Technical Report: CS-TR-2016-002
Department of Computer Science
The University of Texas at San Antonio

An Elastic Mixed-Criticality Task Model and Early-Release EDF Scheduling Algorithms

HANG SU, University of Texas at San Antonio
DAKAI ZHU, University of Texas at San Antonio
SCOTT BRANDT, University of California, Santa Cruz

Many algorithms have recently been studied for scheduling mixed-criticality (MC) tasks. However, most existing MC scheduling algorithms guarantee the timely executions of high-criticality (HC) tasks at the expense of discarding low-criticality (LC) tasks, which can cause serious service interruption for such tasks. In this work, aiming at providing guaranteed services for LC tasks, we study an *Elastic Mixed-Criticality (E-MC)* task model for dual-criticality systems. Specifically, the model allows each LC task to specify its *maximum* period (i.e., minimum service level) and a set of *early-release points*. We propose an *Early-Release (ER)* mechanism that enables LC tasks be released more frequently and thus improve their service levels at runtime, with both *conservative* and *aggressive* approaches to exploiting system slack being considered, which is applied to both EDF and preference-oriented earliest-deadline (POED) schedulers. We formally prove the correctness of the proposed ER-EDF scheduler on guaranteeing the timeliness of *all* tasks through judicious management of the early releases of LC tasks. The proposed model and schedulers are evaluated through extensive simulations. The results show that, by moderately relaxing the service requirements of LC tasks in MC task sets (i.e., by having LC tasks' maximum periods in the E-MC model be 2 to 3 times of their *desired* MC periods), most transformed E-MC task sets can be successfully scheduled without sacrificing the timeliness of HC tasks. Moreover, with the proposed ER mechanism, the runtime performance of tasks (e.g., execution frequencies of LC tasks, response times and jitters of HC tasks) can be significantly improved under the ER schedulers when compared to that of the state-of-the-art EDF-VD scheduler.

Categories and Subject Descriptors: D.4.1 [**Process Management**]: Scheduling; D.4.7 [**Organization and Design**]: Real-Time and Embedded Systems

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Mixed-criticality systems, real-time tasks, elastic task models, scheduling algorithms, earliest deadline first (EDF) scheduling, early-release

ACM Reference Format:
Hang Su, Dakai Zhu, and Scott Brandt, 2016. An Elastic Mixed-Criticality Task Model and Early-Release EDF Scheduling Algorithms. *ACM* 1, 1, Article 1 (February 2016), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

As the next-generation engineering systems, Cyber-Physical Systems (CPS) can have computation tasks with different levels of importance according to their functionalities, which further lead to different criticality levels for those tasks [Barhorst et al. 2009]. For example, in avionics systems, it can be more important to guarantee the cor-

This work was supported in part by the National Science Foundation under CAREER Award CNS-0953005 and grant CNS-1422709. Author's addresses: Hang Su and Dakai Zhu, Department of Computer Science, University of Texas at San Antonio; Scott Brandt, Computer Science Department, University of California, Santa Cruz.

DOI: http://dx.doi.org/10.1145/0000000.0000000

rect executions of flight-critical functionalities (such as cruise control) than mission-critical functionalities (e.g., capturing photos) [Vestal 2007]. Considering the increasing need to execute tasks with multiple criticality levels on a shared computing platform, how to efficiently schedule such tasks while satisfying their specific requirements has been identified as one of the most fundamental issues in CPS [Baruah et al. 2010b].

To incorporate various certification requirements and enable efficient scheduling of such tasks, the *mixed-criticality (MC)* task model has been studied very recently [Vestal 2007]. In general, a MC task has multiple worst case execution times (WCETs) corresponding to different certification levels, where higher certification levels normally result in much larger WCETs. In his seminal work [Vestal 2007], Vestal first formalized the mixed-criticality scheduling problem by considering multiple certification requirements at different degrees of confidence and studied a fixed-priority based scheduling algorithm. Following this line of research, several studies have been reported by considering various task and system models as well as different scheduling algorithms [Baruah et al. 2012; Baruah et al. 2010b; Baruah and Vestal 2008; de Niz et al. 2009; Kelly et al. 2011; Li and Baruah 2012].

Note that, in most existing mixed-criticality scheduling algorithms, when any high-criticality task uses more time than its low-level WCET and causes the system to enter a high-level execution mode, **all** low-criticality tasks will be **discarded** to accommodate the additional computation demands of high-criticality tasks [Baruah et al. 2012; Baruah et al. 2010b; Baruah and Vestal 2008; de Niz et al. 2009; Vestal 2007]. Although this approach is very effective to guarantee the timely executions of high-criticality tasks, the cancellation of low-criticality tasks can cause serious service interruption and significant performance loss, especially for control systems where the performance is mainly determined by the execution frequencies and periods of its control tasks [Zhang et al. 2008]. To mitigate such effects of task cancellation in mixed-criticality systems, several studies have been reported recently (including delayed cancellation [Santy et al. 2012] and bailout protocol [Bate et al. 2015]). However, none of these approaches can provide guaranteed service intervals for low-criticality tasks under the high-level execution mode.

Instead of *completely* discarding low-criticality tasks under the high-level execution mode, it is possible to reduce their computation demands by increasing their service intervals (i.e., periods). Following this idea and aiming at providing minimum service guarantees for low-criticality tasks, in this work, we study an *Elastic Mixed-Criticality (E-MC)* task model for dual-criticality systems. Specifically, by adopting the idea of variable periods in elastic scheduling [Buttazzo et al. 1998; Kuo and Mok 1991], each low-criticality task has a maximum period that determines its minimum service level, which can be guaranteed offline. Moreover, each low-criticality task can have a set of *early-release* points, which can be exploited by our proposed *Early-Release (ER)* mechanism to release its instances more frequently (thus improve its service levels) at runtime through appropriate slack reclamation.

To the best of our knowledge, this is the first work that addresses guaranteed services for low-criticality tasks in mixed-criticality scheduling. Our contributions are summarized as follows (some preliminary results appeared in [Su and Zhu 2013]):

— We study an *Elastic Mixed-Criticality (E-MC)* task model, where each low-criticality task has a maximum period (to specify its minimum service requirement) and a set of early-release points within the maximum period;
— We propose an online *Early-Release (ER)* mechanism and consider both *conservative* and *aggressive* early-releases to improve performance of low-criticality tasks;
— We formally analyze and prove the correctness of the proposed ER-EDF scheduler on meeting the timing constraints of all (especially high-criticality) tasks;

— We investigate the application of ER mechanism to a preference-oriented earliest deadline scheduler, which can improve high-criticality tasks' control performance;
— We evaluate the proposed model and schedulers through extensive simulations. The results show that they are very effective to enhance system schedulability as well as to improve the control performance of both low- and high-criticality tasks with increased execution frequencies, and reduced response times and jitters, respectively.

The remainder of this paper is organized as follows. The closely related work is reviewed in Section 2. Section 3 presents a motivational example and the E-MC task model. The *Early-Release (ER)* mechanism and the ER-EDF scheduling algorithm are presented and analyzed in Section 4. Section 5 introduces a preference-oriented scheduler that aims at improving the control performance of high-criticality tasks. The evaluation results are discussed in Section 6 and Section 7 concludes the paper.

2. CLOSELY RELATED WORK

The mixed-criticality scheduling problem was first formalized by Vestal in his seminal work [Vestal 2007], where tasks with multiple certification requirements at different degrees of confidence and a fixed-priority scheduling algorithm were studied. For sporadic mixed-criticality tasks, a hybrid-priority scheme that combines EDF and Vestal's fixed-priority algorithm was studied in [Baruah and Vestal 2008]. Based on fixed-priority preemptive scheduling (such as RMS), a zero-slack scheduling approach was studied that prevents low-criticality tasks from interfering high-criticality tasks under overload condition, by scheduling tasks with different criticality levels based on their priorities until their *"zero slack"* time points [de Niz et al. 2009].

Focusing on a finite number of mixed-criticality jobs, Baruah et al. proposed the Own-Criticality Based Priority (OCBP) scheme, which adopts a repetitive "Audsley approach" and, if a given set of MC jobs is OCBP schedulable, a fixed-priority table with a given size is obtained [Baruah et al. 2010b]. The work was extended to consider multiple criticality levels and processor speedup factor analysis in [Baruah et al. 2010a]. A sufficient condition for OCBP schedulability was further derived via load-based schedulability analysis in [Li and Baruah 2010b]. By first prioritizing jobs according to EDF and then swapping adjacent low- and high-criticality jobs aiming at executing high-criticality jobs early, a Mixed-Critical EDF (MCEDF) was studied in [Socci et al. 2012], which was shown to dominate OCBP-based schemes with higher schedulability ratio and reduced complexity. In contrast, the Criticality-Based EDF (CBEDF) postpones the executions of high-criticality jobs with the objective of allocating empty time slots to low-criticality jobs for their early executions, which was also shown to outperform OCBP in terms of schedulability of tasks [Park and Kim 2011].

For sporadic mixed-criticality tasks, an extension of OCBP was studied that has a pseudo-polynomial complexity [Li and Baruah 2010a] and a more efficient algorithm with reduced complexity was further proposed in [Guan et al. 2011]. More recently, an *EDF-VD (virtual deadline)* scheduler was studied for dual-criticality systems, where the basic idea is to assign virtual (and smaller) deadlines for high-criticality tasks to ensure their schedulability in the worst case scenario [Baruah et al. 2012]. Based on demand bound function (DBF) analysis, an extension of EDF-VD was studied that adopts an efficient relative deadline tuning technique and can achieve better schedulability [Ekberg and Yi 2012].

For multicore/multi-processor systems, several studies on mixed-criticality scheduling have also been reported [Kelly et al. 2011; Li and Baruah 2012; Mollison et al. 2010; Su et al. 2013]. A comprehensive review of the mixed-criticality scheduling algorithms for various tasks and systems can be found in [Burns and Davis 2015]. Note that, most existing mixed-criticality scheduling algorithms guarantee the timeliness

of high-criticality tasks in the worst case scenario at the expense of discarding low-criticality tasks [Baruah et al. 2012; Baruah et al. 2010b; de Niz et al. 2009; Ekberg and Yi 2012; Vestal 2007]. That is, when any high-criticality task uses more time than its low-level WCET and causes the system to enter the high-level execution mode at runtime, all low-criticality tasks will be discarded to provide the additional computation capacity required by high-criticality tasks. Such cancellations can cause serious service interruption and significant performance loss for low-criticality tasks.

To mitigate the effects of such cancellations on low-criticality tasks, Santy et al. studied an online scheme that calculates a delay-allowance before entering the high-level execution mode and thus delays the cancellation of low-criticality tasks to improve their services under fixed-priority scheduling [Santy et al. 2012]. In [Bate et al. 2015], a bailout protocol was studied to mitigate the negative impacts of mode transitions for low-criticality tasks. Unfortunately, both the delayed approach and the bailout protocol cannot provide any guaranteed service for low-criticality tasks when a system enters the high-level execution mode.

