Technical Report: CS-TR-2013-010, UTSA

Scheduling Algorithms for Elastic Mixed-Criticality Tasks in Multicore Systems (Extended Version)

Prepared on June 10, 2013

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

Daniel Mossé
University of Pittsburgh
mosse@cs.pitt.edu

***Abstract*—The *Elastic Mixed-Criticality (E-MC)* task model and an *Early-Release EDF (ER-EDF)* scheduling algorithm have been studied to address the service interruption problem for low-criticality tasks in uniprocessor systems, where the minimum service requirements of low-criticality tasks are guaranteed by their maximum periods. In this paper, focusing on multicore systems, we first investigate and empirically evaluate the schedulability of E-MC tasks under partitioned-EDF (P-EDF) by considering various task-to-core mapping heuristics. Then, to improve the service levels of low-criticality tasks by exploiting slack at runtime, *with* and *without* task migrations being considered, we study both *global* and *local* early-release schemes. Moreover, considering varied migration overheads between different cores, we propose the *multicore-aware and migration constrained* global-ER schemes. The simulation results show that, compared to the state-of-the-art Global EDF-VD scheduler, P-EDF with various partitioning heuristics can lead to many more schedulable E-MC task sets. Moreover, our proposed global and local ER schemes can significantly improve the execution frequencies (and thus service levels) of low-criticality tasks, while Global EDF-VD may severely affect them by discarding most of their task instances at runtime (especially for systems with more cores). Furthermore, by allowing task migrations, global-ER schemes can dramatically improve low-criticality tasks' service levels for partitionings that do not *balance* high- and low-criticality tasks among the cores.**

I. Introduction

As the next-generation engineering systems, cyber-physical systems (CPS) can have computation tasks with different levels of importance according to their functionalities that further lead to different criticality levels. For instance, it is typically more important to guarantee flight-critical functionality (e.g., flight control) than mission-oriented functionality (e.g., capturing photos in unmanned aerial vehicles) [1]. Note that, in general, the flight-critical functionalities are certified by certification authorities (such as US Federal Aviation Authority and European Aviation Safety Agency) with extremely strict and pessimistic assumptions, which are very hardly to occur in reality [5]. In the mean time, for mission-critical functionalities, they are normally validated by vendor manufacturers in a less rigorous standard. Since tasks with different criticality levels need to share computing resources, how to efficiently schedule such *mixed-criticality (MC)* tasks while satisfying their specific requirements has been identified as one of the most fundamental issues in CPS [5].

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

Without proper provisions for such mixed-criticality tasks, traditional scheduling algorithms are likely to cause the so-called *"priority inversion"* problems [18]. Based on the MC task model where a task generally has multiple worst case execution times (WCETs) according to different certification levels, several studies have been reported for uniprocessor [2], [5], [6], [9], [18], [24] and multiprocessor systems [12], [11], respectively.

Note that, in most existing MC scheduling algorithms, the execution of low-criticality tasks is rather *opportunistic*. 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, *all* low-criticality task instances will be discarded *immediately* to provide the high-level computation need of high-criticality tasks. Such an approach can cause serious service interruption and significant performance loss for low-criticality tasks, especially when they are control tasks, where their performance is mainly affected by the execution frequencies and service intervals [20], [25].

To mitigate such a problem, based on the traditional mixed-criticality task model, 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 service levels [19]. However, this approach still can not provide any guarantee for the service levels of low-criticality tasks. In [17], Mollison *et al.* further studied a hierarchical scheduling algorithm that redistributed slacks to low-criticality tasks.

Based on a different principle, we have studied the *Elastic Mixed-Critical (E-MC)* task model and an *early-release EDF (ER-EDF)* scheduling algorithm for uniprocessor systems [22]. Instead of executing low-criticality tasks opportunistically, the central idea of E-MC is to have variable periods (i.e., service intervals) for low-criticality tasks, where their minimal service levels are represented by their maximum periods and guaranteed offline. At runtime, by properly reclaiming the slack, ER-EDF allows low-criticality tasks to release their instances earlier than their maximum periods (i.e., more frequently) and thus to improve their service levels.

Inspired by but different from all existing work, we study in this paper the scheduling algorithms for E-MC tasks in multicore systems. Focusing on the partitioned-EDF scheduling, we first investigate the schedulability of E-MC tasks under various task-to-core mapping heuristics and then study different early-release techniques with the goal of improving the service levels of low-criticality tasks at runtime. The main contributions of this work are summarized as follows:

- First, we investigate the schedulability of E-MC tasks under partitioned-EDF with various task-to-core mapping heuristics (e.g., First-Fit, Best-Fit and Worst-Fit) and task ordering policies (such as Decreasing Utilization and Decreasing Criticality).
- Second, we propose both *local* and *global* early-release (ER) schemes under partitioned-EDF, which prohibits and allows task migrations, respectively, to improve the service levels of low-criticality tasks at runtime;
- Third, considering difference in task migration overheads between different cores, we devise *multicore-aware and migration constrained* global early-release schemes;
- Finally, we evaluated the proposed schemes through extensive simulations.

