ C_i^h - \Lambda(\pi_i, a) + U_{H_i} \cdot T_i \cdot \lfloor \frac{b-a}{T_i} \rfloor & \text{Otherwise} \end{cases}$$

$$\Psi^L(\pi_i, [a, b]) = \begin{cases} C_i^l - \Lambda(\pi_i, a) + U_{L_i} \cdot T_i \cdot \lceil \frac{b-a}{T_i} \rceil & \text{If} \\ & (b-a)\% T_i \geq C_i^l \\ C_i^l - \Lambda(\pi_i, a) + U_{L_i} \cdot T_i \cdot \lfloor \frac{b-a}{T_i} \rfloor & \text{Otherwise} \end{cases}$$

We define the workload of HC tasks having a high criticality than π_i for the time interval $[t, T_i]$ as follows:

$$W_H^h(\pi_i, t) = \sum_{\pi_j | \chi_i = HC \wedge \Omega(\pi_j, t) = HI} \Psi^H(\pi_j, [t, T_i])$$

Implicitly, the time interval $[t, T_i]$ is the duration left to the expiry of the last period released by task π_i before time point t, i.e. $[t \;\%\; T_i, T_i]$. Thus, we avoid writing the conversion absolute-relative time. In a similar way, we calculate the workload of HC tasks running LO mode and having higher priority than π_i, for time interval $[t, T_i]$ as follows:

$$W_L(\pi_i, t) = \sum_{\pi_j | \chi_j = LC \wedge \pi_j \in hp(\pi_i, t)} \Psi^L(\pi_j, [t, T_i])$$

where $hp(\pi_i, t)$ is the set of tasks having a higher priority than π_i at time point t. Finally, the workload of LC tasks having a higher priority than π_i is given by:

$$W_H^l(\pi_i, t) = \sum_{\pi_j | \chi_j = HC \wedge \pi_j \in hp(\pi_i, t) \wedge \Omega(\pi_j, t) = LO} \Psi^L(\pi_j, [t, T_i])$$

We define $\mathtt{DEM}(\pi_i, t)$, an upper bound on the resource demand over a given time interval [6], of a HC task running in LO mode at any time point t till the expiry of that period to be the remaining budget of such a task for the given period plus the workload of tasks having either a higher criticality or a higher priority. Namely, these are LC tasks having a higher priority, HC tasks running HI mode and HC tasks running LO mode but having higher priority than task π_i.

$$\mathtt{DEM}(\pi_i, t) = W_H^h(\pi_i, t) + W_H^l(\pi_i, t) + W_L(\pi_i, t) + C_i^L - \Lambda(\pi_i, t)$$

One can see that we distinguish between HC tasks running HI, and HC tasks running LO and having higher priority than a given task. This is in fact to avoid counting the tasks satisfying both conditions twice in the workload. Given that the maximum resource amount that can be supplied to the task set during a time interval $[a, b]$ is $b - a$, the system scheduling mode switches from Normal to Critical if the workload exceeds (or is going to exceed) the resource supply.

$$\boxed{\dfrac{\begin{array}{c}\exists t\ \pi_i \mid \chi_i = HC \ \wedge \\ \Omega(\pi_i, t) = LO \ \wedge \ \mathtt{DEM}(\pi_i, t) \geq T_i - (t \ \% \ T_i)\end{array}}{Mode(t) \mapsto \textbf{Critical}}}$$

One can see that the load calculation, as a ground for the system mode switch, is performed on the time interval of the actual trigger task rather than classic entire busy period. This is in fact to reduce the over-approximation of the workload, given that low confidence WCET violation is non-deterministic, and deliver an exact load calculation.

Once the system scheduling mode is switched to Critical, the periods of LC jobs will be extended with the time left of the current release $(T_i - (t \ \% \ T_i))$ of the HC task (π_i) causing the mode switch.

Let us call the HC task causing the actual system mode switch a *trigger* $\mathcal{T}$, and $\mathcal{S}$ the relative time instant of the corresponding mode switch [4]. Thus, we simply write $\mathcal{T}(\pi_i, \mathcal{S})$ for a task π_i being a trigger at time $\mathcal{S}$. In Critical mode, the system uses $Sched_C()$ to schedule tasks rather than $Sched()$ so that LC tasks do not have a chance to execute before any HC task regardless of the HC task mode and priority. This does not mean that LC tasks are discarded but rather they can execute once HC tasks are satisfied.