Aiming at providing guaranteed services for low-criticality tasks, Huang et al. investigated schemes that allow service adaptations and determine the minimal service guarantees for low-critical tasks in the high-level execution mode [Huang et al. 2014]. Basically, by focusing on EDF-VD scheduling [Baruah et al. 2012] and extending the demand bound functions (DBFs) [Ekberg and Yi 2012], the main idea is to reduce the execution frequencies of low-criticality tasks by increasing their periods instead of completely canceling their executions when the system switches to the high-level execution mode. A bi-level deadline scaling scheme was further studied to improve the schedulability of tasks under the EDF-VD scheduler [Masrur et al. 2015]. Based on an improved demand bound analysis [Easwaran 2013; Zhang et al. 2014], we also studied the interplay between the guaranteed services for low-criticality tasks and investigated their trade-offs under different execution modes in our recent work [Su et al. 2014]. Different from the existing studies [Huang et al. 2014] and [Su et al. 2014], which focused on offline analysis, the work reported in this paper focuses on an online *Early-Release (ER)* mechanism and its application to EDF schedulers where the preliminary results have been reported in [Su and Zhu 2013].

3. AN ELASTIC MIXED-CRITICALITY (E-MC) TASK MODEL

In this work, we focus on dual-criticality systems where the low (LO) and high (HI) criticality levels are denoted as ζ^{low} and ζ^{high}, respectively. We consider a set of n tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$ that run on a uniprocessor system where the criticality level of a task τ_i is denoted as ζ_i.

3.1. E-MC Tasks: Guaranteed Services for Both HC and LC Tasks

For a high-criticality (HC) task τ_i (i.e., $\zeta_i = \zeta^{high}$), the same as in the conventional mixed-criticality (MC) model [Baruah et al. 2012], it is modeled with its period p_i, its LO and HI worst case execution times (WCETs) c_i^{low} and c_i^{high} where $c_i^{low} < c_i^{high}$. The LO and HI task utilizations of τ_i are given as $u_i^{low} = \frac{c_i^{low}}{p_i}$ and $u_i^{high} = \frac{c_i^{high}}{p_i}$, respectively. It is assumed that a HC task τ_i has implicit deadline of $d_i = p_i$.

One **key** point of the E-MC task model is to specify the **minimum service requirement** for a LC task τ_i (i.e., $\zeta_i = \zeta^{low}$) through its **E-MC period** p_i^{emc}, which stands for the maximum inter-arrival time for the consecutive jobs of τ_i. When the jobs of a LC task τ_i are released *regularly* at its E-MC period p_i^{emc}, its implicit deadline is denoted as $d_i = p_i^{emc}$. The WCET of a LC task τ_i is denoted as c_i where its minimum utilization is denoted as $u_i^{min} = \frac{c_i}{p_i^{emc}}$.

With the focus on providing minimum service guarantees for LC tasks while ensuring the high-level WCETs of HC tasks, we have the following definition.

Definition 3.1. [**E-MC schedulable**] A set of E-MC tasks is said to be **E-MC schedulable** under a given scheduling algorithm if the high-level execution requirements (i.e., c_i^{high}) of all HC tasks and the minimum service requirements (represented by p_i^{emc}) of all LC tasks can be guaranteed *simultaneously* in the worst case scenario.

Following the notations in [Baruah et al. 2012], 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 LO and HI utilizations for all HC tasks, respectively. Similarly, $U(L,min) = \sum_{\tau_i \in \Gamma}^{\zeta_i = \zeta^{low}} u_i^{min}$ represents the minimum utilization of all LC tasks. Based on these notations and the schedulability condition of the EDF scheduler [Liu and Layland 1973], we can easily obtain the following lemma.

LEMMA 3.2. *A set of E-MC tasks is E-MC schedulable under the EDF scheduler if and only if* $U(H,H) + U(L,min) \leq 1$.

Note that, in the MC model [Baruah et al. 2012], a system can run in different modes with different execution guarantees for tasks. Specifically, the MC model allows the executions of both LC and HC tasks in the LO execution mode. However, only HC tasks' executions are guaranteed in the HI execution mode. From the above definition, we can see that **the schedulability of E-MC tasks considers only the worst case scenario without differentiating a system's LO or HI running modes**. Here, the E-MC model follows a different design principle to provide a *uniform* guarantee for LC tasks on their minimum service levels and HC tasks on their high-level (and implicitly low-level) execution requirements under *all* circumstances. From a different perspective, we can also say that **both models provide the same guarantee for HC tasks while LC tasks have guaranteed executions only in the E-MC model**.

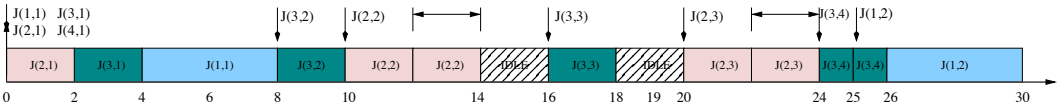
3.2. E-MC Tasks: Early-Release Points for LC Tasks

The main **novelty** of the E-MC model is to specify a set of k_i **early-release points** $ERP_i = \{p_{i,1}, \cdots, p_{i,k_i}\}$ for a LC task τ_i, where $c_i < p_{i,1} < \cdots < p_{i,k_i} < p_i^{emc}$. These early-release points provide opportunities for the LC task to release its instances more frequently than what is allowed by its maximum period and thus to improve its execution frequency at runtime. However, the early-released instances of LC tasks have implications on system workload, which has to be carefully addressed to avoid deadline misses (see Section 4.1 for further discussions).

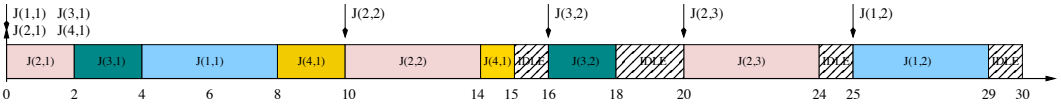
For easy comparison with existing MC scheduling algorithms and to link the E-MC model with MC task model, p_i *of a LC task* τ_i *in the MC model is defined as its* **desired** *period in the E-MC model*. Note that the E-MC periods of LC tasks can be much larger than their desired periods. That is, the guaranteed service for LC tasks in the E-MC model can be much worse than that under the LO running mode in the MC model. However, as shown in the evaluation results (Section 6), the runtime performance of LC tasks with our proposed early-release mechanism based on the E-MC model (see Section 4) can be much better than that of the existing MC scheduling algorithms.

3.3. A Motivational Example

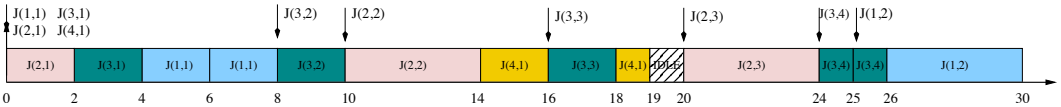
First, we consider a concrete example with four tasks, where their timing parameters are given in Table I. Here, tasks τ_1 and τ_2 are HC tasks while tasks τ_3 and τ_4 are LC tasks. From their timing parameters in the MC model, we can find that the task set is schedulable under the EDF-VD scheduler, where the virtual deadlines of HC tasks can be found as in the table's last column [Baruah et al. 2012]. Suppose that the second and



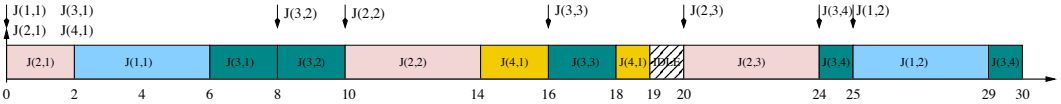
a. The EDF-VD schedule where the LC job $J(4,1)$ is discarded within the interval $[12,14]$.



b. The EDF schedule where LC tasks are released according to their maximum periods.



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



d. The ER-POED schedule where the HC task τ_1 completes early at time 6 and 29.

Fig. 1: Different Schedules for the example E-MC task set within the interval $[0,30]$; The arrows indicate the arrivals of task instances, and $J(i,j)$ denotes the j^{th} job of τ_i.

	For both MC and **E-MC** Models			**E-MC Model only**		EDF-VD
	ζ_i	c_i	p_i	p_i^{emc}	ERP_i	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}	-
τ_4	ζ^{low}	3	30	40	{30}	-

Table I: An Example Task Set

third instances (i.e., jobs) of the HC task τ_2 take its high-level WCET c_2^{high} while other HC jobs take their corresponding low-level WCETs. The EDF-VD schedule of the task set within the interval $[0,30]$ is shown in Figure 1a. Here, the HC job $J(2,2)$ does not complete its execution after running for its low-level WCET at time 12 and thus causes the system to enter the HI execution mode. Hence, under the EDF-VD scheduler, the active LC job $J(4,1)$ is discarded within the interval $[12,14]$ [Baruah et al. 2012], which leads to service interruption for the LC task τ_4 during its first period.

Note that, there is $U(H,H)+U(L,L)>1$ for the example task set. Therefore, under the condition that HC tasks' high-level WCETs have to be guaranteed, it is impossible for the EDF scheduler to satisfy the *desired* service levels for the LC tasks τ_3 and τ_4 (which are indicated by their desired periods 8 and 30, respectively) in the HI execution mode. However, if tasks τ_3 and τ_4 can *relax* their service requirements with their extended maximum periods (which are 16 and 40, respectively), we have $U(H,H)+U(L,min)=1$ for the transformed E-MC task set. Hence, their minimum service levels can be guaranteed under the EDF scheduler in the worst case and the corresponding schedule for the tasks within the interval $[0,30]$ is shown in Figure 1b.

Here, we can see that there are several idle intervals (i.e., *system slack*) in the EDF schedule since not all HC jobs take their high-level WCETs at runtime. Such slack can be exploited to improve the execution frequencies of LC 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. Here, task τ_3 further releases its third and fourth instance early at time 16 and 24, respectively. It is clear that such early releases of LC tasks have to be managed with great care so that they will not affect the timely executions of HC tasks in the worst case scenario.

4. EARLY-RELEASE MECHANISM AND ER-EDF SCHEDULER

Intuitively, when a LC task releases a job at one of its early-release points, extra workload is introduced. Hence, to accommodate the early-released executions of LC tasks (where HC tasks are always released regularly at their periods) at runtime, the proposed *early-release EDF (ER-EDF)* scheduler has to exploit system slack judiciously to avoid deadline misses of other (especially HC) tasks. Note that, both the priority of an early-released job and its reclaimable slack depend directly on the job's deadline. Therefore, the first and ultimately important issue in the Early-Release (ER) mechanism is to *determine the deadline of a LC job, should it be released early*.

4.1. Early-Release: Deadline Assignments and Slack Requirements

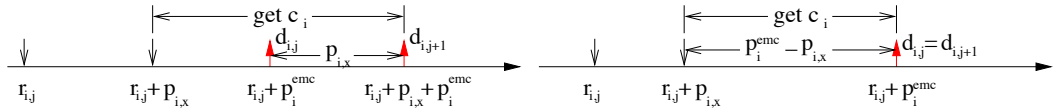
Without loss of generality, suppose that the first j jobs of a LC task τ_i have arrived regularly according to its E-MC period p_i^{emc}. The j^{th} job $J(i,j)$ arrived at time $r_{i,j}$ and has its deadline at time $d_{i,j} = r_{i,j} + p_i^{emc}$. Moreover, let's assume that $J(i,j)$ finished its execution 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. In what follows, we discuss both *conservative* and *aggressive* deadline assignments for τ_i's next job $J(i,j+1)$, should it be released early at time $r_{i,j}+p_{i,x}$.

