An Elastic Mixed-Criticality Task Model and Its Scheduling Algorithm

Hang Su and Dakai Zhu
University of Texas at San Antonio
{hsu, dzhu}@cs.utsa.edu

***Abstract*—To address the service abrupt problem for low-criticality tasks in existing mixed-criticality scheduling algorithms, we study an *Elastic Mixed-Criticality (E-MC)* task model, where the key idea is to have variable periods (i.e., service intervals) for low-criticality tasks. The minimum service requirement of a low-criticality task is ensured by its largest period. However, at runtime low-criticality tasks can be released early by exploiting the slack time generated from the over-provisioned execution time for high-criticality tasks to reduce their service intervals and thus improve their service levels. We propose an *Early-Release EDF (ER-EDF)* scheduling algorithm, which can judiciously manage the early release of low-criticality tasks without affecting the timeliness of high-criticality tasks. Compared to the state-of-the-art EDF-VD scheduling algorithm, our simulation results show that the ER-EDF can successfully schedule much more task sets. Moreover, the achieved execution frequencies of low-criticality tasks can also be significantly improved under ER-EDF.**

I. Introduction

In cyber-physical systems (CPS), which have been denoted as the next-generation engineering systems, the computation tasks can have different levels of importance according to their functionalities that further lead to different criticality levels [1]. To incorporate various certification requirements and enable efficient scheduling of such tasks, the *mixed-criticality* task model has been studied recently [3], [8], [10], where a task generally has multiple worst case execution times (WCETs) according to different certification levels.

Considering the increasing need to execute tasks with mutiple criticality levels on a shared computing system, how to efficiently schedule such mixed-criticality tasks while satisfying their specific requirements has been identified as one of the most fundamental issues in CPS [3]. Note that, without proper provisions for such mixed-criticality tasks, traditional scheduling algorithms are likely to cause the so-called *"priority inversion"* problems [8]. In [11], Vestal first defined and formalized the mixed-criticality scheduling problem with mutiple certification requirements at different degrees of confidence and studied a fixed priority algorithm. For sporadic mixed-criticality tasks, a hybrid-priority algorithm that combines EDF and Vestal's fixed priority algorithm was studied in [4].

From a different aspect of mixed-criticality tasks, De Niz *et al.* proposed a zero-slack scheduling approach, which works on the top of fixed-priority based preemptive scheduling algorithm (such as RMS) [8]. More recently, for the scheduling of sporadic mixed-criticality task systems, Baruah *et al.* proposed a more efficient scheduling algorithm, namely EDF-VD (virtual deadline), that assigns *virtual* (and smaller) deadlines for high-criticality tasks to ensure their schedulability in the worst case scenario [2]. In [10], Santy *et al.* studied an online scheme that calculates a delay-allowance for entering high-level state late and thus delays the cancellation of low-criticality tasks to improve their services.

Note that, most existing mixed-criticality scheduling algorithms guarantee the timeliness of high-criticality tasks in the worst case scenario at the expense of low-criticality tasks. For instance, when any high criticality task uses more time than its low-level WCET and causes the system to enter high-level execution mode, *all* low-criticality tasks will be discarded to provide the required computation capacity for high-criticality tasks [2], [3], [8], [10]. Such an approach can cause serious service abrupt and significant performance loss for low-criticality tasks, especially for control systems where the performance of controllers is mainly affected by the execution frequency and period of control tasks [12].