Our simulation results show that, with all the mapping heuristics being considered, partitioned-EDF can schedule many more E-MC task sets when compared to that of Global EDF-VD [11], the state-of-the-art global MC scheduler. Moreover, partitioned-EDF with early-release techniques can significantly improve the service levels for low-criticality tasks. On the other hand, Global EDF-VD could have a large negative impact due to possibly unbounded cancellation of their task instances at runtime, especially for systems with more cores. Lastly, for task-to-core mappings (e.g., FF and BF) that have unbalanced high- and low-criticality tasks, the global early-release schemes that allows task migrations can drastically increase the service levels for low-criticality tasks when compared to that of the local early-release scheme.

The remainder of this paper is organized as follows. Section II reviews closely realted works. Section III presents task and system models. The schedulability of partitioned-EDF for E-MC tasks is investigated in Section IV. Section V presents both local and global early-release techniques for low-criticality tasks. The evaluation results are discussed in Section VI and Section VII concludes the paper.

II. Closely Related Works

As the first work to address the scheduling of such tasks, Vestal formalized the mixed-criticality scheduling problem with multiple certification requirements at different degrees of confidence and studied a fixed-priority scheduling algorithm in [24]. Focusing on a finite number of mixed-criticality jobs, several scheduling algorithms have been studied based on the Own-Criticality Based Priority (OCBP) approach [4], [5], [16]. In [21], the Mixed-Critical EDF scheduling is studied, which was shown to dominate OCBP-based schemes with higher schedulability ratio and reduced complexity.

For sporadic MC tasks, an extension of OCBP was studied that has a pseudo-polynomial complexity [15] and a more efficient algorithm has been proposed [10]. Based on fixed-priority preemptive scheduling (such as RMS), a zero-slack scheduling approach [18] 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. The authors further studied the *Priority and Criticality Ceiling Protocol (PCCP)* that addresses task synchronization problems in the zero-slack based algorithms [14]. More recently, another 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]. Based on the demand-bound function analysis, an extension of EDF-VD using an efficient relative deadline tuning technique can achieve better schedulability [9].

As multicore processors become popular for modern real-time systems, several works have studied the scheduling problem of mixed-criticality tasks in multiprocessor systems. Focusing on partitioned scheduling, researchers investigated the schedulability of various partitioning heuristics (e.g., First-Fit, Best-Fit and Worst-Fit) and task sorting policies (e.g., Decreasing-Utilization and Decreasing-Criticality) under fixed-priority (RMS) scheduling [12]. For global scheduling, the Global EDF-VD algorithm extends Global-EDF to schedule mixed-criticality tasks in multiprocessor systems [11]. In [3], Baruah *et al.* studied a global-based scheme with OCBP for a finite collection of independent mixed-criticality jobs and partitioned-based EDF-VD for sporadic tasks.

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], [6], [11]. Such an approach can cause serious service interruption and significant performance loss for low-criticality tasks. Based on the traditional mixed-critical task model, Santy *et al.* studied an online scheme that calculates a delay-allowance before entering the high-level execution mode and thus delays the qcancellation of low-criticality tasks to improve their services [19].

In our recent work [22], we proposed an *Elastic Mixed-Critical (E-MC)* task model and studied the *early-release EDF (ER-EDF)* scheduling algorithm. The central idea of E-MC is to have variable periods (i.e., service intervals) for low-criticality tasks, where their minimal service levels are represented by their maximum periods and guaranteed offline. At runtime, by properly reclaiming the slack, ER-EDF allows low-criticality tasks to release earlier than their maximum periods and thus to improve their service levels.

In this work, focusing on partitioned scheduling, we studied the performance of various partitioning heuristics for a set of E-MC tasks running on multiprocessor systems. In addition, when there is not enough slack on a low-criticality task's host processor, we further investigate the scheme that allows it to temporarily migrate and reclaim slack on other processors, which is different from the existing works.

III. System and Task Models

Note that, in the traditional MC task model, low-criticality tasks are executed *opportunistically* with potentially *unbounded* cancellation at runtime to provide the worst-case guarantee for high-criticality tasks [2]. Following a completely different approach, the *Elastic Mixed-Criticality (E-MC)* task model aims at guaranteeing the minimum service requirements of low-criticality tasks *offline* and improving their service levels at runtime without sacrificing the worst-case guarantee for high-criticality tasks [22].