We define the demand bound function of a trigger task π_i to be the workload of that task (running C^h) plus the workload of HC tasks running HI mode and having higher priority than π_i.

$$\mathtt{DEM}^c(\pi_i, \mathcal{S}) = \sum_{\substack{\pi_j \mid \chi_j = HC \\ \wedge\ \Omega(\pi_j, \mathcal{S}) = HI \\ \wedge\ \pi_j \in hp(\pi_i, \mathcal{S})}} \Psi^H(\pi_j, [\mathcal{S}, T_i]) + (C_i^h - \Lambda(\pi_i, \mathcal{S}))$$

To make room for the trigger task to fully execute just in case it violates its low confidence WCET, we consider C_i^h instead of C_i^l in $\mathtt{DEM}^c()$ calculation. This can be an over-approximation but it is much safer and practical given that HC tasks non-deterministically run C_i^h. In case the trigger task sticks to its allotted execution time C_i^l, the surplus time is used to accommodate more LC tasks. The mode trigger task π_i is schedulable (under the stretching pattern) if:

$$\mathtt{DEM}^c(\pi_i, \mathcal{S}) \leq T_i - \mathcal{S}$$

Whenever the current job of the trigger task expires [5], the system scheduling mode switches from **Critical** to **Normal**. The mode change instant is calculated from $\mathcal{S}$ with the time left to the period expiry of π_i, i.e. $t' = \mathcal{S} + (T_i - \mathcal{S})$.

$$\boxed{\dfrac{\exists \pi_i \mid \mathcal{T}(\pi_i, \mathcal{S}) \wedge Mode(\mathcal{S} + (T_i - \mathcal{S})) = \textbf{Critical}}{Mode(\mathcal{S} + (T_i - \mathcal{S})) \mapsto \textbf{Normal}}}$$

Upon such a mode switch, the trigger task is refreshed for the new period where $\Omega(\mathcal{T}, t')$ is set to LO and $\Lambda(\mathcal{T}, t')$ to 0. To such a purpose, we define the following function:

$$\mathtt{Refresh}(\pi_i, t) = (\Omega(\pi_i, t) \mapsto LO) \wedge (\Lambda(\pi_i, t) \mapsto 0)$$

where π_i must be the most recent trigger task [6] and t is the mode switch-back instant $(\mathcal{S} + (T_i - \mathcal{S}))$.

*a) Stretching of **LC** task periods:* To guarantee the runtime resilience, our control-based scheduling algorithm stretches the current job periods of the LC tasks with the duration $(T_i - (t \ \% \ T_i))$, left to the expiry of the current release of the trigger HC task (π_i), when system mode switches to Critical (at time t). Once the system mode is switched back to Normal, one needs to absorb the stretching delay $(T_i - \mathcal{S})$ of LC tasks so that such tasks return to regular periodic dispatch.

*b) Shrinking of **LC** task periods:* The shrinking rate of the LC task periods depends on the actual system workload and the length of the individual LC task periods. In fact, the shrinking is driven by the schedulability of the HC task running in LO mode and having the lowest priority, i.e. a priority lower than LC tasks. We consider the current job of such a HC task, and calculate first how would be the schedulability of that task according to the workload resulting from the shrinking of LC periods with a duration δ. We start with δ equals to the stretching duration $(T_i - \mathcal{S})$, if the resulting workload is schedulable (using a DEM-based online schedulability test) then the shrinking is applied. Otherwise, we consider a tighter shrinking duration $\delta < T_i - \mathcal{S}$ and so on until the workload is schedulable. This binary process can end up having $\delta = 0$ if the resulting workload is not schedulable for any potential shrinking duration.

Let us assume a shrinking duration $\delta \leq T_i - \mathcal{S}$ (the stretching duration due to the most recent trigger task). Let us assume also that η is the instant of the system mode switch back to Normal mode. The shrinking with δ will be split over a number of periods each LC task can perform within the time left $(T_i - \eta)$ to the expiry of the current job of the HC task running LO mode with lowest priority (π_i). The number of LC task (π_j) periods occurring within $[\eta, T_i]$, after shrinking with δ, is given by $\frac{T_i - \eta + \delta}{T_j}$. Then the actual shrinking of each LC task (T_j) period is μ such that $\delta = \mu \cdot \frac{T_i - n}{T_j - \mu}$ which makes $\mu = \frac{T_j \cdot \delta}{T_i - \eta + \delta}$ [7].