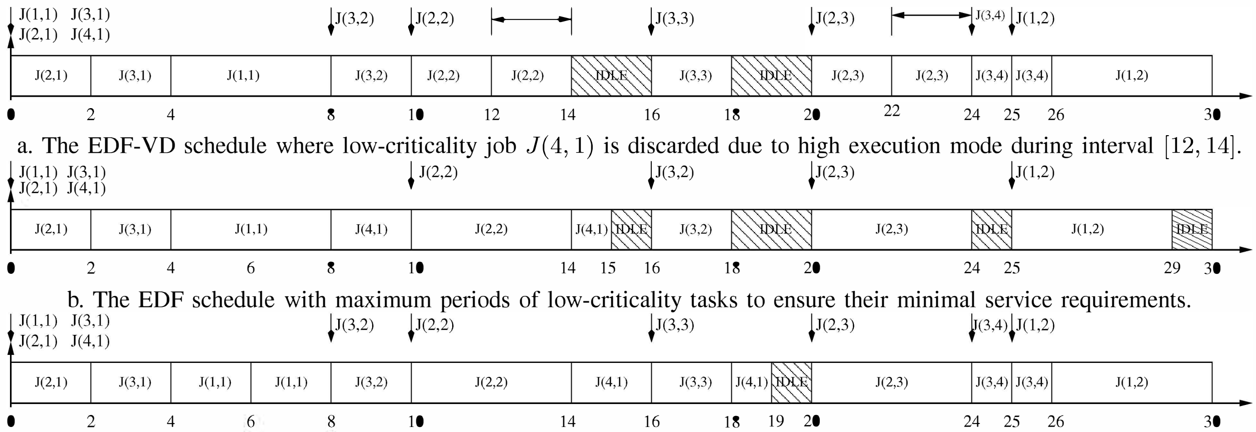
To address such service abrupt and provide minimal service guarantee for low-criticality tasks, we study in this work an *Elastic Mixed-Criticality (E-MC)* task model by adopting the idea of variable periods in elastic scheduling [6], [9]. Specifically, the largest period of a low-criticality task can be determined by its minimum service requirement. However, low-criticality tasks can be released early and more frequently at runtime to improve their service levels. We propose an *Early-Release EDF (ER-EDF)* scheduling algorithm that allows low-criticality tasks to release early without sacrificing the timeliness of high-criticality tasks. We analyze the schedulability of E-MC tasks under ER-EDF and evaluate its performance through extensive simulations.

The results show that, compared to the state-of-the-art EDF-VD mixed-criticality scheduling algorithm [2], the proposed E-MC task model and ER-EDF algorithm are much more effective in scheduling mixed-criticality tasks. First, when low-criticality tasks have reasonable minimum service requirements (with their periods being extended 2 to 5 times), ER-EDF can schedule more task sets than EDF-VD. In addition, the achieved execution frequencies (i.e., service levels) for low-criticality tasks with a few early-release points under ER-EDF are significantly better than those under EDF-VD.

The remainder of this paper is organized as follows. Section II presents the E-MC task model and a motivational example. The ER-EDF scheduling algorithm is presented and analyzed in Section III. The evaluation results are discussed in Section IV and Section V concludes the paper.

This work was supported in part by NSF awards CNS-0855247, CNS-1016974 and NSF CAREER Award CNS-0953005.

978-3-9815370-0-0/DATE13/ ©2013 EDAA



a. The EDF-VD schedule where low-criticality job $J(4,1)$ is discarded due to high execution mode during interval $[12,14]$.

b. The EDF schedule with maximum periods of low-criticality tasks to ensure their minimal service requirements.

c. The ER-EDF schedule where the low-criticality task τ_3 releases early at time 8, 16 and 24.

Fig. 1: Schedules for the example mixed-criticality task set within the interval $[0,30]$; $J(i,j)$ denotes the j^{th} job of τ_i.

II. ELASTIC MIXED-CRITICALITY TASK MODEL

We consider a set of n tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$ running on a uniprocessor system, where the criticality level of a task τ_i is denoted as ζ_i. In this work, we focus on systems with two different criticality levels[1], which are denoted as ζ^{low} and ζ^{high}, respectively. For a high-criticality task τ_i (i.e., $\zeta_i = \zeta^{high}$), the same as in the traditional mixed-criticality task model, it has a period p_i and two different worst case execution time (WCETs), which are denoted as c_i^{low} and c_i^{high} and correspond to its low-level and high-level execution requirements, respectively. Similarly, $u_i^{low} = \frac{c_i^{low}}{p_i}$ and $u_i^{high} = \frac{c_i^{high}}{p_i}$ represent task τ_i's low-level and high-level task utilizations, respectively.

The **major difference** between the *Elastic Mixed-Criticality (E-MC)* task model and the traditional mixed-criticality task model is how to represent low-criticality tasks. Here, a low-criticality task τ_i (i.e., $\zeta_i = \zeta^{low}$) with E-MC model has a *maximum* period p_i^{max} to reflect its *minimum* service requirements, in addition to a *desired* period p_i (which is mainly for existing mixed-criticality scheduling algorithms). Moreover, the low-criticality task τ_i has a set of k_i possible *early-release points* $P_i^{ER} = \{p_{i,1}, \cdots p_{i,k_i}\}$, where $c_i < p_{i,1} < \cdots < p_{i,k_i} < p_i^{max}$ and c_i is τ_i's WCET. The early-release points of low-criticality tasks enable them to release early and improve their execution frequencies at runtime through appropriate slack reclamation. The desired and minimum utilizations of task τ_i are defined as $u_i = \frac{c_i}{p_i}$ and $u_i^{min} = \frac{c_i}{p_i^{max}}$, respectively.