We consider systems that have only two different criticality levels, HI and LO. There is a set of n E-MC tasks $\Gamma = \{\tau_1, \dots, \tau_n\}$ and each task τ_i has a parameter ζ_i to denote its criticality level. To guarantee the computation demands of HI tasks in the worst case scenario, they have the same timing parameters as those in traditional MC task model [2], where c_i^{LO} and c_i^{HI} denotes its low- and high-level worst case execution times (WCETs), respectively; moreover, p_i represents its period [22]. In general, c_i^{LO} and c_i^{HI} (where $c_i^{LO} < c_i^{HI}$) are specified by system designers and certification authorities, respectively [2]. The low and high bounds for utilization of task τ_i are further defined as $u_i^{LO} = \frac{c_i^{LO}}{p_i}$ and $u_i^{HI} = \frac{c_i^{HI}}{p_i}$, respectively.

The main **difference** between E-MC and MC task models comes from the representation of low-criticality tasks. To represent its minimum service level, each low-criticality task τ_i has a maximum period (i.e., service interval) p_i^{max} [22], in addition to its low WCET c_i^{LO} and (desired[1]) period p_i [2]. Moreover, to facilitate the improvement of its service level at runtime, a low-criticality task τ_i also has a set of k_i possible *early-release points* $P_i^{ER} = \{p_i^1, \dots, p_i^{k_i}\}$ [22]. Similarly, the (desired) low and minimum utilizations of τ_i are defined as $u_i^{LO} = \frac{c_i^{LO}}{p_i}$ and $u_i^{MIN} = \frac{c_i^{LO}}{p_i^{max}}$, respectively.

	MC Parameters				Additional E-MC Parameters	
	ζ_i	c_i^{LO}	c_i^{HI}	p_i	p_i^{max}	P_i^{ER}
τ_1	HI	10	20	25	-	-
τ_2	HI	4	8	11	-	-
τ_3	LO	2	-	12	12	{4}
τ_4	LO	2	-	10	10	{7}

TABLE I: An example E-MC task set with four tasks.

As an example, Table I shows the timing parameters for a set of four E-MC tasks. For high-criticality tasks τ_1 and τ_2, their timing parameters are the same as in MC task model and shown in the left part of the table. However, for low-criticality tasks τ_3 and τ_4, in addition to their MC timing parameters, additional E-MC parameters (i.e., p_i^{max} and P_i^{ER}) are shown on the right side of the table. We have $p_i^{max} = p_i$ for low-criticality tasks since they are schedulable under partitioned-EDF even with p_i as shown in Section V and there is no need to have extended and larger p_i^{max} [22].

We consider the set of E-MC tasks to be executed on a multicore system that have m identical cores, which may share different levels of on/off-chip caches. The commonly adopted parallel scheduling constraints are assumed [8]: a task can only run on one core at any given time (i.e., tasks are not parallel); and a core can only be allocated to one task at any given time (i.e., no hyperthreading). Moreover, it is assumed that any instance of task τ_i will not run longer than $c_i^{\zeta_i}$; otherwise, such instance will be aborted and an error will be reported [2]. Given the above definitions and assumptions, a given set of E-MC tasks is said to be **E-MC schedulable** under a given scheduling algorithm if both the high-level WCETs of high-criticality tasks and the minimum service requirements (represented by p^{max}) of low-criticality tasks can be guaranteed in the worst-case scenario [22].

[1]Essentially, p_i is kept in the E-MC task model to act as the bridge for comparison with existing MC scheduling algorithms [22].

IV. Schedulability of E-MC Tasks under P-EDF

In [22], we studied the EDF schedulability condition for E-MC tasks running on uniprocessor systems. In this section, we investigate and empirically evaluate the schedulability of E-MC tasks under the *partitioned-EDF (P-EDF)* scheduling with various task-to-core mapping heuristics on multicore systems.

A. Schedulability Condition

First, based on the definitions of task utilizations, similar to those in [2], we use the following notation:

- $U_\Gamma(H, L) = \sum_{\tau_i \in \Gamma \wedge \zeta_i = HI} u_i^{LO}$
- $U_\Gamma(H, H) = \sum_{\tau_i \in \Gamma \wedge \zeta_i = HI} u_i^{HI}$
- $U_\Gamma(L, L) = \sum_{\tau_i \in \Gamma \wedge \zeta_i = LO} u_i^{LO}$
- $U_\Gamma(L, MIN) = \sum_{\tau_i \in \Gamma \wedge \zeta_i = LO} u_i^{MIN}$

Suppose that a given task-to-core partitioning is $\Pi = \{\Gamma_1, \cdots, \Gamma_m\}$, where $\Gamma = \Gamma_1 \cup, \cdots, \cup \Gamma_m$. From Lemma 1 in [22], we can have the following theorem.