We calculate first the resource demand $\mathtt{DEM}^\delta(\pi_i, n)$ of the HC task, running LO and having the lowest priority level, assuming the actual shrinking μ of LC task periods, from the mode change instant until the expiry of its current job period.

$$\begin{aligned}\mathtt{DEM}^\delta(\pi_i, \eta) = \ & W_H^h(\pi_i, \eta) + W_H^l(\pi_i, \eta) + W_L^\delta(\pi_i, \eta) + \\ & (C_i^l - \Lambda(\pi_i, \eta))\end{aligned}$$

The workload of LC tasks after shrinking is given as follows:

$$W_L^\delta(\pi_i, \eta) = \sum_{\pi_j \mid \chi_j = LC \wedge \pi_j \in hp(\pi_i, \eta)} U_{L_j}^\mu \cdot (T_j - \mu) \cdot \lceil \frac{T_i - \eta}{T_j - \mu} \rceil$$

Figure 6 depicts the period shrinking of two LC tasks for a total duration $\delta = 12$. We omitted HC tasks and only the lowest priority HC task is depicted. The periods of π_2, released

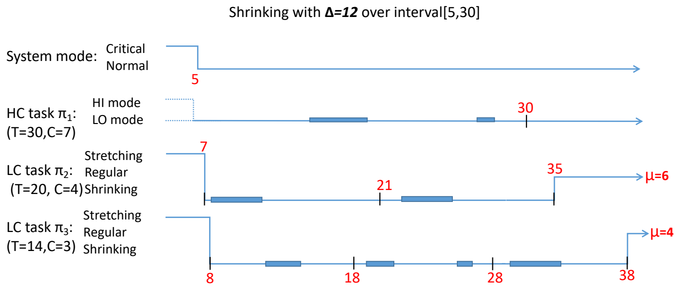
[4] For the sake of notation, we consider $\mathcal{S}$ to be a time instant relative to the current release of the trigger task so that we avoid the conversion relative-absolute time.

[5] The period of the most recent $\mathcal{S}$.

[6] $\mathcal{T}(\pi_i, \mathcal{S})$ and $\forall t \in [\mathcal{S}, T_i] \ \forall \pi_j \neq \pi_i \ \neg \mathcal{T}(\pi_j, t)$.

[7] μ is the actual shrinking of each period of a given LC task π_j whereas δ is the accumulated shrinking over $[\eta, T_i]$.

Figure 6: Example of LC task periods shrinking



within interval [5,30], are shrunk with $\mu = 6$ whereas the periods of π_3 are shrunk with $\mu = 4$. Given that we have two periods of π_2, respectively three for π_3, within [5,30] thus the accumulated shrinking $2 \times 6 = 12$, respectively $3 \times 4 = 12$, equals δ.

D. Multimode Scheduling Algorithm

Our scheduling algorithm is a control-based where the scheduling parameters and criteria (priority only, priority and criticality, priority-criticality-mode) considered to arbitrate tasks depend on the actual system workload and task modes. The overall scheduling algorithm is depicted in Algorithm 1 where t is a clock variable to model the time progress. We introduce a function $Use()$ to dictate the scheduling criteria to be used during runtime, in terms of priority, default criticality and/or runtime criticality. The corresponding scheduling function $(Sched(), Sched_I()$ or $Sched_C())$ is then accordingly applied.

Let us introduce $lp_l(t) = \pi_i \mid \ \chi_i = HC \ \wedge \ \Omega(\pi_i, t) = LO \ \wedge \forall \pi_j \ Sched_I(\pi_i, \pi_j, t) \neq \pi_i$ to be the lowest priority HC task running LO mode. Similarly, we use $lp_h(t) = \pi_i \mid \ \chi_i = HC \ \wedge \ \Omega(\pi_i, t) = HI \ \wedge \forall \pi_j \ Sched_c(\pi_i, \pi_j, t) \neq \pi_i$ to refer to the lowest priority HC task running HI mode. Whenever the execution period of a HC task expires, we refresh the task mode accordingly to be LO.

The initialization function is given by:

$$Init() = \begin{cases} t = 0 & \wedge \\ Mode(t) = Normal & \wedge \\ \mathcal{P} = Regular & \wedge \\ \forall i \mid \chi_i = HC \ Refresh(\pi_i, t) & \wedge \\ Use(Sched()) & \end{cases}$$

The statement in line 3 describes when to refresh both status and mode of each HC task upon the release of a new period. The task mode switch from LO to HI is given in lines 6-8. Lines 10-18 describe a system mode switch from Normal to Critical where a shrinking operation is applied. Lines 21-29 describe the system mode switch back to Normal whenever the current period of the most recent trigger task expires. Lines 32-38 outline when a shrinking operation for the LC task periods is released.