A set of E-MC tasks is said to be **E-MC scheduleable** if the high-criticality tasks' high-level execution requirements and low-criticality tasks' minimum service requirements can be guaranteed in the worst case scenario. Following the notations in [2], we define $U(H,L) = \sum_{\tau_i \in \Gamma}^{\zeta_i = \zeta^{high}} u_i^{low}$ and $U(H,H) = \sum_{\tau_i \in \Gamma}^{\zeta_i = \zeta^{high}} u_i^{high}$ as the low-level and high-level utilizations of high-criticality tasks, respectively. Similarly, for low-criticality tasks, $U(L,L) = \sum_{\tau_i \in \Gamma}^{\zeta_i = \zeta^{low}} u_i$ and $U(L,min) = \sum_{\tau_i \in \Gamma}^{\zeta_i = \zeta^{low}} u_i^{min}$ represent their desired and minimum utilizations, respectively. Based on the notations, we can easily get the following lemma.

Lemma 1: A set of E-MC tasks is E-MC schedulable under EDF if there is $U(H,H) + U(L,min) \leq 1$.

[1]We will study E-MC tasks with more criticality levels in our future work.

A. A Motivational Example

We first show the effectiveness of E-MC in modeling and scheduling mixed-criticality tasks through a concrete example. There are four tasks in the example, where the first two are high-criticality tasks and the other two are low-criticality tasks. Their timing parameters are given in Table I.

	Basic MC Parameters			Parameters for E-MC		EDF-VD [2]
	ζ_i	c_i	p_i	p_i^{max}	P_i^{ER}	Virtual Deadline
τ_1	ζ^{high}	{4,10}	25	-	-	13.85
τ_2	ζ^{high}	{2,4}	10	-	-	5.54
τ_3	ζ^{low}	2	8	16	{8}	8
τ_4	ζ^{low}	3	30	40	{30}	30

TABLE I: An Example Task Set

From the tasks' basic MC parameters, we can find that they are scheduleable under EDF-VD [2], where the virtual deadlines of tasks are shown in the last column. Suppose that the second and third jobs of task τ_2 take its high-level WCET c_2^{high} while other high-criticality jobs take their corresponding low-level WCETs, the EDF-VD schedule (which relies on tasks' virtual deadlines) of the task set within the interval $[0,30]$ is shown in Figure 1a. The system enters high-level execution mode at time 12 and the active low-criticality job $J(4,1)$ during the high-level interval $[12,14]$ is discarded, which leads to service abrupt for τ_4 during its first period.

Note that there is $U(H,H) + U(L,L) > 1$ for the example task set and it is impossible to guarantee the *desired* service levels for the low-criticality tasks τ_3 and τ_4. However, if the minimum service requirements of τ_3 and τ_4 can be specified by their maximum periods, which are 16 and 40 respectively, such requirements can be guaranteed under EDF since there is

$U(H,H)+U(L,min)=1$. The corresponding EDF schedule for interval $[0,30]$ is shown in Figure 1b.

Here, we can see that there are several idle intervals (i.e., slack) in the EDF schedule since not all high-criticality jobs take their high-level WCETs. Such slack can be exploited to improve the execution frequencies of low-criticality tasks. For instance, task τ_3 can release its second job $J(3,2)$ at time 8 (its early-release point) instead of waiting until time 16 (its maximum period) as shown in Figure 1c. It is clear that such early releases of low-criticality tasks have to be managed with great care to not affect the timeliness of high-criticality tasks.

III. Early-Release EDF Scheduling Algorithm

In this section, we study an *Early-Release EDF (ER-EDF)* scheduling algorithm, which enables low-criticality tasks to release earlier than their maximum periods and thus to improve their service levels. To prevent such early releases from affecting the execution of high-criticality tasks, there are a few key issues to address in ER-EDF. First and the most important is to determine *the deadline of a low-criticality job should it be released early*. Suppose that the first j jobs of a low-criticality task τ_i have arrived regularly according to its maximum period p_i^{max}, where the j^{th} job $J(i,j)$ arrived at time $r_{i,j}$ and has its deadline at time $d_{i,j}=r_{i,j}+p_i^{max}$. Moreover, let's assume that $J(i,j)$ finished its execution no later than $r_{i,j}+p_{i,x}$, where $p_{i,x}$ $(x=1,\cdots,k_i)$ is one of τ_i's early-release points.

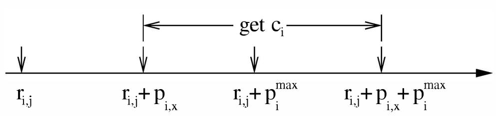


Fig. 2: The deadline of an early-release job.