Conservative Deadline Assignment: A simple and conservative scheme has been discussed in our preliminary study [Su and Zhu 2013]. It assigns the deadline for any newly arrived (including early-release) job according to a LC task's maximum E-MC period. That is, as illustrated in Figure 2a, if the next job $J(i,j+1)$ is released at the early-release point $r_{i,j+1} = r_{i,j} + p_{i,x}$, its deadline will be $d_{i,j+1} = r_{i,j+1} + p_i^{emc}$. The intuition behind this scheme is to have the same number of early-release points between the arrival time and deadline of any (early-release) job of a LC task. This can simplify the online steps with uniform handling of the early-release jobs for LC tasks.

Next, we determine the amount of slack needed S^{demd} for the LC task τ_i to *safely* release job $J(i,j+1)$ at its early-release point $r_{i,j+1} = r_{i,j} + p_{i,x}$. Note that job $J(i,j+1)$ would need c_i time units within its execution window $[r_{i,j+1}, d_{i,j+1}]$. From Figure 2a, we can see that $J(i,j+1)$'s execution window spans across τ_i's original deadline $d_{i,j}$, which divides the window into two intervals: $[r_{i,j+1}, d_{i,j}]$ and $[d_{i,j}, r_{i,j+1} + p_i^{emc}]$.

The first interval falls in the previous job $J(i,j)$'s execution window, where τ_i's time share within the interval was either consumed by $J(i,j)$ already or has been transformed into system slack. For the second interval, which is after τ_i's original deadline $d_{i,j}$, we can easily find that it has the length of $p_{i,x}$. According to [Brandt et al. 2003], this interval can safely contribute $p_{i,x} \cdot u_i^{min} = p_{i,x} \cdot \frac{c_i}{p_i^{emc}}$ time units for $J(i,j+1)$'s execution based on task τ_i's own minimal utilization. Therefore, the amount of slack needed to accommodate $J(i,j+1)$ to be released at time $r_{i,j} + p_{i,x}$ can be found as: $S^{demd} = c_i - p_{i,x} \cdot \frac{c_i}{p_i^{emc}}$.

Aggressive Deadline Assignment: The main idea of the *aggressive* approach is to accommodate $J(i,j+1)$'s computation demand by exploiting system slack *exclusively*



a. conservative deadline assignment b. aggressive deadline assignment

Fig. 2: Different deadline assignments for the early-release job.

without utilizing τ_i's future time share. That is, it does not allow the execution of the early-release job $J(i, j+1)$ go beyond task τ_i's original deadline $d_{i,j}$ to avoid interference with future jobs of other tasks. This makes the aggressive approach be *more general* where it can be adopted in both EDF and a preference-oriented earliest-deadline (POED) scheduler (see Section 5). Note that, the conservative approach works only with the EDF scheduler as discussed next.

Here, as illustrated in Figure 2b, the deadline of the early-released $J(i, j+1)$ is always set as $d_{i,j+1} = d_{i,j}$, regardless of when the instance is released and at which early-release point. Essentially, the aggressive approach requires the job $J(i, j+1)$ to complete its execution by task τ_i's current deadline $d_{i,j}$, should it be released early. Hence, the computation demand of $J(i, j+1)$ has to be accommodated exclusively by system slack with the amount of $S^{demd} = c_i$. However, we want to point out that, the aggressive approach can only utilize those early-release points $p_{i,x}$ that are at least c_i units away from p_i^{emc} (i.e., $p_i^{emc} - p_{i,x} \geq c_i$).

4.2. Early-Release EDF (ER-EDF) Scheduler

The same as EDF, the ER-EDF scheduler prioritizes ready jobs according to their deadlines, where jobs with earlier deadlines have higher priorities [Liu and Layland 1973]. For jobs with the same deadlines, their priorities may be arbitrarily ordered without affecting system schedulability. We consider preemptive scheduling and ER-EDF will be invoked when a job completes its execution or when a new job arrives. Note that, different from classical periodic tasks that release their jobs only at their regular periods, in the E-MC model, a LC task can have its next job instance released at one of its early-release time points once its current instance finishes the execution. Therefore, the ER-EDF scheduler needs to be invoked at appropriate early-release points of LC tasks as well. Moreover, the decision on whether to release a job early would demand judicious reclamation of system slack, as discussed below.

As the *centerpiece* of the ER-EDF scheduler, the major steps of the early-release decision algorithm are shown in Algorithm 1. It will be invoked at any *after-completion* early-release time point of a LC task. Here, for the LC task τ_i, we assume that its current job (which was released at time r_i) has finished its execution on/before $t = r_i + p_{i,x}$, an early-release time point of task τ_i. Moreover, we assume that $SlackQ(t)$ keeps track of the available system slack and $ReadyQ(t)$ holds all ready jobs at time t. As discussed in the last section, the first step is to determine the *expected* (either conservative or aggressive) deadline d_i^{exp} for task τ_i's next job, should it be released at time t (line 3). Then, the amount of required system slack S^{demd} for task τ_i to safely release its next job at time t is calculated (line 4).

As mentioned previously, in earliest-deadline based scheduling, not all available system slack may be reclaimed by task τ_i's next job. Basically, each piece of system slack has a *size* (i.e., the amount of unused execution time) and a *deadline* (which is inherited from the job whose early completion gave rise to the slack and indicates its priority). For a given job, a piece of system slack is *reclaimable* only if the slack's deadline is no later than that of the job [Zhu and Aydin 2009]. Here, for the next job of task τ_i, to find out the amount of reclaimable slack on/before its deadline d_i^{exp} in the current

Algorithm 1 : Early-Release Algorithm of ER-EDF

1: //Invoked at time $t = r_i + p_{i,x}$, only if τ_i's current job has finished its execution.
2: **Input:** t, τ_i, $p_{i,x}$, $SlackQ(t)$ and $ReadyQ(t)$;
3: Determine the expected deadline d_i^{exp} for τ_i's next job;
4: Find the amount of required slack S^{demd}; //see Section 4.1
5: **if** ($S^{demd} \leq$ ***CheckSlack***$(SlackQ(t), d_i^{exp})$) **then**
6: //Next job of τ_i can be released early at time t
7: $r_i = t$; $d_i = d_i^{exp}$; //reset release time and deadline
8: ***Enqueue***$(ReadyQ(t), J_i(c_i, d_i))$; //add the job to ready queue
9: ***ReclaimSlack***$(SlackQ(t), S^{demd})$;
10: **else**
11: //Set the next early-release time point for τ_i (if $x < k_i$);
12: ***SetTimer***$(r_i + p_{i,x+1})$;
13: **end if**

slack queue $SlackQ(t)$, a function ***CheckSlack***$(SQ(t), d_i^{exp})$ is used (line 5), which will be discussed in more detail in the next section.

If the amount of reclaimable slack turns out to be no less than the required slack S^{demd} for task τ_i to safely release its next job at time t, the job will be released and inserted into the ready job queue $ReadyQ$ (lines 7 and 8). Then, the reclaimed S^{demd} units of system slack is removed from the slack queue $SlackQ(t)$ with the helper function ***ReclaimSlack()*** (line 9; see next section for details). Otherwise, if there is not enough reclaimable slack, task τ_i cannot release its next job at its current early-release time point t. In case task τ_i has more early-release time points, a new timer is set to invoke the scheduler at its next early-release point (line 12).

Clearly, the number of early-release points for LC tasks has a great impact on the scheduling overhead as well as the potential execution improvement for LC tasks. As shown in our evaluation results (see Section 6), having a few (e.g., 4 to 6) such early-release points for LC tasks is effective enough to improve their execution frequencies with reasonable scheduling overheads.

4.3. Slack Management with Wrapper-Tasks

Wrapper-task has been studied as an efficient mechanism to manage slack under EDF scheduling [Zhu and Aydin 2009]. In this work, we *extend* and exploit the wrapper-task technique to safely provide the needed slack and enable LC tasks release their jobs at their early-release time points. As there is only one active job for any task at a time, in what follows, we use task τ_i to represent its current job when there is no ambiguity.

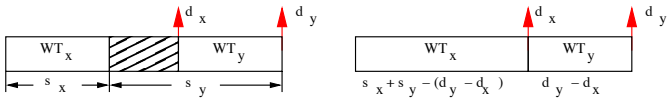
Essentially, a wrapper-task (***WT***) represents a piece of system slack with two parameters (s, d), where s denotes the slack size and d is the deadline that equals to that of the task giving rise to this slack [Zhu and Aydin 2009]. For instance, when a HC task τ_j completes its execution early at runtime with execution time c_j that is no more than the task's high-level WCETs c_j^{high} (i.e., $c_j \leq c_j^{high}$), the over-provisioned time will be converted to be slack ($c_j^{high} - c_j$, d_j), where d_j is the task's current deadline.

All wrapper-tasks are kept in a *slack queue (**SlackQ**)*. With EDF scheduling being considered, similar to ready tasks whose deadlines represent their priorities, the priority of a wrapper-task is also determined by its deadline. Therefore, in ***SlackQ***, wrapper tasks are kept in the increasing order of their deadlines. In addition, we assume that all ready tasks are in a *ready queue (**ReadyQ**)*, which are also in the increasing order of their deadlines (i.e., tasks with smaller deadlines are in the front of ***ReadyQ***).

At runtime, wrapper-tasks compete for the processor with ready tasks. Therefore, when both *SlackQ* and *ReadyQ* are *not empty* and the header wrapper-task WT_h in *SlackQ* has an earlier deadline than that of the *ReadyQ*'s header task τ_h, WT_h will get the processor. In this case, it 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 [Zhu and Aydin 2009]:

— *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 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.

Slack Push-Back: Note that, when slack is generated due to the early completion of a task, it inherits the deadline of the task and thus has the highest priority. Therefore, such slack will most likely wrap the execution of other ready tasks and be pushed forward (with a later deadline) [Zhu and Aydin 2009]. However, from previous discussion, we know that LC tasks prefer slack with earlier deadlines so that they can obtain more reclaimable slack for their early releases. Next, we show how slack can be pushed *backward* (to have an earlier deadline) and become more reclaimable.



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

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

Suppose that, at time t, 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 at time t 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 (i.e., scheduled) after time d_x. That is, for part of slack represented by WT_y (dashed part in Figure 3a with the size of $s_y - (d_y - d_x)$ units), we can safely push it *backward* and make that part to have 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 $s_x + s_y - (d_y - d_x)$. If there are wrapper-tasks with deadlines earlier than d_x, such slack push-back can be performed iteratively. As shown in our evaluation results (see Section 6), such slack push-back does provide more reclaimable slack for LC tasks at a given time and thus improve their execution frequencies.