Theorem 1: For a set Γ of E-MC tasks running on an m-core system, a given task-to-core partitioning $\Pi = \{\Gamma_1, \cdots, \Gamma_m\}$ is E-MC feasible under the partitioned-EDF, if:

$$U_{\Gamma_k}(H, H) + U_{\Gamma_k}(L, MIN) \leq 1 \qquad (1)$$

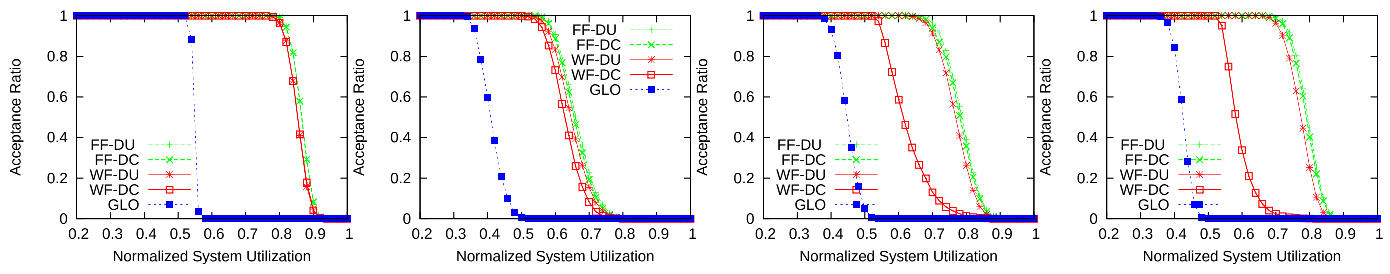
where $k = 1, \cdots, m$.

$\diamondsuit$

Note that the E-MC schedulability conditions represented in Equation (1) depend only on the high utilization of high-criticality tasks and the minimum utilization of low-criticality tasks. Based on such utilizations, it is well-known that finding the optimal partitioning of a given set of tasks to satisfy Equation (1) is NP-hard [8]. Therefore, in what follows, we focus on various task-to-core partitioning heuristics and empirically evaluate their E-MC schedulability performance.

B. Evaluations of E-MC Schedulability under P-EDF

We have evaluated [22] the schedulability of E-MC tasks under EDF and compared it against EDF-VD [2] on uniprocessor systems. The results show that better schedulability (in terms of acceptance ratio of generated task sets) can be achieved under EDF for E-MC tasks when the minimum service levels of low-criticality tasks can be set as one-fifth to half of their desired service levels. In this section, we first evaluate the



a. $m = 8$, $prob(HI) = 0.2$; b. $m = 8$, $prob(HI) = 0.5$; c. $m = 8$, $prob(HI) = 0.8$; d. $m = 16$, $prob(HI) = 0.8$;

Fig. 1: The acceptance ratio of E-MC task sets under P-EDF with various partitioning heuristics and Global EDF-VD.

schedulability of E-MC tasks under P-EDF and Global EDF-VD [11].

Note that, when partitioning tasks among cores, there are two main issues [8]: (a) core selection for a given task; and (b) the order of tasks being allocated. In this work, we consider the core selection heuristics similar to those used in [12], namely *First-Fit (FF)*, *Best-Fit (BF)* and *Worst-Fit (WF)*. For the order of tasks being allocated, the following heuristics are considered:

- Decreasing Utilization (DU): tasks are sorted in non-increasing order of their utilizations; u_i^{HI} and u_i^{MIN} are used for high- and low-criticality tasks, respectively;
- Decreasing Criticality (DC): high-criticality tasks are allocated before low-criticality tasks; for tasks with the same criticality level, they are sorted in non-increasing order of their utilizations; again, u_i^{HI} and u_i^{MIN} are used for high- and low-criticality tasks, respectively;

Here, task sets are generated following a similar method as used in [11]. The high (or desired low) utilization of a high- (or low-) criticality task is randomly generated within $[0.05, 0.5]$. Then, for high-criticality tasks, their low utilization are calculated based on the ratio of high over low utilizations being randomly generated within $[1, 8]$; for low-criticality tasks, their minimum utilizations are set as $\frac{1}{k}$ of their corresponding desired low utilizations. It turns out that even with $k = 1$, more task sets can be scheduled under P-EDF compared to Global EDF-VD. Therefore, we set $k = 1$ (i.e., $u_i^{MIN} = u_i^{LO}$) for low-criticality tasks in this work. Moreover, for each data point, $10,000$ task sets are generated. Also, $prob(HI)$ denotes the probability of a task having high-criticality.

Figure 1 shows the acceptance ratio of generated task sets under P-EDF with different partitioning heuristics and Global EDF-VD (denoted as *GLO*) with varying normalized system utilization, which is defined as $\frac{U^{bound}}{m}$. Here, $U^{bound} = \max\{U_\Gamma(H,H), U_\Gamma(H,L) + U_\Gamma(L,L)\}$ [11] and we consider $m = 8$ and $m = 16$. For the cases of 8-core systems (i.e., $m = 8$), we consider $prob(HI) = 0.2, 0.5$ and 0.8, and the results are shown in Figures 1a, 1b, and 1c, respectively. Figure 1d further shows the results for the case of $m = 16$ and $prob(HI) = 0.8$, that is, doubling the number of cores compared with Figure 1c. Since the results for BF-heuristics are very close to those for FF-heuristics, we only show the results for FF and WF heuristics under P-EDF.