As shown in Figure 2, if the next job $J(i,j+1)$ is released at the early release point $r_{i,j+1}=r_{i,j}+p_{i,x}$, the new deadline for $J(i,j+1)$ would be $d_{i,j+1}=r_{i,j+1}+p_i^{max}$. That is, the *expected* deadline of a low-criticality job is always assigned according to the task's maximum period. First, such deadline assignment for early-released low-criticality jobs ensures their minimum service requirements. Moreover, it ensures the same number of early-release points before a job's deadline and enables uniform handling of future early releases.

Once we know how to assign deadlines for early-release jobs, the second important issue in ER-EDF is to determine whether it is *feasible* to release a job at a given early-release point. Intuitively, releasing a job at an earlier time point (with an earlier deadline) introduces extra workload to the system. Therefore, to avoid overload condition and affecting the execution of other (especially high-criticality) tasks, such early-release decisions require judicious slack allocation and demand efficient slack management [7], [13]. In this work, we adopt and extend the wrapper-task mechanism [13], which has shown to be an effective and efficient slack management technique. In what follows, we first discuss the early-release decision algorithm. The wrapper-task based slack management technique will be detailed in Section III-B.

Algorithm 1 : Early-Release Algorithm of ER-EDF

1: **Input:** τ_i, $p_{i,x}$ and $SlackQ(t)$ at current time $t=r_i+p_{i,x}$
2: $S_{demand}=c_i-p_{i,x}\cdot\frac{c_i}{p_i^{max}}$;//calculate demanded slack
3: **if** ($S_{demand}\le$ *CheckSlack*($SlackQ(t)$, $t+p_i^{max}$)) **then**
4:     //next job of τ_i can be released earlier at time t
5:     *ReclaimSlack*($SlackQ(t)$, S_{demand});
6:     $r_i=t; d_i=t+p_i^{max}$; //reset release time and deadline
7:     *Enqueue*($ReadyQ$, J_i); //add the job to ready queue
8: **else**
9:     //set the next early-release point for τ_i (if $x<k_i$);
10:     *SetTimer*($r_i+p_{i,x+1}$);
11: **end if**

A. Early-Release Decisions

As the *centerpiece* of ER-EDF, the major steps for the early-release decision algorithm are shown in Algorithm 1. The algorithm will be invoked at any low-criticality task τ_i's early-release time point $t=r_i+p_{i,x}$, which is after the finish time of its current job that is released at time r_i. Moreover, we assume that $SlackQ(t)$ contains the available slack at time t and $ReadyQ$ holds all arrived ready jobs.

The amount of slack S_{demand}, which is needed for task τ_i to *safely* release its next job at its early-release time point t $(=r_i+p_{i,x})$, is first calculated (line 2). From previous discussions, the deadline of the to-be-released job of τ_i will be $t+p_i^{max}$ (as shown in Figure 2). Note that, the minimum service requirement of τ_i is assumed to be statically guaranteed. Therefore, during the interval $[r_i+p_i^{max}, t+p_i^{max}]$, which is between the current and new deadlines of τ_i and has the length of $p_{i,x}$, the amount of time that can *safely* be allocated to τ_i is $p_{i,x}\cdot u_i^{min}=p_{i,x}\cdot\frac{c_i}{p_i^{max}}$ [5]. Here, the new job of τ_i would need c_i time units within the interval $[t,t+p_i^{max}]$ (as shown in Figure 2) should it be released at time t. Hence, we can get S_{demand} as shown in the algorithm.

From [13], we know that slack has a deadline (i.e., its priority) in EDF-based scheduling and not all available slack can be utilized by a given task. Specifically, for task τ_i's new job that has a deadline at time $t+p_i^{max}$, only the slack that has its deadline no later than $t+p_i^{max}$ can be reclaimed. Here, to find out the amount of *reclaimable* slack before a given time d in the current slack queue Q, a function *CheckSlack*(Q,d) is used that will be discussed in detail in the next section.

If there is enough amount of reclaimable slack, S_{demand} units of slack will be reclaimed by task τ_i and be removed from the slack queue with the help of function *ReclaimSlack()* (line 5); Then, the new job of task τ_i is released and put into the ready job queue (lines 6 and 7). Otherwise, τ_i cannot release its job at time t. To give τ_i a chance to release its job at future early-release points (if any), a new timer is set at τ_i's next early-release point (line 10). Clearly, the number of early-release points has a great impact on the execution improvement of low-criticality tasks. Our evaluation results (see Section IV) show that a few (e.g., 5) such points are effective enough to improve low-criticality tasks' executions.