Upon each mode switch, a refreshment of some of the tasks is performed, if needed. Moreover, the scheduling function to be employed is specified using function $Use()$

In principle, a shrinking is applied as long as the stretching duration δ is not completely amortized. To simplify the

Algorithm 1: Elastic multimode scheduling

```
1  $Init()$;
2  while $True$ do
3      if $\exists \pi_i \mid Status(\pi_i, t) = Done \wedge t\%T_i = 0)$ then
4          $Refresh(\pi_i)$;
5      end
6      if $\exists \pi_i \mid \chi_i = HC \wedge \Lambda(\pi_i, t) \geq$
         $C_i^l \ \wedge Status(\pi_i, t) \neq Done$ then
7          $\Omega(\pi_i, t) = HI$;
8          $Use(Sched^I())$;
9      end
10     if $Mode(t) = Normal \wedge \mathtt{DEM}(lp_l(t), t) <$
         $lp_l(t).T - t\%lp_l(t).T)$ then
11         $\mathcal{T} = lp_l(t)$;
12         $\mathcal{S} = t$;
13         $Mode(t) = Critical$;
14         $\mathcal{P} = Stretching$;
15         $Use(Sched^c())$;
16         foreach $\pi_j \mid \chi_j = LC$ do
17             $T_j \mapsto T_j + (lp_l(t).T - t\%lp_l(t).T)$;
18             $\delta = \delta + (\mathcal{T}.T - \mathcal{S})$;
19         end
20     end
21     if
         $Mode(t) = Critical \wedge \exists \pi_i \mid \mathcal{T}(\pi_i, \mathcal{S}) \wedge t\%T_i = 0$
         then
22         $Mode(t) = Normal$;
23         $\mathcal{P} = Regular$;
24         $\eta = t$;
25         if $\exists \pi_j \mid \Omega(\pi_j, t) = HI$ then
26             $Use(Sched_I())$;
27         end
28         else
29             $Use(Sched())$;
30         end
31     end
32     if $Mode(t) = Normal \ \wedge \ \delta > 0$ then
33         if $DEM^{\delta}(lp_l(t), t) \leq lp_l(t).T - t$ then
34             foreach $\pi_j \mid \chi_j = LC$ do
35                 $T_j = T_j - \mu_j$;
36             end
37             $\mathcal{P} = Shrinking$;
38             $\delta = 0$;
39         end
40     end
41 end
```

algorithm, we have specified a one-go shrinking action, but the shrinking might be performed on several chunks due to preemption of the system Normal mode. This can be achieved using an extra variable to track the accumulated stretching delays.

IV. Schedulability Analysis

In this section we show how to analyze the schedulability of MCS running our new scheduling algorithm. Our schedulability analysis is in fact an online test checking the actual workload of the different modes and compare it against the resource supply that can be provided for each mode during a given time interval. We consider the mode switch instants to be the ground to calculate both demand and supply bound functions for our online schedulability test. This makes our schedulability test applicable no matter of how many mode switches happen during the system execution.

The ultimate goal of our algorithm and the underlying schedulability analysis is:

- guarantee the feasibility of HC tasks under all potential modes and patterns, i.e. $\forall t\ \pi_i \mid \chi_i = HC, t\ \%\ T_i = 0 \Rightarrow Status(\pi_i, t) = Done$.
- minimize the degradation of LC tasks, and compensate for all potential degradation.

To perform the schedulability test, we define the demand bound function $\texttt{DBF}\ (\pi_i, [t, t+z])$ to be the resource demand $\texttt{DEM}(\pi_i, t)$ of a HC task π_i for the entire busy period z starting at time instant t. We simply write:

$$\texttt{DBF}(\pi_i, [t, t+z]) = \texttt{DEM}(\pi_i, t | \Psi(\pi_i, [t, T_i \mapsto t+z]))$$

$\texttt{DBF}^c(\pi_i, [t, t+z])$ and $\texttt{DBF}^\delta(\pi_i, [t, t+z])$ are accordingly built on $\texttt{DEM}^c(\pi_i, t)$ and $\texttt{DEM}^\delta(\pi_i, t)$ respectively. t is the time instant of the Normal mode release, which could be either "0" for the initial system release or a time instant where the system mode switches back to Normal.

A given system remains under Normal mode as long as all HC tasks are schedulable, $\texttt{DBF}()$ of the lowest priority HC task π_i does not exceed the potential resource supply for the time interval $[t, T_i]$. To check schedulability, regardless of the individual task modes, we analyze $\texttt{DBF}()$ of the lowest priority HC task.