With slack push-back being considered, Algorithm 2 details the steps for the function *CheckSlack(Q, d)*. Here, to get more reclaimable slack before time d, all slack is first pushed backward (if possible) (lines 3 to 8). Then, the amount of reclaimable

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

1: **Input:** Slack queue with m wrapper-tasks $Q = \{WT_1, \cdots, WT_m\}$ and time d;
2: **Output:** the amount of reclaimable slack $S^r(d)$;
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: $k = 1$; $S^r(d) = 0$; //initialize the amount of reclaimable slack
10: **while** ($k \leq m$ AND $d_k \leq d$) **do**
11: $S^r(d) += s_k$; $k++$; //accumulate reclaimable slack
12: **end while**
13: **if** ($k \leq m$ AND $s_k > d_k - d$) **then**
14: $S^r(d) += s_k - (d_k - d)$; //part of WT_k is reclaimable
15: **end if**

slack, including all completely reclaimable wrapper-tasks (lines 10 to 12) as well as the last partially reclaimable wrapper-task (lines 13 to 15), is accumulated. The function *ReclaimSlack(Q, s)* has similar steps and its algorithm is omitted for brevity. Interested readers can refer to [Zhu and Aydin 2009] for more details.

4.4. Analysis of ER-EDF

From Algorithm 1, we can see that the wrapper-task based slack management plays a critical rule in the ER-EDF scheduler. Therefore, we start the analysis with the review of two important results regarding to slack management from our existing work [Zhu and Aydin 2009]. Then, we show that the newly added rule on slack push-back will not cause any deadline miss either. A real-time job is denoted below as $J(r, c, d)$, where r is its arrival time, c the worst case execution time, and d the deadline.

LEMMA 4.1. **(Lemma 1 in [Zhu and Aydin 2009])** *Consider a set of real-time jobs which can be scheduled by preemptive EDF in a feasible manner. If we replace a job $J_a(r_a, c_a, d_a)$ in the set by two jobs $J_x(r_a, c_x, d_x)$ and $J_y(r_a, c_y, d_y)$, where $c_x + c_y \leq c_a$ and $d_a \leq d_x \leq d_y$, the revised set of jobs is still feasible.*

LEMMA 4.2. **(Theorem 3 in [Zhu and Aydin 2009])** *At any time t, if every wrapper-task is converted to a ready real-time job with the same timing parameters, the augmented set of ready jobs can still be scheduled by EDF in a feasible manner.*

Essentially, the above two lemmas state that slack management (including *slack push-forward* and *slack reclamation*) with the wrapper-task technique does not introduce additional workload to the system and thus no real-time job will miss its deadline [Zhu and Aydin 2009].

For the *slack push-back* rule introduced in this work, it will not cause any deadline miss either. This is formally represented in the following lemma, where the proof follows the similar reasoning as that of Lemma 4.1 based on the concept of *processor demand* [Baruah et al. 1990] and is omitted for brevity.

LEMMA 4.3. *Suppose that a set Φ of real-time jobs which can be feasibly scheduled by preemptive EDF has two jobs $J_a(r_a, c_a, d_a)$ and $J_b(r_b, c_b, d_b)$, where $r_a = r_b$, $d_a < d_b$ and $c_b > d_b - d_a$. A new set Φ^{new} of jobs is also feasible if the two jobs are replaced by $J_a^{new}(r_a, c_a + c_b - (d_b - d_a), d_a)$ and $J_b^{new}(r_b, d_b - d_a, d_b)$, respectively.*

When there is no early-release of LC tasks at runtime (due to, for example, insufficient system slack), the executions of tasks under ER-EDF will be the same as that under EDF. Define the *canonical execution* of tasks under ER-EDF as the one when all tasks take their WCETs (i.e., HC tasks take their high-level WCETs while LC tasks take their WCETs) and all LC tasks are released according to their maximum periods (i.e., there is no early-release) at runtime. From Section 4.3, we know that the wrapper-task mechanism does not affect tasks' executions when there is no slack reclamation [Zhu and Aydin 2009]. Therefore, we can easily obtain the following lemma.

LEMMA 4.4. *For a set of tasks that are E-MC schedulable under EDF, there is no deadline miss in the canonical execution of the tasks under ER-EDF.*

In what follows, assuming that a set of E-MC tasks is schedulable under ER-EDF in the canonical case, we prove that the early-releases of LC tasks at runtime will not result in any deadline miss. Essentially, we show that the early-release jobs of LC tasks under ER-EDF will not introduce more workload into the system than that of the canonical case for both aggressive and conservative deadline assignments.

4.4.1. ER-EDF with Aggressive Deadline Assignment. First, for the *aggressive* deadline assignment, recall that *only* system slack is exploited to accommodate the computation demand of an early-release job. Based on Lemma 4.1 to Lemma 4.3 related to the wrapper-task based slack management, we know that no extra workload will be introduced into the system when the reclaimable system slack is exploited to support the additional execution demand of a task [Zhu and Aydin 2009]. Therefore, as long as the amount of reclaimable system slack before a LC task's current deadline is no less than its WCET, the early release of a new job for the LC task with aggressive deadline assignment through appropriate slack reclamation will not cause any deadline miss. More formally, we have the following lemma.

LEMMA 4.5. *Suppose that the current job of a LC task τ_i completes its execution no later than time t, which is one of task τ_i's early-release time points. If the amount of reclaimable system slack $S^r(d_i)$ at time t (where d_i is τ_i's current deadline) is no less than its WCET c_i, the early release of a new job with the deadline of d_i for task τ_i at time t by reclaiming c_i units of system slack will not lead to any deadline miss.*

Based on the above lemma, we can immediately obtain the following theorem regarding to the correctness of ER-EDF with aggressive deadline assignment.

THEOREM 4.6 (**ER-EDF with Aggressive Deadline Assignment**). *For an E-MC task set with $U(H,H)+U(L,min) \leq 1$, there is no deadline miss at runtime when the task set is scheduled under ER-EDF and the aggressive approach is adopted to assign deadlines for the early-release jobs of LC tasks.*

4.4.2. ER-EDF with Conservative Deadline Assignment. In comparison, with *conservative* deadline assignment, certain *future* time share of a LC task needs to be exploited to accommodate the computation needs of its early-release jobs in addition to system slack. Next, we show that such exploitation of future time share of a LC task is safe and will not cause any deadline miss as well. The analysis relies on another existing result for EDF-based scheduling: the flexible *dynamic deadline* technique [Brandt et al. 2003], as given by the following lemma.

LEMMA 4.7. **(Lemma 1 in [Brandt et al. 2003])** *Given a feasible EDF schedule, if a task leaves at its deadline (where it has zero lag), a new task with the same utilization can enter the system at that time and the resulting EDF schedule remains feasible.*

From Section 4.1, we know that a LC task τ_i can contribute $p_{i,x} \cdot \frac{c_i}{p_i^{emc}}$ units of its future time share for its new job $J_{i,j+1}$, should it be released at τ_i's early-release time point $r_{i,j} + p_{i,x}$, with the conservative deadline assignment. Essentially, this is equivalent to τ_i leaving at its deadline d_i while a new task with the same utilization but a different period $p_{i,x}$ arriving at the same time. Thus, according to Lemma 4.7, if the execution of these time units for $J_{i,j+1}$ occurs between the task τ_i's current deadline d_i and its future deadline $d_i + p_{i,x}$ (see Figure 2a), there will be no deadline miss.

However, once job $J_{i,j+1}$ is released, the execution of its units (including τ_i's future time share and the reclaimed system slack) under ER-EDF can span anywhere between its release time and its deadline. That is, it is possible that task τ_i's future time share $p_{i,x} \cdot \frac{c_i}{p_i^{emc}}$ may appear before τ_i's deadline d_i in the resulting ER-EDF schedule, which violates the condition of Lemma 4.7. In what follows, we show that such early appearance of τ_i's future time share in the ER-EDF schedule is still safe and will not cause any deadline miss. Specifically, we have the following lemma.

LEMMA 4.8. *Consider a set Φ of real-time jobs that can be feasibly scheduled by preemptive EDF. If we replace two jobs $J_a(r_a, c_a, d)$ and $J_b(r_b, c_b, d)$ with a new job $J_k(r_k, c_k, d)$, where $r_k = r_a < r_b$ and $c_k = c_a + c_b$, the revised job set Φ^{new} is still feasible.*

PROOF. Note that, under EDF scheduling, for jobs with the same deadline, their priorities can be arbitrarily assigned without affecting their schedulability. Without loss of generality, assume that jobs J_a, J_b and J_k have the lowest priority for all jobs in Φ that have the deadline of d. Thus, the job replacement will not affect the executions of other jobs in Φ with deadlines earlier than or at d. Moreover, for the jobs with deadline after d, if such jobs are released at or after r_b, their executions will not be affected by the job replacement either. Therefore, only for the jobs that are released before r_b and have deadlines after d, may their executions be affected by the job replacement.

Suppose that there is a job $J_x \in \Phi$ where $r_x < r_b$ and $d_x > d$. In the original EDF schedule, if J_x is not executed within the interval $[r_a, r_b]$ (i.e., its execution occurs either before r_a or after r_b), its execution will not be affected by the job replacement as well. Now, suppose that J_x is executed within the interval $[r_a, r_b]$ for x time units in the original EDF schedule (which must happen after J_a's execution of c_a units). After the job replacement, the new job J_k needs $c_a + c_b$ time units and thus will occupy J_x's execution time within the interval $[r_a, r_b]$ for part (if $x > c_b$) or all its units (if $x \leq c_b$). However, the exact same time units after r_b from the execution of J_b can be returned back to J_x. Therefore, J_x will be executed for the same time units before d after the job replacement, which means J_x can finish its execution before its deadline. Hence, no job will miss its deadline after the job replacement. □

Hence, we can get the following lemma and theorem regarding to the correctness of ER-EDF with conservative deadline assignment.

LEMMA 4.9. *Suppose that the current job of a LC task τ_i completes its execution no later than time $t = r_{i,j} + p_{i,x}$, which is one of task τ_i's early-release time points. If the amount of reclaimable system slack $S^r(d_i)$ before $d_i + p_{i,x}$ (where d_i is τ_i's current deadline) is no less than $c_i - p_{i,x} \cdot \frac{c_i}{p_i^{emc}}$, the early release of a new job for task τ_i at time t by assigning its deadline as $d_i + p_{i,x}$ and reclaiming $c_i - p_{i,x} \cdot \frac{c_i}{p_i^{emc}}$ units of system slack will not lead to any deadline miss under the ER-EDF scheduler.*

THEOREM 4.10 (**ER-EDF with Conservative Deadline Assignment**). *For an E-MC task set with $U(H,H) + U(L,min) \leq 1$, there is no deadline miss at runtime when the task set is scheduled under ER-EDF and the conservative approach is adopted to assign deadlines for the early-release jobs of LC tasks.*

5. EARLY-RELEASE FOR A PREFERENCE-ORIENTED EDF SCHEDULER

The early-release mechanism discussed in the last section aims at improving the execution frequencies of LC tasks, which turns to be quite effective as shown in our evaluation results (see Section 6). With the increased execution frequencies, the control performance for those LC tasks can be significantly improved. However, for HC tasks, their execution frequencies (i.e., periods) are fixed. Therefore, to improve their control performance, we need to consider other factors, such as delay (i.e., response time) and jitters [Baruah et al. 1999; Enrico and Cervin 2008; Balbastre et al. 2004].