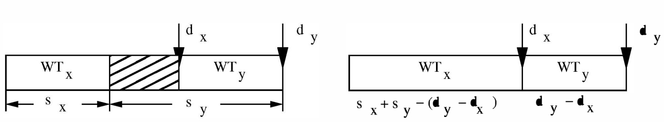
B. Slack Management with Wrapper-Tasks

Wrapper-task has been studied as an efficient mechanism to manage slack for energy savings and reliability enhancements [13]. In this work, we *extend* and exploit the wrapper-task approach to safely provide the needed slack and enable low-criticality tasks release their jobs at their early-release time points. Essentially, a wrapper-task (*WT*) represents a piece of dynamic slack with two parameters (s, d), where s denotes the amount of slack and d is the deadline that equals to that of the task giving rise to this slack [13].

Similar to ready tasks, which are kept in a *ready queue (ReadyQ)* in the increasing order of their deadlines (i.e., tasks with smaller deadlines are in front of *ReadQ* and tie is broken to favor the task with smaller index), wrapper-tasks are kept in a *slack queue (SlackQ)* in the increasing order of their deadlines as well. At runtime, unclaimed wrapper-tasks compete for the processor with ready tasks. With EDF-based scheduling, when both queues are *not empty* and the header wrapper-task WT_h of *SlackQ* has earlier deadline than *ReadyQ*'s header task τ_h, WT_h will *wrap* τ_h's execution by lending its time to τ_h. When the wrapped execution completes, τ_h returns its borrowed slack by creating a new piece slack with the length of wrapped execution and τ_h's deadline (i.e., the slack is actually *pushed forward* with a later deadline). When *ReadyQ* is empty, WT_h executes no-ops and the corresponding slack is wasted. The basic operations of wrapper-tasks and *SlackQ* can be summarized as follows [13]:

- *GenerateSlack(SlackQ, s, d):* Create a wrapper-task *WT* with parameters (s, d) and add it to *SlackQ* with increasing deadline order. Here, all wrapper-tasks in *SlackQ* represent dynamic slack with different deadlines. Therefore, the newly created *WT* may merge with an existing wrapper-task in *SlackQ* if they have the same deadline;
- *CheckSlack(SlackQ, d):* Find out the amount of *reclaimable* slack before time d (i.e., the total size of all wrapper-tasks that have their deadlines no later than d);
- *ReclaimSlack(SlackQ, s):* Reclaim the slack and remove wrapper-tasks from the front of *SlackQ*, which have accumulated size of s. The last wrapper-task may be partially removed by adjusting its remaining size.

Push-Backward Slack: At runtime, it is very likely that a job of high-criticality task τ_j completes early and only takes its low-level WCETs c_j^{low}. Here, the over-provisioned time for the job will turn to be slack with the operation *GenerateSlack(SlackQ,* $c_j^{high} - c_j^{low}$, d_j*)*, where d_j is the job's deadline. Moreover, it is highly possible for the slack to be pushed forward (and has a later deadline) with wrapped-executions. However, from previous discussion, we know that



a. Before slack push-backward b. After slack push-backward;

Fig. 3: Push slack backward to make it more reclaimable.

Algorithm 2 : *CheckSlack(Q,d)* with slack push-backward.

1: **Input:** Slack queue $Q = \{WT_1, \cdots, WT_m\}$ and time d;
2: **Output:** the amount of reclaimable slack;
3: **for** (WT_k: $k = m \rightarrow 2$) **do**
4:   **if** ($s_k > d_k - d_{k-1}$) **then**
5:     $s_{k-1} = s_{k-1} + s_k - (d_k - d_{k-1})$;
6:     $s_k = d_k - d_{k-1}$; //push slack backward
7:   **end if**
8: **end for**
9: $k = 1$; $S_r = 0$; //initialize the amount of reclaimable slack
10: **while** ($k \leq m$ AND $d_k \leq d$) **do**
11:   $S_r = S_r + s_k$; $k++$; //accumulate reclaimable slack
12: **end while**
13: **if** ($k \leq m$ AND $s_k > d_k - d$) **then**
14:   $S_r = S_r + s_k - (d_k - d)$; //part of WT_k is reclaimable
15: **end if**

low-criticality tasks prefer slack with earlier deadlines to obtain more reclaimable slack for their early releases.

Next, we show how slack can be pushed *backward* (with an earlier deadline) and become more reclaimable. Suppose that there are two pieces of slack represented by two wrapper-tasks $WT_x : (s_x, d_x)$ and $WT_y : (s_y, d_y)$ with $d_x < d_y$ as shown in Figure 3a. Here, the amount of reclaimable slack before d_x and d_y are s_x and $s_x + s_y$, respectively.