Theorem IV.1 (Schedulability under Normal mode)**.** *The HC taskset is schedulable when the system runs in mode **Normal**, with at least one **HC** task under mode **LO**, if the following holds:*

$$\forall t\ Mode(t) = Normal\ \forall \pi_i \mid \Omega(\pi_i, t) = LO\ \wedge\ lp_l(t) = \pi_i \\ and\ \textit{DBF}(\pi_i, [t, t+z]) \leq z$$

Proof. It is trivial. Given that π_i is the least priority ($lp_l(t)$) HC task ($\Omega(\pi_i, t) = LO$), then $\forall \pi_j \neq \pi_i\ \pi_j \in hp(\pi_i, t)$. Since we only consider fixed priority policies, thus $lp_l(t) = \pi_i \Rightarrow lp_l(t' \in [t, t+z]) = \pi_i$, i.e π_i remains the lowest priority HC task over [t,t+z]. From $\texttt{DBF}(\pi_i, [t, t+z])$ definition $W_H^h(\pi_i, t)$ and $W_H^l(\pi_i, t)$ [8] include the workload of each newly released HC job in the time interval [t,t+z] having either a higher priority ($\pi_j \in hp(\pi_i, t) \wedge \Omega(\pi_j, t) = LO$) or a higher task mode ($\Omega(\pi_j, t) = HI$), and the execution budget left for the actual period of time instant t ($C_i^L - \Lambda(\pi_i, t)$). Thus, if π_i is schedulable then $\forall \pi_j \mid Sched(i, j, t' \in [t, t+z]) = \pi_j \wedge \Omega(\pi_j) \geq \Omega(\pi_i)$ is schedulable. □

This Theorem implies that, in case the lowest priority task is a high critical, the schedulability test includes all HC and LC tasks. Thus, the schedulability of HC tasks implies the schedulability of the entire task set.

In case the system is in Normal mode but all HC tasks run mode HI, there is no point to consider LC tasks as any HC task has priority over all LC tasks.

Theorem IV.2 (Schedulability when all **HC** tasks run **HI** mode)**.** *The **HC** taskset is schedulable when the system runs in mode **Normal**, with all **HC** tasks under mode **HI**, if the following holds:*

$$\forall t\ Mode(t) = Normal\ \wedge\ \forall \pi_j\ \Omega(\pi_j, t) = HI \\ and\ \textit{DBF}^c(lp_h(t), [t, t+z]) \leq z$$

Proof. It is trivial. □

In a similar way, the schedulability of the HC taskset under shrinking pattern is defined by the schedulability of the lowest priority HC task running LO mode. This is because such a task is comparable to LC tasks, thus it can be affected by the shrinking workload.

Theorem IV.3 (Schedulability under Shrinking pattern)**.** ***HC** taskset is schedulable when the system runs a shrinking with a delay δ if:*

$$\forall t\ Mode(t) = Normal\ \wedge\ \mathcal{P} = Shrinking \Rightarrow \\ \textit{DBF}^\delta(lp_l(t), [t, t+z]) \leq z$$

Proof. It is similar to that of Theorem. IV.1. □

Again, this theorem implies not only the schedulability of HC tasks but the schedulability of the entire task set in case the lowest priority task of Π is a HC task.

Whenever a HC task, running in mode LO, is jeopardized to miss its deadline under mode Normal our scheduling algorithm anticipates a system mode change to **Critical**. Thus, HC taskset is schedulable under Critical mode if the lowest priority HC task running in mode LO, known as a trigger task, is schedulable.

Theorem IV.4 (Schedulability under critical mode)**.** ***HC** taskset is schedulable when the system runs **Critical** mode if:*

$$\forall t\ Mode(t) = Critical,\ \exists \pi_i \mid \Omega(\pi_i, t) = LO\ \wedge \\ \forall \pi_j \mid \chi_j = HC,\ Sched_C(\pi_i, \pi_j, t) \neq \pi_i \Rightarrow \\ \textit{DBF}^c(\pi_i, [t, t+z]) \leq z$$

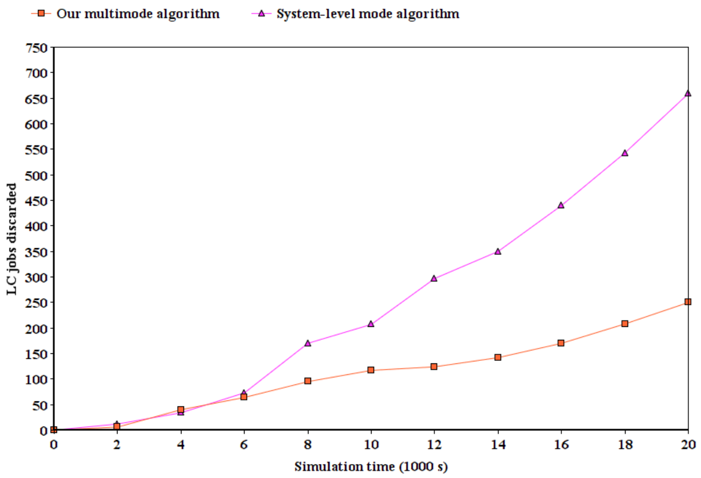
Proof. The condition $\forall \pi_j \mid \chi_j = HC\ Sched_C(\pi_i, \pi_j, t) \neq \pi_i$ implies that π_i is either the lowest priority HC task or the HC task having the lowest task mode ($\Omega(\pi_i, t) = LO$) given that $Sched_C()$ relies on both task runtime mode and priority. By definition $\texttt{DBF}^c(\pi_i, [t, t+z])$, includes the workload of all HC tasks $\pi_j \mid \chi_j = HC \wedge\ \Omega(\pi_j, t) = HI\ \wedge\ \pi_j \in hp(\pi_i, t)$. Thus, if π_i is schedulable then any other HC task will be schedulable. □