From the results, we can see that many more task sets can be scheduled under P-EDF with the partitioning heuristics under consideration when compared to Global EDF-VD. For P-EDF, not surprisingly, FF performs better than WF since it aims at scheduling more tasks in systems with a given number of cores. Moreover, different task orderings (DC vs. DU) have very little impact on the schedulability of P-EDF when FF heuristic is used for core selection.

When using WF for core selection, P-EDF performs better with the Decreasing Utilization (DU) ordering of tasks (and is very close to that of FF) since it is well-known such a heuristic can achieve more balanced workload among cores and thus increase the schedulability of the tasks [12]. However, when WF is combined with the Decreasing Criticality (DC), P-EDF performs worse in terms of schedulability. The reason is that the DC heuristic aims at balancing high- and low-criticality workload on cores under WF instead of increasing its schedulability. However, as shown Section VI, better improvements for the execution frequencies (i.e., service levels) of low-criticality tasks can be obtained under P-EDF with WF and DC, especially for the local early-release scheme where task migrations are not allowed.

The different mixtures of high- and low-criticality tasks (i.e., different settings of $prob(HI)$) also have some effects on a task set's schedulability, where task sets with balanced number of high- and low-criticality tasks (i.e., $prob(HI) = 0.5$) have worse schedulability. When there are more cores in a system (e.g., $m = 16$), as shown in Figure 1d, Global EDF-VD has slightly worse schedulability. For P-EDF, WF with DC also performs slightly worse, but other heuristics perform better as there are more chances to fit tasks on more cores.

V. Global Early-Release for P-EDF

The schedulability conditions for a set of E-MC tasks under P-EDF represented in Equation (1) provides worst-case guarantees for low-criticality tasks and high-criticality tasks' high computation demands. However, high-criticality tasks rarely utilize their high WCETs [24] and a large amount of dynamic slack can be expected at runtime. Such slack can be exploited to execute low-criticality tasks more frequently and thus to improve their service levels with proper slack reclamation and early-release management [22].

In what follows, we first review the essential ideas of early-release of low-criticality tasks (Section V-A) and present a motivational example to show that task migrations can further improve low-criticality tasks' service levels (Section V-B). Details for both *local* and *global* early-release schemes under P-EDF (i.e., without and with task migrations, respectively) are given in Section V-C. By taking task migration overheads into consideration, we describe the *multicore-aware and migration-constrained* early-release schemes in Section V-D.

A. Ideas of Early-Release for Low-Criticality Tasks

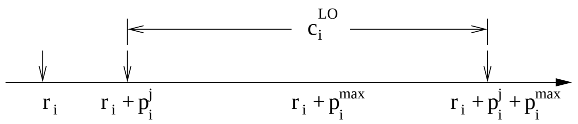


Fig. 2: Early-release of a low-criticality task instance [22].

Suppose that a low-criticality task τ_i (i.e., $\zeta_i = LO$) has its last instance released at time r_i as illustrated in Figure 2. We further assume that this instance finishes its execution at time t, which is no later than an early-release point p_i^j of task τ_i (that is, there are $r_i < t \leq r_i + p_i^j$ and $1 \leq j \leq k_i$). When no early-release is considered, the next instance of task τ_i *should* be released at time $r_i + p_i^{max}$, which satisfies τ_i's minimum service requirement. However, if early-release is enabled and task τ_i releases the next instance at its early-release point $r_i + p_i^j$, the deadline of this new instance will be $r_i + p_i^j + p_i^{max}$ as shown in Figure 2.

Intuitively, such an early-release of a task instance introduces extra workload into the system. Therefore, to avoid overloads that affect the execution of high-criticality tasks, judicious slack management is required for such early-release decisions [22]. Specifically, if such an early-released instance will be executed on task τ_i's host core, we have that τ_i can get its own share of $p_i^j \cdot u_i^{MIN}$ for the interval $[r_i + p_i^{max}, r_i + p_i^j + p_i^{max}]$ [7]. Hence, to provide c_i^{LO} time units for task τ_i's new instance before its deadline $r_i + p_i^j + p_i^{max}$ on its host core, the amount of local slack needed will be $S^L = c_i^{LO} - p_i^j \cdot u_i^{MIN}$ [22].

B. A Motivational Example: Benefits of Task Migrations

Suppose that the example E-MC task set with four tasks as shown in Table I will be executed on a dual-core system. We further assume that there are $\Gamma_1 = \{\tau_1, \tau_3\}$ and $\Gamma_2 = \{\tau_2, \tau_4\}$ for the task-to-core mapping. That is, tasks τ_1 and τ_3 are mapped to the first core, while tasks τ_2 and τ_4 to the second core. Therefore, we have:

$$U_{\Gamma_1}(H,H) + U_{\Gamma_1}(L,MIN) = 20/25 + 2/12 = 29/30 < 1$$
$$U_{\Gamma_2}(H,H) + U_{\Gamma_2}(L,MIN) = 8/11 + 2/10 = 102/110 < 1$$

From Theorem 1, we know that the example task set is E-MC schedulable with the above task-to-core mapping under P-EDF. That is, when the tasks on each core are scheduled under EDF, both the high WCETs of high-criticality tasks and the minimum service requirements of low-criticality tasks can be guaranteed. The partial schedule of the task set within the interval $[0, 30]$ under P-EDF is shown in Figure 3a. Here, for the high-criticality tasks τ_1 and τ_2, we assume that their first two instances use their low WCETs while the third instance of task τ_2 uses its high WCET.