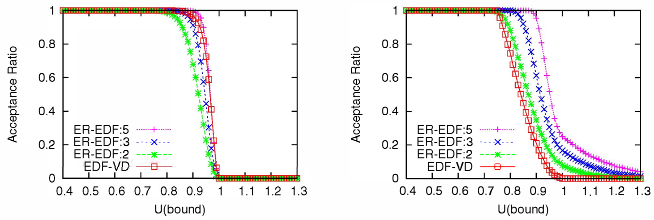
If there is $d_y - d_x < s_y$, for the slack represented by WT_y, at most $(d_y - d_x)$ units can be consumed after time d_x. That is, for part of slack (with $s_y - (d_y - d_x)$ units) represented by WT_y (dashed part in Figure 3a), we can safely push it backward and make that part has the deadline of d_x. After such transformation, the updated wrapper-tasks WT_x and WT_y are shown in Figure 3b. Although the amount of reclaimable slack before d_y remains the same, the one before d_x increases to be $s_x + s_y - (d_y - d_x)$. Note that, if there are wrapper-tasks with deadlines earlier than d_x, such push-backward can be performed iteratively.

Algorithm 2 details the steps of function *CheckSlack(Q, d)* with slack push-backward being considered. Here, all slack is pushed backward (if possible) to get more reclaimable slack before time d (lines 3 to 8). Then, the amount of reclaimable slack, including all completely reclaimable wrapper-tasks (lines 10 to 12) and the last partially reclaimable wrapper-task (lines 13 to 15), is accumulated.

C. Analysis of ER-EDF

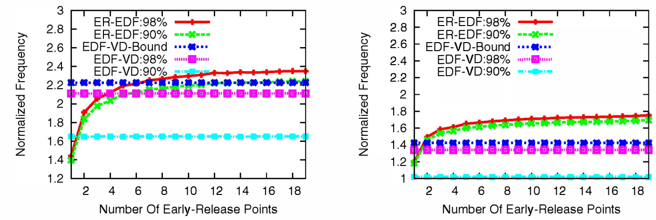
From Section III-A, the time allocated to an early-release job of a low-criticality task comes from two parts: *its own contribution* and *reclaimed slack*. Based on the results in [5] and [13], both parts can be safely utilized by the early-release job without introducing extra workload and affecting other (current and future) tasks' allocations. Hence, we can have the following theorem regarding to the correctness of ER-EDF.

Theorem 1: An E-MC task set can be successfully scheduled under ER-EDF without causing any deadline miss if there is $U(H, H) + U(L, min) \leq 1$.



a. $\frac{c^{low}}{c^{high}} \in [0.01, 0.3]$ b. $\frac{c^{low}}{c^{high}} \in [0.01, 0.9]$;

Fig. 4: Acceptance ratio; $Prob(HIGH) = 0.5$;



a. $\frac{c^{low}}{c^{high}} = 0.1$; b. $\frac{c^{low}}{c^{high}} = 0.5$;

Fig. 5: Effects of early-release points; $U(H, H) = 0.7$.

IV. Evaluations and Discussions

In this section, we evaluate the effectiveness of our proposed E-MC task model and ER-EDF algorithm through extensive simulations. For comparison, we also implemented EDF-VD [2], the state-of-the-art mixed-criticality scheduler. Here, whenever a high-criticality task uses more time than its low-level WCET and causes the system to enter the high-level execution mode at runtime, EDF-VD will discard all (current and future) low-criticality tasks until there is no more ready high-criticality tasks (at that time, the system can safely switch back to the low-level execution mode) [2].

A. Acceptance Ratio of Schedulable Task Systems

With the focus on the minimum service requirements of low-criticality tasks, which are represented by their maximum periods in the E-MC task model, we first show that more task sets can be scheduled under ER-EDF compared to that of EDF-VD. The synthetic tasks are generated following a similar workload generation scheme as used in [2]. Here, the parameter $Prob(HIGH)$ denotes the probability of a generated task T_i being a high-criticality task. The periods of tasks are generated uniformly within the range of $[10, 100]$. The utilization of a task T_i is uniformly generated within the range $[0.02, 0.2]$. If T_i is a low-criticality task, it represents its normal utilization u_i. Otherwise, when T_i is a high-criticality task, it is actually task T_i's high-level utilization u_i^{high}. The low-level WCET c_i^{low} and utilization u_i^{low} of a high-criticality task T_i is further obtained from its low-to-high execution ratio $\frac{c_i^{low}}{c_i^{high}}$, which is uniformly generated within a given range. Tasks are generated until the utilization bound U_{bound} is met, where U_{bound} is the larger of high-criticality utilization $U(H, H)$ and low-criticality utilization $U(H, L)+U(L, L)$ [2].