[8] With $\Psi^H(\pi_i, [t, z])$ and $\Psi^L(\pi_i, [t, z])$ calculated for the entire busy period.

Table II: Task attributes of the case study

Task	χ	T	C^l	C^h	ρ
Aircraft flight data(π_1)	HC	55	8	8.9	6
Steering(π_2)	HC	80	6	6.3	9
Target tracking(π_3)	HC	40	4	4.2	3
Target sweetening(π_4)	HC	40	2	2	4
AUTO/CCIP toggle(π_5)	HC	200	1	1	12
Weapon trajectory(π_6)	HC	100	7	7.5	10
Reinitiate trajectory(π_7)	LC	400	6.5	-	14
Weapon release(π_8)	HC	10	1	1.2	1
HUD display(π_9)	LC	52	6	-	7
MPD tactical display(π_{10})	LC	52	8	-	8
Radar tracking(π_{11})	HC	40	2	2.2	2
HOTAS bomb button (π_{12})	LC	40	1	-	5
Threat response display(π_{13})	LC	100	3	-	11
Poll RWR(π_{14})	LC	200	2	-	13
Perodic BIT(π_{15})	LC	1000	5	-	15

Figure 7: Comparison of the LC task jobs discarded



V. Case Study

To study the applicability and performance of our multimode scheduling algorithm and show the underlying schedulability analysis, we have analyzed an actual example from the avionic domain [14]. The most relevant attributes of the task set description are given in Table II.

We have synthetically calculated C^h from C^l by considering the worst case response time of data fetching. The original taskset description of [14] states how many data each task exchanges during each period. The best case response time of data fetching is instantaneous whereas the worst case response time is $20\mu s$ for data words, $40\mu s$ for a command and $40\mu s$ for a status. The scheduling policy adopted to schedule the task set is FP (fixed priority).

To analyze the case study, we have mechanized the system model and scheduling algorithms in Uppaal [8]. When we run the taskset using a classic priority-based scheduling, tasks π_{10} and π_{11} miss their deadlines making thus the system not schedulable. When the system runs fixed priority policy with *task level* scheduling mode only, task π_{10} *misses its deadline* (response time 106).

When the taskset runs the *system-level* scheduling mode, all HC tasks meet their deadlines whereas multiple LC jobs are discarded to achieve the schedulability of HC tasks. The number of LC task jobs discarded is depicted in Fig. 7.

When the system runs our *multimode scheduling algorithm*, all the high criticality tasks meet their deadlines. To achieve the schedulability of the HC tasks, our scheduling algorithm postpones the execution of some of the LC tasks. We consider each postponing operation with a delay longer than the corresponding LC task slack time to be a discard case. This is because a delay longer than the available slack time will absolutely lead the task execution to miss its deadline. The number of LC task jobs discarded by our algorithm is depicted in Figure 7.

Compared to the state of the art, for the given case study, our multimode scheduling algorithm guarantees the schedulability of all HC tasks whereas Task-level scheduling algorithms do not. Moreover, the discard rate of the LC task jobs achieved by our algorithm is 1.0% to 4.58% whereas the discard rate achieved by the state of the art system-level bi-mode scheduling [13], [33] is 2.1% to 11.5%. The discard rate is calculated to be the number of jobs discarded to the total number of jobs released.

An important observation from this experiment is that, although the proposed algorithm achieves less discards to low criticality tasks, it requires around 30% extra overhead compared to most of the state of the art algorithms. By overhead we mean the data size to track the system runtime and the time to process such data. Thus, the combination of task-level and system-level mode switches is *not efficient* in making real-time scheduling decisions. Another observation is that the compensation of LC tasks is slow given that LC tasks have the period lengths comparable to the period of the lowest priority HC task.