To improve their control performance, we can execute HC tasks at earlier times to reduce their response times (delay). As the probability for HC tasks taking more time than their low-level WCETs is rather low at runtime [Baruah et al. 2012], such early executions of HC tasks make system slack available at an earlier time, which can be better exploited by LC tasks. In order to execute HC tasks early at runtime, different from EDF-VD that assigns smaller (virtual) deadlines for such tasks (with the objective to improve system schedulability) [Baruah et al. 2012], we adopt a recently studied Preference-Oriented Earliest-Deadline (POED) scheduler [Guo et al. 2015], which is first reviewed in the next section. Then, the corresponding early-release scheme under the POED scheduler is investigated in Section 5.2.

5.1. Preference-Oriented Earliest-Deadline (POED) Scheduler

Basically, under the POED scheduler, real-time tasks are divided into two different groups: the tasks that prefer to be executed *as soon as possible* (denoted as ASAP tasks), and those that prefer to be executed *as late as possible* (denoted as ALAP tasks) [Guo et al. 2015]. In this work, we consider HC tasks as ASAP tasks while LC tasks as ALAP tasks. One of the scheduling principles of POED is to execute ASAP (HC) tasks before the execution of ALAP (LC) tasks whenever it is possible to do so. For such a purpose, the POED scheduler puts ASAP (HC) tasks at the center stage and, whenever possible, gives them higher priorities when making scheduling decisions to execute them early. That is, at runtime, the priorities of ready/active tasks depend on not only their deadlines but also their execution preferences.

More specifically, when a ready ASAP (HC) task has the earliest deadline, it will be executed immediately, the same as in the traditional EDF scheduler. However, when a ready ALAP (LC) task has the earliest deadline, its execution will depend on whether there are ready ASAP (HC) tasks or not. If there exists a ready ASAP (HC) task that has a later deadline, instead of executing the earliest deadline ALAP (LC) task immediately, the POED scheduler will check to see if it is possible to delay its execution by looking ahead the (current and future) tasks within the interval between the current time and the ALAP (LC) task's deadline. If it is safe to delay the execution of the ALAP (LC) task, POED will execute the ASAP (HC) task first even though it has a later deadline in order to better fulfill the execution preferences of both tasks. When there is no ready ASAP (HC) task, the ALAP (LC) task can be executed immediately. With appropriate processing of the tasks within the look-ahead interval, we have shown that the twisted executions of tasks with different preferences under the POED scheduler will not cause any deadline miss [Guo et al. 2015].

5.2. Early-Release POED Scheduler

Note that, the look-ahead approach in the POED scheduler assumes that future tasks arrive regularly according to their pre-defined periods [Guo et al. 2015]. However, such an assumption conflicts directly with the flexible dynamic deadline technique that assumes tasks may change their periods freely at their deadlines [Brandt et al. 2003]. Therefore, the conservative deadline assignment that exploits future task share with

Algorithm 3 : *The ER-POED Scheduler*

```
1: Input: invocation time t; ReadyQ_HI(t), ReadyQ_LO(t) and SlackQ(t).
2: if (invoked by an Early-Release timer) then
3:   Suppose t = r_i + p_{i,x} is an early-release time point of a completed LC task τ_i
4:   if (CheckSlack(SlackQ(t), d_i) ≥ c_i) then
5:     ReclaimSlack(SlackQ(t), c_i);
6:     Enqueue(ReadyQ_LO(t), J_i(c_i, d_i) ); //add the job to ready queue
7:   else
8:     SetTimer(r_i + p_{i,x+1}); //set the next ER time point if (x < k_i)
9:   end if
10: else if (invoked by other scheduling events) then
11:   if (ReadyQ_HI(t)== ∅ OR ReadyQ_LO(t)== ∅) then
12:     if (ReadyQ_HI(t)== ∅ AND ReadyQ_LO(t)== ∅) then
13:       CPU is idle and corresponding slack in SlackQ(t) is consumed;
14:     else
15:       if (ReadyQ_HI(t) != ∅) then
16:         J_k = Dequeue(ReadyQ_HI(t)); //J_k has the earliest deadline
17:       else
18:         J_k = Dequeue(ReadyQ_LO(t)); //J_k has the earliest deadline
19:       end if
20:       Execute J_k; //slack wrapped-execution may occur
21:     end if
22:   else
23:     //Suppose that J_h = Header(ReadyQ_HI(t)) and J_l = Header(ReadyQ_LO(t));
24:     if (d_h ≤ d_l) then
25:       J_h = Dequeue(ReadyQ_HI(t));
26:       Execute J_h; //slack wrapped-execution may occur
27:     else
28:       w = CheckLookAheadInterval(d_l);
29:       if (w > 0) then
30:         J_h = Dequeue(ReadyQ_HI(t));
31:         Execute J_h for w units; //slack wrapped-execution may occur
32:       else
33:         J_l = Dequeue(ReadyQ_LO(t));
34:         Execute J_l; //slack wrapped-execution may occur
35:       end if
36:     end if
37:   end if
38: end if
```

the flexible dynamic deadline technique **cannot** be applied to the POED scheduler. Hence, in this work we consider only the aggressive deadline assignment for the early-released jobs of LC tasks under the POED scheduler.

Algorithm 3 outlines the major steps of the ER-POED scheduler. Here, we use two queues $ReadyQ_{HI}(t)$ and $ReadyQ_{LO}(t)$ to keep track of ready HC and LC tasks, respectively. Moreover, the same as in ER-EDF, $SlackQ(t)$ is used to keep the available system slack. In addition to the normal scheduling events (such as job arrivals and completion), ER-POED may also be invoked at any early-release time point of a completed LC task. If ER-POED is invoked by an early-release timer of a LC task τ_i, the amount of reclaimable slack for τ_i will be checked (line 4). Note that, with the aggressive deadline assignment, an early-release job for task τ_i requires c_i units of system

slack. If there is enough slack, by reclaiming c_i units of system slack, a new job for task τ_i will be released and added to $ReadyQ_{LO}(t)$ accordingly (lines 5 and 6). Otherwise, a timer is set for the next early-release time point of τ_i (line 8).

In case that ER-POED is invoked by other scheduling events, a ready job (if exists) needs to be selected for execution. If there is no ready job (i.e., both $ReadyQ_{HI}(t)$ and $ReadyQ_{LO}(t)$ are empty) for execution, CPU is idle and corresponding system slack in $SlackQ(t)$ (if any) is consumed. Otherwise, if all ready jobs are either HC jobs ($ReadyQ_{LO}(t)$ is empty) or LC jobs ($ReadyQ_{HI}(t)$ is empty), the header job with the earliest deadline of the corresponding queue will be selected for execution (lines 15 to 20). Note that, if there exists a piece of slack in $SlackQ(t)$ that has an earlier deadline, wrapped-execution will occur [Zhu and Aydin 2009], which is not shown in the algorithm for simplicity.

When there are both HC and LC ready jobs (that is, both $ReadyQ_{HI}(t)$ and $ReadyQ_{LO}(t)$ are not empty), the header HC job that is considered as ASAP tasks will be executed as normal if it has the earliest deadline (lines 24 to 26). However, when the header LC job has the earliest deadline, the look-ahead interval between the current time t and the header LC job's deadline will be checked and see if it is possible to execute the HC job first (line 28). Here, for brevity, the details of the function to check look-ahead interval are not shown and can be found in [Guo et al. 2015]. The basic idea is to consider all (current and future) jobs that have deadlines no later than d_l and map their executions within the look-ahead interval. If there is empty time slot at the beginning of the look-ahead interval after mapping those jobs, the HC job can be executed during the empty time slot (lines 30 and 31). Otherwise, the LC job has to be executed immediately to avoid missing its deadline (line 34).

Note that, the same as in the POED scheduler [Guo et al. 2015], the early execution of HC (ASAP) tasks under ER-POED happens only when such execution can be safely performed. Moreover, with the aggressive deadline assignment, only system slack is exploited to accommodate the computation requirement of an early-release job of a LC task under ER-POED, which will not disturb the executions of (current and future) jobs within the look-ahead intervals. Therefore, following the same reasoning as in Section 4 and based on the results in [Guo et al. 2015], we can get that the early-released jobs under ER-POED will not cause any deadline miss since slack reclamation with the wrapper-task based slack management does not introduce additional workload to the system [Zhu and Aydin 2009].

5.3. An Example Execution of ER-POED

For the example task set shown in Table I, Figure 1(d) illustrates the ER-POED schedule within the interval $[0, 30)$. Here, for the HC task τ_1 that is executed ASAP in the ER-POED schedule, its jobs $J(1, 1)$ and $J(1, 2)$ finish their executions at times 6 and 29, respectively. In comparison, in the ER-EDF schedule as shown in Figure 1(c), these jobs finish at times 8 and 30, respectively. Therefore, the execution of HC tasks can be performed at earlier times in the ER-POED schedule, which can reduce the response times (delay) for such tasks and thus improve their control performance.

Moreover, considering the LC task τ_3, it releases four task instances within the interval in the ER-POED schedule, which is the same as that in the ER-EDF schedule. However, its jobs $J(3, 2)$ and $J(3, 4)$ are released early with deadlines of 16 and 32 in the ER-POED schedule (with aggressive deadline assignment), respectively. In comparison, these two jobs are assigned deadlines of 24 and 40 in the ER-EDF schedule that adopts the conservative deadline assignment. Therefore, by executing HC tasks at earlier times, ER-POED not only improves the control performance of HC tasks but also enables LC tasks more efficiently exploit system slack, which can potentially improve LC tasks' control performance as well.

6. EVALUATIONS AND DISCUSSIONS

We evaluate the effectiveness of our proposed E-MC model and early-release schedulers on improving execution frequencies for LC tasks and reducing response times and jitters for HC tasks through extensive simulations. In addition to the ER-EDF and ER-POED schedulers, for comparison, we implemented two state-of-the-art MC schedulers: EDF-VD [Baruah et al. 2012] and adaptive EDF-VD [Huang et al. 2014].

Here, the EDF-VD scheduler utilizes a virtual deadline factor x for HC tasks to improve system schedulability. In the adaptive EDF-VD scheduler, for a given virtual deadline factor x, a scaling factor y is assigned for LC tasks to determine their high-level periods (and thus degraded service levels). When the system enters the high-level execution mode due to any HC task running for more time than its low-level WCET, EDF-VD discards all currently active (and future arriving) LC tasks [Baruah et al. 2012] while the adaptive EDF-VD allows LC tasks execute according to their high-level periods (which is y times of their MC periods) [Huang et al. 2014]. It is assumed that the system returns back to the normal (low-level) execution mode only when the system becomes idle (i.e., when there is no ready task in the queue).

Both EDF-VD schedulers adopts the conventional MC task models. For fair comparison, as described below, we obtain E-MC tasks from MC tasks by assigning the E-MC periods and early-release points for LC tasks while keeping HC tasks and the WCETs of LC tasks unchanged. That is, under the condition that all schedulers guarantee the same executions of HC tasks, we evaluate how our proposed schedulers can improve the execution performance of both LC and HC tasks.