From the schedule, we can see that the over-provisioned high WCETs for high-criticality tasks can generate lots of slack at runtime. In particular, there are 18 units of slack time (10 units on core one and 8 units on core two that are marked as "idle") in the schedule of this dual-core system within the interval of $[0, 30]$. We should mention that the slack time also includes those generated from the spare capacities[2], which are $\frac{1}{30}$ and $\frac{4}{55}$ on cores one and two, respectively.

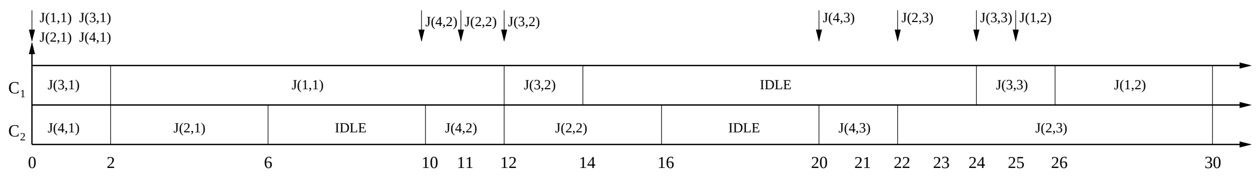
If the *early-release EDF (ER-EDF)* technique [22] is applied on each core *independently*, where a low-criticality task can only reclaim the slack time on its host core with no task migration being considered (i.e., *local* early-release, L-ER), Figure 3b shows the partial schedule of the task set within the interval $[0, 30]$ under P-EDF. On the first core, task τ_3 releases its third, forth and fifth instances (labeled as "L-ER") early at time 16, 20 and 24, respectively, by effectively reclaiming the slack time generated by the high-criticality task τ_1. Compared to the P-EDF schedule with no early-release in Figure 3a, two more task instances of τ_3 can be executed within $[0, 30]$. Similarly, task τ_4 releases early once at time 7 on the second core.

Note that, to regulate the releases of task instances, a low-criticality task can only release its instances (early) at one of its early-release points [22]. Therefore, not all slack on a core may be reclaimed by its low-criticality tasks due to the *time-dependent availability* of slack [26] as well as the alignment of low-criticality tasks' early-release points. Hence, from Figure 3b, we can see that there are still 14 units of *unused* slack time (6 and 8 units on cores one and two, respectively).

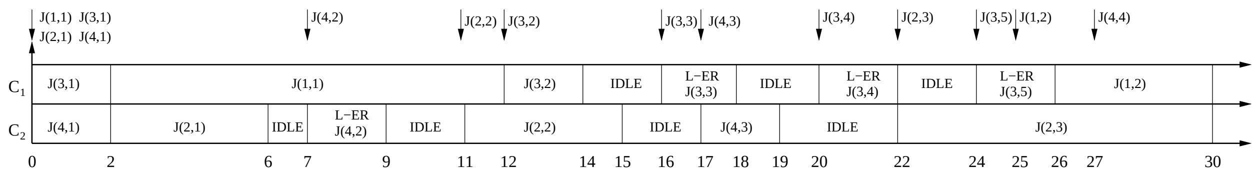
Moreover, due to unevenly distributed high-criticality tasks among the cores, at an early-release point of a low-criticality task, it is possible that other cores have plenty of available slack while the amount of reclaimable slack on the task's host core is not enough. For such cases, if task migrations are allowed, a low-criticality task can have its early-released instance be *temporarily* migrated and executed on another core by properly reclaiming the core's slack time. Figure 3c further shows the schedule of the example task set under P-EDF with *global* early-release (G-ER) that allows task migrations.

Here, at time 14 (an early-release point of τ_4), there is not enough slack on τ_4's host core two. However, the slack time on core one is enough to temporarily execute an instance of τ_4, should it release and migrate the instance to core one at time 14. Therefore, task τ_4 has its third instance $J(4, 3)$ be executed on core one (marked as "G-ER") as shown in Figure 3c. Similarly, the forth instance $J(4, 4)$ of τ_4 can also be released