VI. Conclusion

This paper introduced a flexible multimode scheduling algorithm for mixed criticality systems by combining the system-level and task-level mode switch techniques. The proposed algorithm relies on a job-level mode switch, where we restrict the HC task behavior to only the job that either exceeds its low confidence WCET or triggers a system mode switch. This technique provides an exact schedulability test for the system mode switches. Low criticality tasks are not discarded under critical mode, rather their periods are stretched to loosen the underlying workload. Such tasks are later compensated for the degradation, due to stretching, by shrinking their subsequent periods accordingly. We have mechanized our new multimode scheduling algorithm in Uppaal and analyzed an actual avionic system component as a case study.

The efficiency of our elastic algorithm remains in the fact that considering a short range load calculation of high criticality tasks leads to accurate and non-aggressive system mode switches.

Although combining task-level and system-level scheduling modes offers a higher flexibility and accuracy, it experiences a heavy overhead to calculate real-time scheduling decisions. Thus, such a combination is not suitable for the scheduling of safety critical real-time systems.

As a future work, we aim to study potential optimizations of the proposed algorithm overhead.

Acknowledgment

This work was in part funded by Independent Research Fund Denmark under grant number DFF-7017-00348, Compositional Verification of Real-time MULTI-CORE SAFETY Critical Systems.

References

[1] ISO 26262-1:2011 Road vehicles–Functional safety. Technical report, ISO, 2011.

[2] K. Abel Ouedraogo, J. Beugin, E. M. El Koursi, J. Clarhaut, D. Renaux, and F. Lisiecki. Toward an application guide for safety integrity level allocation in railway systems. *International Journal of Risk Analysis*, 2018.

[3] A.Burns and S. Baruah. Towards a more practical model for mixed criticality systems. In *Workshop on Mixed- Criticality Systems (co-located with RTSS)*, 2013.

[4] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *ECRTS 2012*, pages 145–154, 2012.

[5] S. Baruah, A. Burns, and Z. Guo. Scheduling mixed-criticality systems to guarantee some service under all non-erroneous behaviors. In *ECRTS 2016*, pages 131–138, July 2016.

[6] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Syst.*, 2(4):301–324, Oct. 1990.

[7] I. Bate, A. Burns, and R. I. Davis. A bailout protocol for mixed criticality systems. In *ECRTS 2015*, pages 259–268, July 2015.

[8] G. Behrmann, A. David, and K. G. Larsen. *A Tutorial on Uppaal*, pages 200–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[9] J. Boudjadar. An efficient energy-driven scheduling of dvfs-multicore systems with a hierarchy of shared memories. In *IEEE/ACM 21st DS-RT Conference*, pages 1–8, 2017.

[10] A. Burns and R. I. Davis. A survey of research into mixed criticality systems. *ACM Comput. Surv.*, 50(6):82:1–82:37, Nov. 2017.

[11] A. Burns, R. I. Davis, S. Baruah, and I. Bate. Robust mixed-criticality systems. *IEEE Transactions on Computers*, To appear, 2018.

[12] A. Burns and B. Littlewood. Reasoning about the reliability of multi-version, diverse real-time systems. In *2010 31st IEEE Real-Time Systems Symposium*, pages 73–81, 2010.

[13] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *RTSS'09*, pages 291–300, 2009.

[14] R. Dodd. Coloured petri net modelling of a generic avionics missions computer. Technical report, Department of Defence, Australia, Air Operations Division, 2006.

[15] A. Easwaran. Demand-based scheduling of mixed-criticality sporadic tasks on one processor. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 78–87, Dec 2013.

[16] P. Ekberg and W. Yi. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Real-Time Systems*, 50(1):48–86, Jan 2014.

[17] J. P. Erickson, N. Kim, and J. H. Anderson. Recovering from overload in multicore mixed-criticality systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 775–785, May 2015.

[18] T. Fleming and A. Burns. Incorporating the notion of importance into mixed criticality systems. In *Proceedings of Workshop on Mixed Criticality Systems*, page 33, 2014.

[19] O. Gettings, S. Quinton, and R. I. Davis. Mixed criticality systems with weakly-hard constraints. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, pages 237–246. ACM, 2015.

[20] X. Gu and A. Easwaran. Dynamic budget management with service guarantees for mixed-criticality systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 47–56. IEEE, 2016.

[21] X. Gu, A. Easwaran, K.-M. Phan, and I. Shin. Resource efficient isolation mechanisms in mixed-criticality scheduling. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 13–24, July 2015.

[22] G. Howard, M. Butler, J. Colley, and V. Sassone. Formal analysis of safety and security requirements of critical systems supported by an extended stpa methodology. In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pages 174–180, 2017.