6.1. Experiment Settings

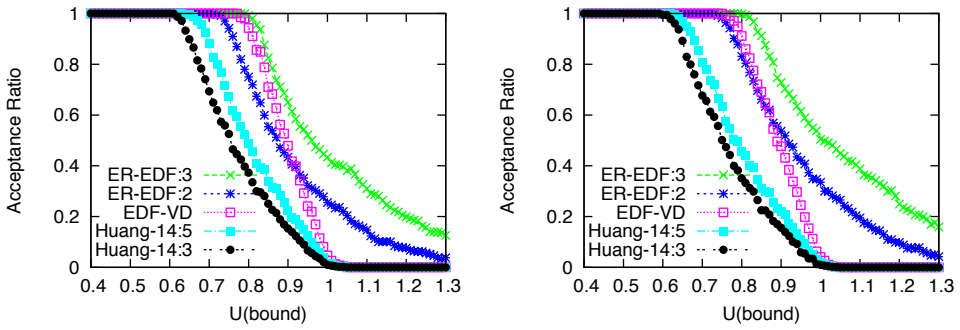
As it is difficult to obtain the mixed-criticality applications in real systems, we evaluate our algorithms using synthetic tasks, which are generated based on a similar methodology as adopted in [Guan et al. 2011; Li and Baruah 2012]. Here, we first describe the parameters (and their ranges) used in the task generation process. The parameter $Prob(HI)$ denotes the probability of a generated task τ_i being a HC task, which is used to control the workload mixture of HC/LC tasks in the system. A larger $Prob(HI)$ indicates that there is more HC tasks in the system and vice versa.

The periods of tasks are generated uniformly within the range of $[50, 200]$ (time units). The utilization of a task τ_i is uniformly generated within the range $[0.05, 0.15]$. If the task τ_i is a LC task, the number represents its normal utilization u_i. Otherwise, when τ_i is a HC task, the number is τ_i's high-level utilization u_i^{high}. The low-level utilizations u_i^{low} of a HC task τ_i is further obtained from a high-to-low execution ratio Z, which is uniformly generated within a given range. The WCETs of a task are obtained from its period and utilizations accordingly. The number of tasks in a given task set and the system load are controlled by a utilization bound U_{bound}, which is the larger of high-criticality utilization $U(H, H)$ of HC tasks and low-criticality utilization $U(H, L) + U(L, L)$ of all tasks [Baruah et al. 2012]. For a task set with a given setting of U_{bound}, tasks are repeatedly generated and added to the task set until U_{bound} is met.

Once a MC task set is obtained, we derive the corresponding E-MC task set by extending LC tasks' desired periods with a factor η to get their E-MC periods. Finally, k early release points of LC tasks are generated within their E-MC periods.

6.2. Minimum Services (E-MC Periods) of LC Tasks and Acceptance Ratio

First, we evaluate how E-MC periods of LC tasks can be exploited to enhance the schedulability of a task system. 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 con-



a. $Z \in [2, 10]$ b. $Z \in [1, 8]$;

Fig. 4: Acceptance ratio of schedulable task sets; $Prob(HI) = 0.5$;

sider balanced HC/LC workload and set $Prob(HI) = 0.5$. For each data point, 1000 task sets are generated. The results for EDF-VD are in line with what has been reported in [Baruah et al. 2012], where more task sets can be schedulable when the high-to-low execution ratio for HC tasks is relatively larger.

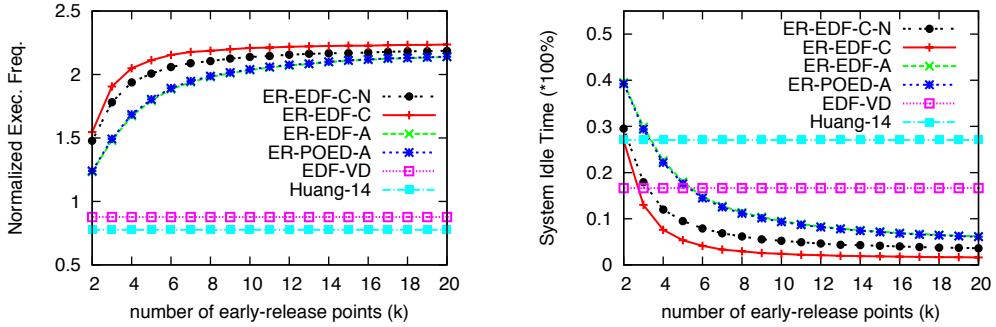
The acceptance ratio of the transformed E-MC task sets (obtained by having the E-MC period for a LC task τ_i as η times of its desired period, $p_i^{emc} = \eta \cdot p_i$) under ER-EDF is denoted by ***ER-EDF:***η. Here, we can see that, if the periods of LC tasks can be extended to 3 times, the number of schedulable task sets under ER-EDF becomes even more than that of EDF-VD. That is, compared to that of MC task sets under EDF-VD, by moderately extending LC tasks' maximum E-MC periods as three times of their desired MC periods (i.e., $\eta = 3$), significantly more task sets become schedulable under ER-EDF (especially for higher system loads).

The acceptance ratio under the adaptive EDF-VD [Huang et al. 2014] is shown by bounding the scaling factor y to be 3 or 5 (which is denoted as *Huang-14:3* and *Huang-14:5*, respectively). Here, for a given MC task set, if the resulting scaling factor for LC tasks' periods is no more than the bound, the task set is considered as schedulable, and vice versa. Clearly, to accommodate the executions of LC tasks in the HI running mode, the acceptance ratio of adaptive EDF-VD (even with the bound of y being 5) is worse than that of EDF-VD. However, better acceptance ratio is obtained with higher bound on the scaling factor. In the remaining simulations, we set y's bound to be 5.

Next, we evaluate the effects of runtime behaviors of tasks on their control performance. Here, we introduce another parameter, $Prob(c^{low})$, which indicates the probability of a HC task τ_i taking its low-level WCET c_i^{low} at runtime. Moreover, for each task set, we simulate the executions of its tasks for the first 1,000,000 time units.

6.3. Effects of Early-Release Points

In this section, we first evaluate how the number of early-release points for LC tasks can affect the control performance of both LC and HC tasks. Here, the task sets are generated according to the following parameters: $U_{bound} = 0.9$, $Prob(HI) = 0.5$, $Z \in [1, 8]$ and $\eta = 2$ (i.e., $p_i^{emc} = 2 \cdot p_i$ for LC tasks). The early-release points of a LC task τ_i are assumed to uniformly distribute within $[c_i, p_i^{emc}]$. Considering the fact that the probability of HC tasks taking more time than their low-level WCETs is rather low, it is assumed that $Prob(c^{low}) = 0.9$ for HC tasks (i.e., 90% of HC task instances take their low-level WCETs at runtime). Here, each data point represents the average result of 100 schedulable task sets.



a. Normalized execution frequencies for LC tasks; b. System idle time;

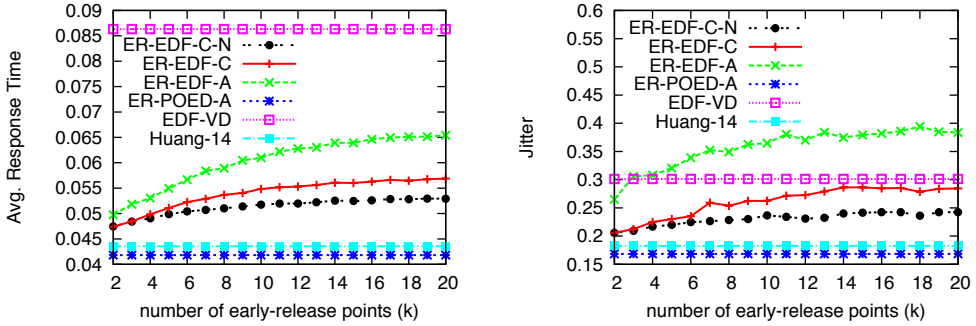
Fig. 5: Effects of ER points on LC tasks; $Prob(HI) = 0.5$, $U_{bound} = 0.9$, $Prob(c^{low}) = 0.9$.

First, Figure 5a shows how the number of early-release points of LC tasks affects their execution frequencies. Here, the Y-axis is the normalized execution frequencies for LC tasks with the one corresponding to their desired (MC) periods being the baseline. Note that, the executions of LC tasks are independent of their early-release points under both EDF-VD schedulers. For EDF-VD, LC tasks can be cancelled when the system enters the HI execution mode due to some HC tasks running for more time than their low-level WCETs, which leads to the normalized execution frequencies for LC tasks being less than 1. However, it is interesting to see that, for adaptive EDF-VD (denoted as "Huang-14"), the resulting normalized execution frequency for LC tasks is even worse than that of EDF-VD. The reason is that, although LC tasks are allowed to be executed according to $y*p_i$ under the adaptive EDF-VD scheduler in the HI running mode, these additional LC tasks significantly extend the time intervals for the system being operated under the HI running mode. This results in extended periods of time where LC tasks are executed according to $y * p_i$ (where y can be as large as 5), which turns out to be even worse than the case of EDF-VD where the system returns to the LO running mode sooner and executes LC tasks according to their p_i.

Not surprisingly, for our proposed ER-EDF and ER-POED schedulers, the normalized execution frequencies for LC tasks can be as high as more than 2 (where the conservative and aggressive deadline assignment schemes are denoted as "*-C" and "*-A", respectively). Clearly, having more early-release points results in increased execution frequencies for LC tasks as there are more opportunities for those tasks to release their jobs at earlier times. Having a few (e.g., 4 to 6) early-release points is sufficient for ER-EDF and ER-POED to obtain much better execution frequencies for LC tasks when compared to that of EDF-VD. Moreover, as the conservative deadline assignment enables LC tasks use their future time shares and thus needs less system slack for LC tasks to release their jobs early, ER-EDF with conservative approach achieves better execution frequencies for LC tasks when compared to that of the aggressive approach. As expected, by disabling slack push-back for ER-EDF with conservative approach (denoted as "ER-EDF-C-N"), the resulting execution frequencies for LC tasks become slightly worse. For ER-POED, it aims at executing HC tasks early and allowing them to release system slack at earlier times. Hence, it provides better opportunities for LC tasks to reclaim such slack and thus leads to slightly higher execution frequencies when compared to that of ER-EDF with the aggressive deadline assignment approach.

Recall that not all available system slack may be reclaimable by a task and a LC task can only utilize its reclaimable system slack at their early-release time points. There-

fore, it is inevitable for the system to be idle with its unusable slack. Figure 5b shows the percentage of system operation time being idle, where less idle time indicates that more system slack has been utilized. Clearly, having more early-release points for LC tasks leads to less system idle time under the proposed early-release schedulers. However, when there are only a few early-release points (i.e., $k \leq 4$), it turns out that it is extremely difficult for LC tasks with *large* WCETs to take advantage of their early-release points where the early-released jobs of LC tasks with *small* WCETs do not consume much system slack. Therefore, it is interesting to see that there can be more idle time under our proposed ER schedulers even though the (average) normalized execution frequencies for LC tasks are higher than those of the EDF-VD schedulers.