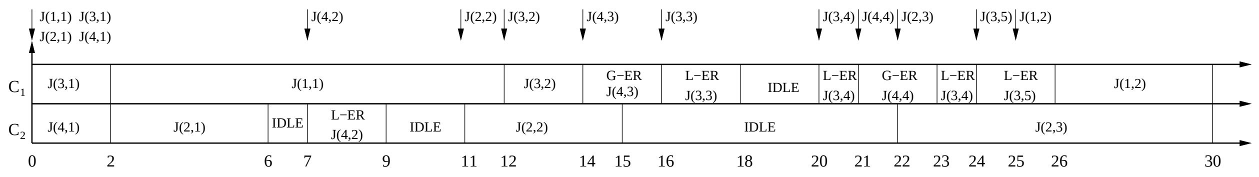
[2]Here, two wrapper-tasks $WT_1(c = 1, p = 30)$ and $WT_2(c = 4, p = 55)$ are created, for cores one and two, respectively, to periodically introduce the spare capacity as dynamic slack into the system [22], [26].



a. The schedule of the example task set under P-EDF with no early-release.



b. The schedule of the example task set under P-EDF with local early-release.



c. The schedule of the example task set under P-EDF with global early-release that allows task migrations.

Fig. 3: Different schedules for the example E-MC task set within the interval $[0, 30)$ on a dual-core system.

early and executed on core one. Note that compared to the schedule in Figure 3b, where $J(4,4)$ is released at time 27 and executed after time 30, $J(4,4)$ is able to release at time 21 and be executed before time 30. Moreover, the total idle time (wasted slack) within the interval under consideration is also reduced to 12 from 14.

C. Global Early-Release considering Task Migrations

Note that, for an E-MC feasible task-to-core mapping under P-EDF, the high computation demands of high-criticality tasks are guaranteed on their host cores. Thus, there is no migration needed (or allowed) for such high-criticality tasks at runtime. Moreover, due to the time-dependent availability of slack time [26], the amount of available slack time on cores at any given time can have quite large variation. Such variation may also come from unbalanced high- and low-criticality workload on cores in the task-to-core mapping as shown in Section VI. Hence, it is possible that, at an early-release point of a low-criticality task, other cores have enough reclaimable slack while the task's host core does not. Therefore, for low-criticality tasks, the global early-release with task migrations can achieve better improved service levels when compared to applying ER-EDF on each core independently (i.e., local early-release).

Algorithm 1 summarizes the formal steps of the global early-release scheme for P-EDF, which will be invoked at an early-release point ($t = r_i + p_i^j$) of a low-criticality task τ_i on its host core C_x. We assume that the wrapper-task mechanism is exploited on each core to manage their slack times independently [22], [26]. The available slack at time t on core C_x is saved in its slack queue $SQ_x(t)$. Note that the algorithm can be invoked simultaneously on multiple cores. Therefore, proper synchronization is needed when accessing

Algorithm 1 : P-EDF with Global Early-Release

```
 1: Input: $\tau_i$, $C_x$ and $t = r_i + p_i^j$ (an ER point of $\tau_i$);
 2: $d_i^{new} = t + p_i^{max}$; //deadline of $\tau_i$'s new instance
 3: $S^L = c_i^{LO} - p_i^j \cdot u_i^{MIN}$; //local slack needed on core $C_x$
 4: if ($S^L \leq$ CheckSlack($SQ_x(t)$, $d_i^{new}$)) then
 5:     ReclaimSlack($SQ_x(t)$, $S^L$);//local early-release on $C_x$
 6:     Enqueue($ReadyQ_x$, $\tau_i$); //add $\tau_i$ to $C_x$'s ready queue
 7: else
 8:     $S^G = c_i^{LO}$; //slack needed for global early-release
 9:     if ($\exists y, S^G \leq$ CheckSlack($SQ_y(t)$, $d_i^{new}$) $\wedge y \neq x$) then
10:         ReclaimSlack($SQ_y(t)$, $S^G$);//global early-release
11:         Enqueue($ReadyQ_y$, $\tau_i$); //migrate $\tau_i$ to core $C_y$
12:         AddSlack($SQ_x(t)$, $p_i^j \cdot u_i^{MIN}$, $d_i^{new}$);//slack on $C_x$
13:     else
14:         SetTimer($r_i + p_i^{j+1}$);//set next ER time for $\tau_i$
15:     end if
16: end if
```

the global data structures (such as the cores' slack and ready-task queues), which are not shown for simplicity.

In the algorithm, the expected deadline of task τ_i's next instance is first obtained assuming it is released at time t (line 2). This deadline will be used to properly reclaim the slack time either locally on core C_x or globally on other cores. Note that, under the EDF scheduling, a task instance may only reclaim the slack time that is available no later than the deadline of the task instance [26].

Considering the overhead of task migrations (which will be addressed in more detail in the next section), it is preferable for task τ_i being executed locally on its host core C_x. Thus, the amount of local slack needed S^L for τ_i to release the next instance at time t on core C_x is determined next (line 3). If there is enough reclaimable slack on core C_x, task τ_i will reclaim the slack properly with the help of function *ReclaimSlack()* [26] and add its new instance to C_x's ready task queue $ReadyQ_x$ (lines 5 and 6).