[23] B. Hu, K. Huang, P. Huang, L. Thiele, and A. Knoll. On-the-fly fast overrun budgeting for mixed-criticality systems. In *Proceedings of the IEEE & ACM International Conference on Embedded Software*, pages 1–10. IEEE, 2016.

[24] P. Huang, G. Giannopoulou, N. Stoimenov, and L. Thiele. Service adaptions for mixed-criticality systems. In *In Proceedings of ASP-DAC*, 2014.

[25] P. Huang, P. Kumar, G. Giannopoulou, and L. Thiele. Run and be safe: Mixed-criticality scheduling with temporary processor speedup. In *DATE 2015*, 2015.

[26] P. Huang, P. Kumar, N. Stoimenov, and L. Thiele. Interference constraint graph - a new specification for mixed-criticality systems. In *ETFA 2013*, pages 1–8, 2013.

[27] P. Huyck. Arinc 653 and multi-core microprocessors; considerations and potential impacts. In *DASC'12*, pages 6B4–1–6B4–7, 2012.

[28] M. Jan, L. Zaourar, and M. Pitel. Maximizing the execution rate of low criticality tasks in mixed criticality system. In *Proceedings of Workshop on Mixed-Criticality, RTSS 2013*, pages 43–48, 2013.

[29] J. Lee, H. S. Chwa, L. T. X. Phan, I. Shin, and I. Lee. MC-ADAPT: Adaptive task dropping in mixed-criticality scheduling. *ACM Trans. Embed. Comput. Syst.*, 16(5s):163:1–163:21, Sept. 2017.

[30] D. Liu, N. Guan, J. Spasic, G. Chen, S. Liu, T. Stefanov, and W. Yi. Scheduling analysis of imprecise mixed-criticality real-time tasks. *IEEE Transactions on Computers*, 2018.

[31] D. Liu, J. Spasic, N. Guan, G. Chen, S. Liu, T. Stefanov, and W. Yi. Edf-vd scheduling of mixed-criticality systems with degraded quality guarantees. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 35–46, 2016.

[32] A. Löfwenmark and S. Nadjm-Tehrani. Understanding shared memory bank access interference in multi-core avionics. In *Proceedings of WCET'16*, OpenAccess Series in Informatics (OASIcs), 2016.

[33] B. Madzar, J. Boudjadar, J. Dingel, T. E. Fuhrman, and S. Ramesh. Formal analysis of predictable data flow in fault-tolerant multicore systems. In *FACS '16*, pages 153–171, 2016.

[34] A. V. Papadopoulos, E. Bini, S. Baruah, and A. Burns. AdaptMC: A control-theoretic approach for achieving resilience in mixed-criticality systems. In *30th Euromicro Conference on Real-Time Systems, ECRTS 2018*, pages 14:1–14:22, 2018.

[35] T. Park and S. Kim. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *Proceedings of EMSOFT '11*, pages 253–262. ACM, 2011.

[36] R. M. Pathan. Improving the Quality-of-Service for Scheduling Mixed-Criticality Systems on Multiprocessors. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:22, Dagstuhl, Germany, 2017.

[37] J. Ren and L. T. X. Phan. Mixed-criticality scheduling on multiprocessors using task grouping. In *27th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 25–34, July 2015.

[38] F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with FP. In *ECRTS '12*, pages 155–165, July 2012.

[39] F. Santy, G. Raravi, G. Nelissen, V. Nelis, P. Kumar, J. Goossens, and E. Tovar. Two protocols to reduce the criticality level of multiprocessor mixed-criticality systems. In *Proceedings of RTNS '13*, pages 183–192, New York, NY, USA, 2013. ACM.

[40] H. Su, P. Deng, D. Zhu, and Q. Zhu. Fixed-priority dual-rate mixed-criticality systems: Schedulability analysis and performance optimization. In *Proceedings of RTCSA*, 2016.

[41] H. Su, N. Guan, and D. Zhu. Service guarantee exploration for mixed-criticality systems. In *Proceedings of RTCSA*, pages 1–10, Aug 2014.

[42] H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 147–152, 2013.

[43] H. Su, D. Zhu, and S. Brandt. An elastic mixed-criticality task model and early-release edf scheduling algorithms. *ACM Trans. Des. Autom. Electron. Syst.*, 22(2):28:1–28:25, Dec. 2016.

[44] H. Su, D. Zhu, and J. Zhu. On the implementation of rt-fair scheduling framework in linux. In *IUCC 2015*, pages 1258–1265, 2015.

[45] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 239–243, 2007.