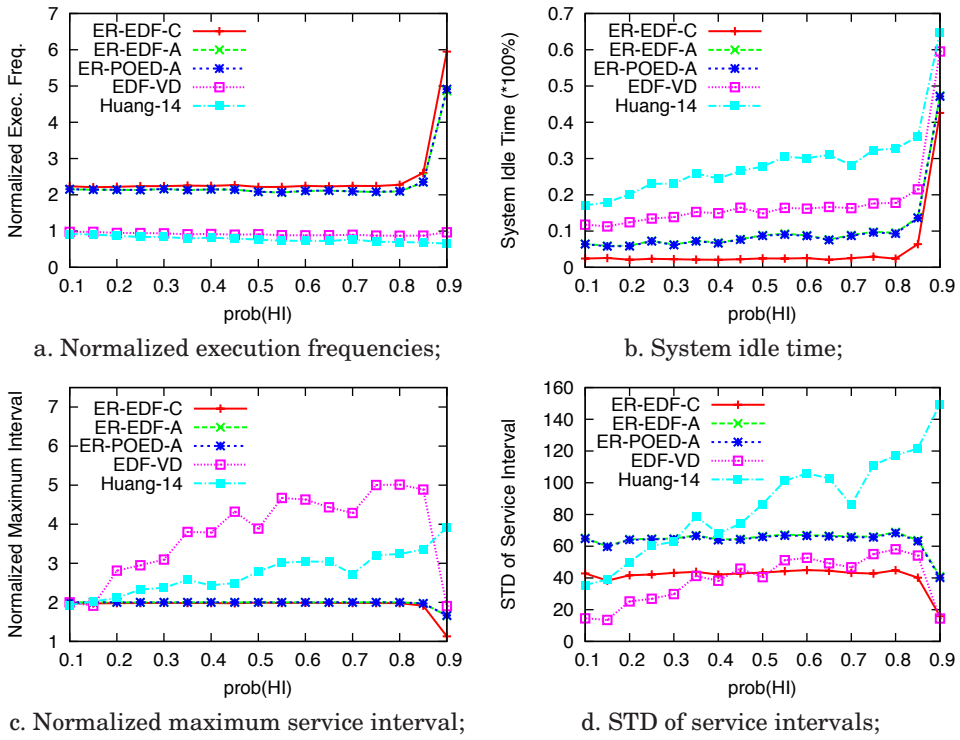
a. Average response time (delay) for HC tasks; b. Jitters for HC tasks;

Fig. 6: Effects of ER points on HC tasks; $Prob(HI) = 0.5$, $U_{bound} = 0.9$, $Prob(c^{low}) = 0.9$.

Next, we evaluate the effects of early-release points on HC tasks by focusing on their *delay* (i.e., average response time) and *jitter* [Enrico and Cervin 2008; Balbastre et al. 2004]. In the evaluation, we use the *normalized response time*, which is defined as the ratio of a task's *response time* (the difference between its completion and arrival times) over its period. Similarly, the jitter of a task is defined as the ratio of the difference between the longest and shortest response times of all its jobs over its period.

Figure 6a shows the normalized average response times (i.e., delay) of HC tasks. First, under ER-EDF, the deadlines of LC tasks depend on their maximum E-MC periods (i.e., $2 * p_i$), which lead to less interference on the executions of HC tasks compared to that under EDF-VD. Hence, the resulting response times for HC tasks under ER-EDF are better than that under EDF-VD. With more early-release points, LC tasks have more chances to release its jobs and increase the interference for HC tasks' executions, which results in increased response times for HC tasks. Due to the extended system operation time in the HI running mode and rather large high-level periods $y * p_i$ for LC tasks (where y can be as large as 5), the overall interference on HC tasks' executions turns to be much less under adaptive EDF-VD, where the resulting response times for HC tasks are much smaller than both EDF-VD and ER-EDF.

The response times for HC tasks are significantly smaller under ER-POED due to their ASAP execution preference, which enables them be executed at earlier times whenever possible. Here, the delay of HC tasks under ER-POED remains almost the same as the number of early-release points of LC tasks increases. The reason is that, LC tasks are treated as ALAP tasks under ER-POED and their early release-release jobs have little effect on the execution of HC tasks. The jitters of HC tasks follow the similar patterns, which are further shown in Figure 6b.



a. Normalized execution frequencies; b. System idle time;

c. Normalized maximum service interval; d. STD of service intervals;

Fig. 7: Effects of $Prob(HI)$ on LC tasks; $U_{bound} = 0.9$, $Prob(c^{low}) = 0.9$ and $k = 10$.

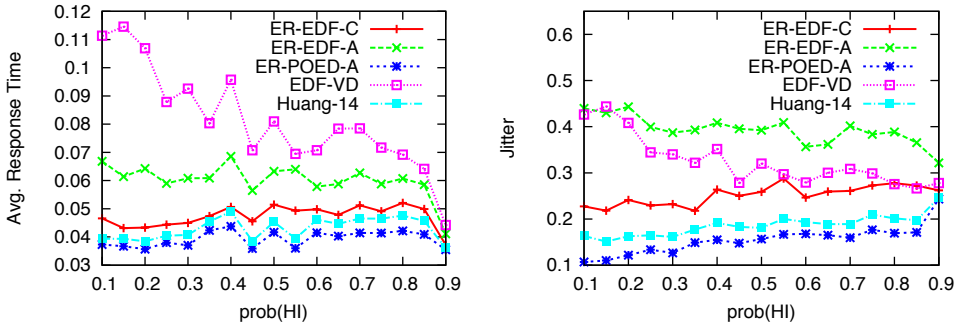
6.4. Effects of $Prob(HI)$ (Varying Workloads of HC Tasks)

In this section, we evaluate the effects of different workload mixtures with varying $Prob(HI)$, where larger values of $Prob(HI)$ indicate that there are more HC tasks and fewer LC tasks in the system, and vice versa. The results are shown in Figure 7. Here, we set $U_{bound} = 0.9$, $Z \in [1, 8]$, $Prob(c^{low}) = 0.9$ and $\eta = 2$. Each LC task has $k = 10$ early-release points within its E-MC period p^{emc}.

First, Figure 7a shows the normalized execution frequencies for LC tasks. The resulting execution frequencies of LC tasks under EDF-VD are slightly worse than those enabled by their desired periods due to their cancelled executions. Again, it can be slightly worse for adaptive EDF-VD due to the extended periods of the HI running mode. With the early-release mechanism, both ER-EDF and ER-POED can execute LC tasks twice more frequently than what is enabled by their desired MC periods. When $Prob(HI)$ becomes very large (e.g., 0.9), the generated task sets are dominated by HC tasks with only a few LC tasks, where the LC tasks can explore excessive system slack and get much improved execution frequencies. However, this still results in significant system slack waste due to limited number of LC tasks, as shown in Figure 7b.

To illustrate how smoothly LC tasks are executed at runtime, Figures 7cd further show the normalized maximum service interval (which is the ratio of the largest interval between consecutive jobs of a LC task over its desired MC period p_i) and standard deviation of their service intervals, respectively. From the results, we can see that the execution intervals for LC tasks are limited by their E-MC periods (i.e., $2 * p_i$) under ER-EDF and ER-POED, which confirms that these schedulers can ensure their mini-

mum service levels. When there are more HC tasks with larger values of $Prob(HI)$, it is more likely for the system to enter the HI running mode and cancel LC tasks under EDF-VD, where their maximum intervals can be as high as 5 times of their desired MC periods. Moreover, it becomes more difficult to accommodate the executions of LC tasks in the HI running mode under adaptive EDF-VD when there are more HC tasks, which results in quite larger high-level periods (up to $4 * p_i$) as well as maximum intervals for LC tasks. As shown in Figures 7d, except for the cases with fewer number of HC tasks (with $Prob(HI) \leq 0.3$) where most LC jobs are not discarded and executed according to their desired periods under EDF-VD,the executions of LC tasks under ER-EDF with conservative deadline assignment are relatively more smooth with smaller standard deviation on their service intervals. However, as the aggressive approach can release jobs with quite small relative deadlines, the service interval variations for LC tasks are relatively large. For adaptive EDF-VD, LC tasks's service intervals can be either p_i or $y * p_i$, which leads to large standard deviation, especially for higher $Prob(HI)$.



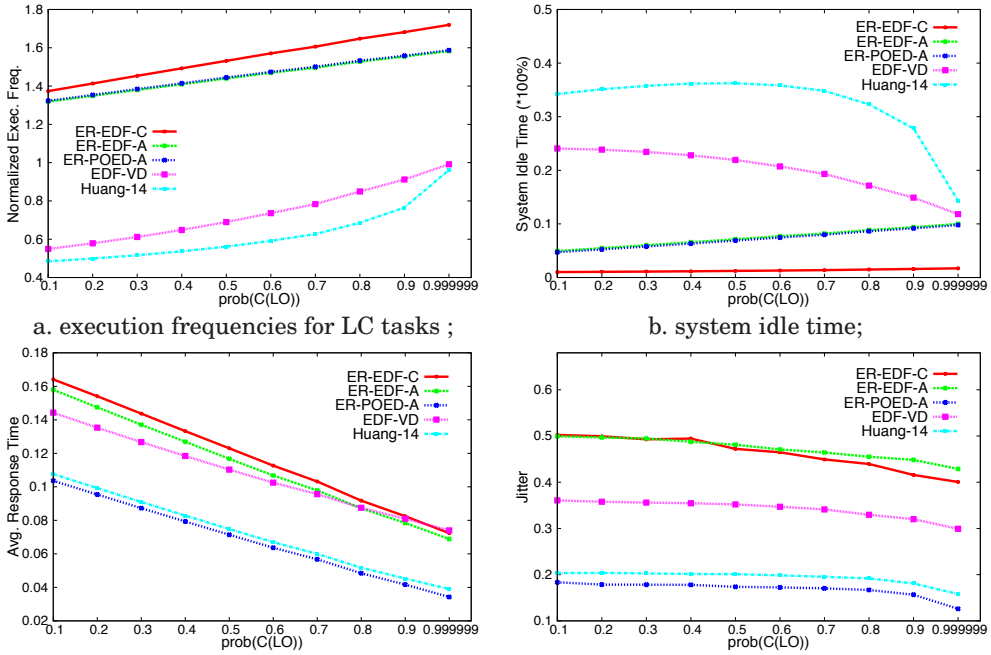
a. Average response time (delay); b. Jitter;

Fig. 8: Effects of $Prob(HI)$ on HC tasks; $U_{bound} = 0.9$, $Prob(c^{low}) = 0.9$ and $k = 10$

For HC tasks, Figure 8ab show their average response times and jitters, respectively, under different scheduling algorithms. As $Prob(HI)$ increases, there are more HC tasks and the system is more likely to enter the high-level execution mode where all LC tasks are discarded under EDF-VD. Therefore, the response times and jitters for HC tasks generally decrease under EDF-VD. The response times for HC tasks under ER-EDF are better than EDF-VD due to less interference from the relatively larger deadlines of LC tasks during executions. Moreover, with the rather large high-level periods for LC tasks and extended intervals of the HI running mode under the adaptive EDF-VD, the resulting response times and jitters of HC tasks turn out to be even better than that of ER-EDF. However, by treating HC tasks with ASAP preference and executing them early at runtime, ER-POED achieves the smallest response times and jitters, and thus the highest control performance for HC tasks.