Figure 4 shows the acceptance ratio, which is the number of schedulable task sets over total number of task sets generated, under different scheduling algorithms when U_{bound} varies within the range of $[0.4, 1.3]$. Here, we consider balanced workload and set $Prob(HIGH) = 0.5$. Similar results have been obtained with other settings. Moreover, for each data point, 1000 task sets are generated. The results for EDF-VD are in line with what has been reported in [2], where more task sets can be schedulable when the low-to-high execution ratio for high-criticality tasks is smaller.

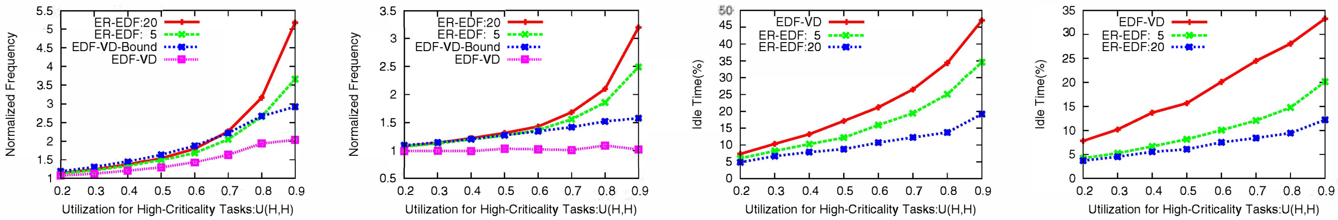
For ER-EDF, to specify a low-criticality task's minimum service requirement, its maximum period is obtained by extending its desired period upto k times (i.e., its minimum service requirement is no worse than one-k^{th} of its desired service level under EDF-VD) and the corresponding ratio of schedulable tasks is denoted by *ER-EDF:k*. Here, we can see that, if low-criticality tasks' periods can be extended to 5 times, the number of scheduleable task sets under ER-EDF is comparable to that of EDF-VD even when the low-to-high execution ratio of high-criticality tasks is small (Figure 4a). For cases where the low-to-high execution ratio of high-criticality tasks is large (Figure 4b), extending the low-criticality tasks' periods to only 2 times can achieve more schedulable task sets under ER-EDF. In the following experiments, we set $k = 3$.

B. Effects of Early-Release Points

Next, we evaluate how the number of early-release points for low-criticality tasks can affect their performance under ER-EDF. The task sets are generated as described above with fixed $U(H, H) = 0.7$ and $\frac{c^{low}}{c^{high}} = 0.1$ or 0.5. The maximum periods of low-criticality tasks are set such that $U(H, H) + U(L, min) = 1.0$, which makes it schedulable under ER-EDF. The early-release points are assumed to uniformly distribute within the period of a low-criticality task.

The results are shown in Figure 5. Here, each data point represents the average of 100 task sets. The Y-axis is the normalized execution frequencies for low-criticality tasks with the one corresponding to their maximum periods (i.e., minimum service levels) being the baseline. ER-EDF:$x\%$ and EDF-VD:$x\%$ denotes the achieved frequencies, when the probability of high-criticality tasks taking their low-level WCETs is $x\%$ (where $x = 90$ and 98), under ER-EDF and EDF-VD, respectively. EDF-VD-Bound shows the highest frequency if no low-criticality task is discarded under EDF-VD.

Not surprisingly, having more early-release points generally results in better execution frequencies for low-criticality tasks under ER-EDF since it gives those tasks more opportunities to execute more (but with higher overhead). Compared to that of EDF-VD, having a few (e.g., 5) early-release points is sufficient for ER-EDF to obtain comparable (or better) execution frequencies for low-criticality tasks. Moreover, when high-criticality tasks are more likely to take their low-level WCETs, more slack can be expected at runtime and fewer low-criticality tasks may be discarded, which leads to better frequency improvements under both EDF-VD and ER-EDF.



a. frequency; $\frac{c^{low}}{c^{high}} = 0.1$; b. frequency; $\frac{c^{low}}{c^{high}} = 0.5$; c. idle time; $\frac{c^{low}}{c^{high}} = 0.1$; d. idle time; $\frac{c^{low}}{c^{high}} = 0.5$;

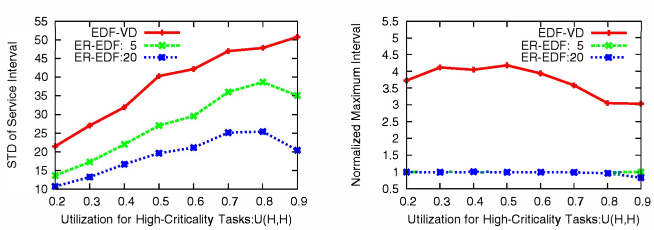
Fig. 6: The effects of $U(H,H)$ on the execution frequencies for low-criticality tasks; $Prob(c^{low}) = 90\%$ and $\frac{p_i^{max}}{p_i} \leq 3$.