6.5. Effects of $Prob(c^{low})$ (Runtime Behaviors of HC Tasks)

Next, we evaluate how the runtime behaviors of HC tasks affect the performance of the proposed early-release schedulers with varying $Prob(c^{low})$. Here, we assume that a HC task τ_i takes either c_i^{low} or c_i^{high} at runtime. Thus, larger values of $Prob(c^{low})$ mean that HC tasks are more likely to take their low-level WCETs at runtime, which leads to more available system slack, and vice versa. For task systems with the setting of $Prob(HI) = 0.5$, $U_{bound} = 0.9$, $Z \in [1, 8]$, $\eta = 2$ and $k = 10$, Figure 9 shows the results.



a. execution frequencies for LC tasks ;

b. system idle time;

c. average response time (delay) for HC tasks;

d. jitter for HC tasks;

Fig. 9: The effects of $Prob(c^{low})$; $U_{bound} = 0.9$, $Prob(HI) = 0.5$ and $k = 10$.

As shown in Figure 9a, with the early-release jobs of LC tasks, the achieved execution frequencies for LC tasks under ER-EDF and ER-POED are always better than 1 (which corresponds to their desired periods). Moreover, as $Prob(c^{low})$ becomes larger, more system slack can be expected as HC tasks are more likely to run for their low-level WCETs, which results in more early-released jobs and increased execution frequencies for LC tasks under both ER-EDF and ER-POED.

The normalized execution frequencies for LC tasks under both EDF-VD schemes are less than 1 due to either cancelled executions or much large extended periods of LC tasks in the HI running mode. When $Prob(c^{low})$ is small, most of the HC jobs may take their high-level WCETs and cause the system to transit to the HI execution mode, which leads to quite low execution frequencies for LC tasks (as low as around half of the one enabled by their desired periods). When $Prob(c^{low}) = 0.999999$, the resulting normalized execution frequencies of LC tasks under both EDF-VD schemes are close to (but still less than) 1. The reason is that, although the probability of HC tasks take their high-level WCETs is very low (0.0001%), once the system enters high-level execution mode, it can last quite long with $U_{bound} = 0.9$ where the releases of LC tasks return to normal only after the system becomes idle. Again, due to the extended HI running mode under the adaptive EDF-VD and rather large high-level periods $y * p_i$ (where y can be as high as 5) for LC tasks, it turns out that LC tasks are executed less frequently under the adaptive EDF-VD when compared to that of EDF-VD. The corresponding system idle time under different scheduling algorithms can be seen in Figure 9b, which matches the results in Figure 9a.

Figure 9cd show the response times (delay) and jitters for HC tasks, respectively, under different scheduling algorithms. Clearly, ER-POED performs the best as it puts HC tasks in the center stage when making scheduling decisions. Here, as HC tasks have

higher chances to take their low-level (smaller) WCETs when $Prob(c^{low})$ increases, they are more likely to finish their executions earlier at runtime, which leads to reduced response times (delay) under all scheduling algorithms. Moreover, as the difference between the response times for jobs of a HC task becomes smaller with increased $Prob(c^{low})$, the jitters for HC tasks also decrease. As it is more likely for LC tasks being discarded with smaller values of $Prob(c^{low})$ where HC tasks have higher chances to take their high-level WCETs under EDF-VD, the reduced interference leads to slightly better response times for HC tasks than that of ER-EDF where LC tasks are guaranteed to have executions according to their E-MC periods.

7. CONCLUSIONS

To address the service interruption problem for low-criticality tasks in the conventional mixed-criticality scheduling algorithms, in this work, we study an *Elastic Mixed-Criticality (E-MC)* task model and earliest-deadline based scheduling algorithms. Different from the traditional mixed-criticality task model, the central idea of E-MC is to have variable periods (i.e., service intervals) for low-criticality tasks, where their minimal service requirements are guaranteed by their maximum periods. Moreover, each low-criticality task can have a set of early-release points that allow the task to release its job instance at an earlier time instead of always waiting until its maximum period. We propose an *Early-Release (ER)* mechanism and investigate, with both *conservative* and *aggressive* approaches being considered, its application to the EDF and POED schedulers. We formally prove the correctness of the proposed ER mechanism on guaranteeing the timeliness of all (especially high-criticality) tasks through the judicious management of system slack and the early-release jobs for low-criticality tasks.

The proposed model and schedulers are evaluated through extensive simulations. The results show that, for LC tasks in a given MC task set, by moderately (2 to 3 times) extending their desired periods to obtain their maximum periods, the resulting E-MC task set becomes more schedulable under ER-EDF compared to that of the original MC task set under EDF-VD. Moreover, the execution frequencies for low-criticality tasks and response times (i.e., delay) and jitters for high-criticality tasks can be significantly improved at runtime under the proposed ER schedulers, which can in turn improve their control performance, when compared to those of the EDF-VD schedulers.

REFERENCES

P. Balbastre, I. Ripoll, J. Vidal, and A. Crespo. 2004. A task model to reduce control delays. *Real-Time Systems* 27, 3 (2004), 215–236.

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

S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. M.-Spaccamela, S. van der Ster, and L. Stougie. 2012. The Preemptive Uniprocessor Scheduling of Mixed-Criticality Implicit-Deadline Sporadic Task Systems. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*. 145–154.

S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari. 1999. Scheduling periodic task systems to minimize output jitter. In *Proc. of Int'l Conf. on Real-Time Computing Systems and Applications (RTCSA)*. 62–69.

S. Baruah, H. Li, and L. Stougie. 2010a. Mixed-criticality scheduling: improved resource augmentation results. In *Proc. of the Int'l Conf. on Computers and their Applications (CATA)*. 217–223.

S. Baruah, H. Li, and L. Stougie. 2010b. Towards the design of certifiable mixed-criticality systems. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 13–22.

S. Baruah and S. Vestal. 2008. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Euromicro Conference on Real-Time Systems*. 147–155.

S. K. Baruah, L. E. Rosier, and R. R. Howell. 1990. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-time Tasks on One Processor. *Real-Time Syst.* 2, 4 (Oct. 1990), 301–324.

I. Bate, A. Burns, and R. I. Davis. 2015. A Bailout Protocol for Mixed Criticality Systems. In *Proc. of the 27th Euromicro Conference on Real-Time Systems (ECRTS)*. 259–268.

S.A. Brandt, S. Banachowski, C. Lin, and T. Bisson. 2003. Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes. In *Prof. of The IEEE Real-Time Systems Symposium (RTSS)*. 396–407.

A. Burns and R. Davis. 2015. Mixed Criticality Systems-A Review. In *Department of Computer Science, University of York, Tech. Rep.*

G. Buttazzo, G. Lipari, and L. Abeni. 1998. Elastic task model for adaptive rate control. In *Prof. of The 19th IEEE Real-Time Systems Symposium (RTSS)*. 286–295.

D. de Niz, K. Lakshmanan, and R. Rajkumar. 2009. On the scheduling of mixed-criticality real-time task sets. In *Proc. of The 30th IEEE Real-Time Systems Symposium (RTSS)*. 291–300.

A. Easwaran. 2013. Demand-Based Scheduling of Mixed-Criticality Sporadic Tasks on One Processor. In *Proc. of the 34th IEEE Real-Time Systems Symposium (RTSS)*. 78–87.

P. Ekberg and W. Yi. 2012. Bounding and Shaping the Demand of Mixed-Criticality Sporadic Tasks. In *Proc. of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*. 135–144.

B. Enrico and A. Cervin. 2008. Delay-aware period assignment in control systems. In *Proc. of the 29th IEEE Real-Time Systems Symposium (RTSS)*. 291–300.

N. Guan, P. Ekberg, M. Stigge, and W. Yi. 2011. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *Proc. of the IEEE Real-Time Systems Symposium*. 13–23.

Y. Guo, H. Su, D. Zhu, and H. Aydin. 2015. Preference-Oriented Real-Time Scheduling and Its Application in Fault-Tolerant Systems. *Journal of System Architecture (JSA)* 60, 2 (2015), 127–139.

P. Huang, G. Giannopoulou, N. Stoimenov, and L. Thiele. 2014. Service adaptions for mixed-criticality systems. In *Proc. of 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 125–130.

O.R. Kelly, H. Aydin, and B. Zhao. 2011. On Partitioned Scheduling of Fixed-Priority Mixed-Criticality Task Sets. In *Proc. of the 8th IEEE Int'l Conf. on Embedded Software and Systems (ICESS)*. 1051–1059.

T.W. Kuo and A.K. Mok. 1991. Load adjustment in adaptive real-time systems. In *Proc. of the 12th IEEE Real-Time Systems Symposium (RTSS)*. 160–170.

H. Li and S. Baruah. 2010a. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *Proc. of the 31st IEEE Real-Time Systems Symposium (RTSS)*. 183–192.

H. Li and S. Baruah. 2010b. Load-based schedulability analysis of certifiable mixed-criticality systems. In *Proc. of the 10th ACM Int'l Conf. on Embedded Software (EMSoft)*. 99–108.

H. Li and S. Baruah. 2012. Global Mixed-Criticality Scheduling on Multiprocessors. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*. 166–175.

C.L. Liu and J.W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.

A. Masrur, D. Muller, and M. Werner. 2015. Bi-Level Deadline Scaling for Admission Control in Mixed-Criticality Systems. In *Proc. of the Conf. on Embedded and Real-Time Comp. Sys. and App.* 100–109.

M.S. Mollison, J.P. Erickson, J.H. Anderson, S.K. Baruah, and J.A. Scoredos. 2010. Mixed-Criticality Real-Time Scheduling for Multicore Systems. In *Proc. of the Conf. on Comp. and Info. Tech. (CIT)*. 1864–1871.

T. Park and S. Kim. 2011. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *Proc. of Intl Conf. on Embedded Software (EMSoft)*. 253–262.

F. Santy, L. George, P. Thierry, and J. Goossens. 2012. Relaxing Mixed-Criticality Scheduling Strictness for Task Sets Scheduled with FP. In *Proc. of the Euromicro Conf. on Real-Time Systems (ECRTS)*. 155–165.

D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. 2012. *Mixed Critical Earliest Deadline First*. Technical Report. Technical Report TR-2012-22, Verimag Research Report.

H. Su and D. Zhu. 2013. An elastic Mixed-Criticality task model and its scheduling algorithm. In *Proc. of the Design, Automation and Test in Europe (DATE)*. 147–152.

H. Su, D. Zhu, and N. Guan. 2014. Service Guarantee Exploration for Mixed-Criticality Systems. In *Proc. of 20th IEEE Int'l Conf. on Embedded and Real-Time Comp. Systems and App. (RTCSA)*. 1–10.

H. Su, D. Zhu, and D. Mossé. 2013. Scheduling Algorithms for Elastic Mixed-Criticality Tasks in Multicore Systems. In *Proc. of IEEE Int'l Conf. on Embedded and Real-Time Comp. Systems and App.* 352–357.

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

F. Zhang, K. Szwaykowska, V. Mooney, and W. Wolf. 2008. Task Scheduling for Control Oriented Requirements for Cyber-Physical Systems. In *Proc. of The IEEE Real-Time Systems Symposium (RTSS)*. 47–56.

T. Zhang, N. Guan, Q. Deng, and W. Yi. 2014. On the Analysis of EDF-VD Scheduled Mixed-Criticality Real-Time Systems. In *Proc. of the IEEE Int'l Symposium on Industrial Embedded Systems*. 179–188.

D. Zhu and H. Aydin. 2009. Reliability-Aware Energy Management for Periodic Real-Time Tasks. *IEEE Trans. on Computers* 58, 10 (2009), 1382–1397.