C. *Effects of High-Criticality Task Utilization $U(H,H)$*

To evaluate how ER-EDF performs under different workloads, Figure 6 shows the achieved execution frequencies for low-criticality tasks when $U(H,H)$ varies from 0.2 to 0.9. Here, the low-to-high execution ratio of high-criticality tasks is set as 0.1 and 0.5. The probability of high-criticality tasks taking their low-level WCETs is 90%. We consider 5 or 20 early-release points for low-criticality tasks in ER-EDF, which are denoted as ER-EDF: 20 and ER-EDF: 5, respectively.

From Figure 6a (where $\frac{c^{low}}{c^{high}} = 0.1$), we can see that higher normalized execution frequencies for low-criticality tasks can be obtained as $U(H,H)$ increases. For EDF-VD, it comes from the fact that, to make the task set scheduleable under ER-EDF, the maximum periods of low-criticality tasks are relatively larger and the baseline frequencies become smaller at higher $U(H,H)$. For ER-EDF, it is mainly due to increased amount of available slack at runtime when there are more high-criticality tasks. When there are 20 early-release points, ER-EDF can obtain upto 2.5 times better execution frequencies for low-criticality tasks compared to that of EDF-VD. Similar results for the case of $\frac{c^{low}}{c^{high}} = 0.5$ are shown in Figure 6b. The idle time in the result schedules is shown in Figure 6cd, which further illustrates that ER-EDF can make better use of slack and thus provide better service for low-criticality tasks.

Figures 7ab show the standard deviation and normalized maximum execution intervals between consecutive served jobs for low-criticality tasks, respectively. We can see that the low-criticality tasks are executed more smoothly under ER-EDF. The maximum interval under EDF-VD can be as large as 4 times compared to ER-EDF due to discarded jobs under EDF-VD at high-level execution mode.



a. standard deviation; b. maximum interval

Fig. 7: Interval; $\frac{c^{low}}{c^{high}} = 0.1$, $Prob(c^{low}) = 90\%$, $\frac{p_i^{max}}{p_i} \leq 3$.

V. Conclusions

In this work, we study an *Elastic Mixed-Criticality (E-MC)* task model to address the service abrupt problem for low-criticality tasks in conventional mixed-criticality scheduling algorithms. The central idea of E-MC is to have variable periods (i.e., service intervals) for low-criticality tasks, where the minimal service requirement is guaranteed by their largest periods. To improve their execution frequencies, we propose an *Early-Release EDF (ER-EDF)* scheduling algorithm that allows low-criticality tasks to release earlier than their largest periods at runtime by judiciously exploiting slack without affecting the timeliness of high-criticality tasks. Our simulation results confirm the effectiveness of the E-MC model and ER-EDF algorithm in scheduling mixed-criticality tasks when comparing to the state-of-the-art EDF-VD algorithm [2].

References

[1] J. Barhorst, T. Belote, P. Binns, J. Hoffman, J. Paunicka, P. Sarathy, J. Scoredos, P. Stanfill, D. Stuart, and R. Urzi. A research agenda for mixed-criticality systems. *Cyber-Physical Systems Week*, 2009.

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

[3] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *RTAS*, pages 13–22, 2010.

[4] S. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *ECRTS*, pages 147–155, 2008.

[5] S.A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes. In *RTSS*, pages 396–407, 2003.

[6] G. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *RTSS*, pages 286–295, 1998.

[7] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *RTSS*, pages 295–304, 2000.

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

[9] T.W. Kuo and A.K. Mok. Load adjustment in adaptive real-time systems. In *RTSS*, pages 160–170, 1991.

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

[11] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS*, 2007.

[12] F. Zhang, K. Szwaykowska, V. Mooney, and W. Wolf. Task scheduling for control oriented requirements for cyber-physical systems. In *RTSS*, pages 47–56, 2008.

[13] D. Zhu and H. Aydin. Reliability-aware energy management for periodic real-time tasks. *IEEE Trans. on Computers*, 58(10):1382–1397, 2009.