However, if the amount of reclaimable slack for τ_i is not enough on core C_x, τ_i will turn to other cores for possible global early-release of its next instance. Note that the objective is **not** to *permanently* migrate task τ_i (and all its future instances) to other cores. Instead, our goal is to find a core C_y with available slack that is enough to *temporarily* execute τ_i's next instance only. Once this instance completes its execution on C_y, future instances of τ_i are still assumed to be released on its host core C_x.

Based on the above design principle, we can find that the amount of slack needed for τ_i's next instance on any other core will be $S^G = c_i^{LO}$ (line 8), which is different from the amount of local slack needed on C_x. Note that there is no reserved time share for task τ_i on other cores and all time needed for executing τ_i's next instance has to come from the reclaimable slack. If there exists a core C_y that has enough reclaimable slack for τ_i at time t, the next instance of τ_i will be added to the ready task queue $ReadyQ_y$ after properly reclaiming the slack on core C_y (lines 10 and 11). In this case, task τ_i's own share (in the amount of $p_i^j \cdot u_i^{MIN}$) on core C_x will become slack and needs to be added to C_x's slack queue (line 12).

In case there is no core that has enough reclaimable slack for task τ_i at its early-release time t, a timer is set for the next early-release time point of τ_i (line 14). If t is the last early-release point for task τ_i, we assume that $p_i^{k_i+1} = p_i^{max}$ and the next instance of τ_i will be released normally (i.e., p_i^{max} time units after its last release time r_i).

Analysis of P-EDF with Global Early-Release: From [22] we know that, with proper slack reclamation, the early-release of τ_i's next instance on its host core C_x does not introduce any *extra* workload on C_x and it will not cause any deadline miss for (especially high-criticality) tasks on C_x. Based on the results in [7], [26] and following the similar reasonings as in [22], when the next instance of task τ_i is temporarily migrated to another core C_y after reclaiming the slack time properly, no additional workload will be introduced to core C_y and there is no deadline misses as well. Therefore, for a given E-MC feasible task-to-core mapping under P-EDF, the global early-release scheme (as shown in Algorithm 1) can guarantee that there is no deadline miss for any task at runtime.

D. Multicore/Cache-Aware Migration-Constrained G-ER

As mentioned earlier, due to migration overheads, a low-criticality task is given preference to release its instances locally on its host core. However, when the reclaimable slack on the host core is not enough, Algorithm 1 does *not* address the problem of choosing the most appropriate core for migrating the low-criticality task.

Note that, due to different levels of caches shared between the cores on a multicore chip [23], the overheads for task

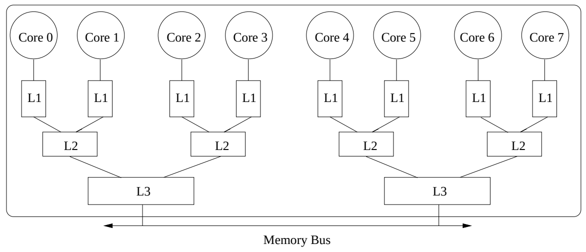


Fig. 4: An example cache architecture for a 8-core chip.

migrations between different cores can have quite big variations due to invalidation and repopulation of caches at different levels. For instance, Figure 4 shows an architecture for a 8-core chip, where two neighbor cores share an L2 cache, the first (last) four cores share L3 cache while all cores share the off-chip memory. Therefore, the overhead for migrating a task between two neighbor cores is relatively small compared to migration between other cores. In fact, the overhead of accessing higher-level caches increases exponentially (1-2 cycles to access L1 cache, 5-10 cycles for L2 cache, etc) [13].

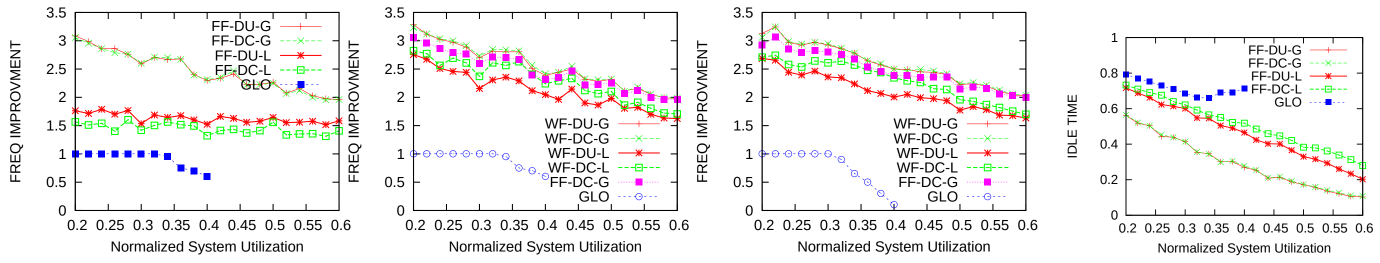
In this section, by specifically taking such overhead variations for task migrations into consideration, we propose the *multicore/cache-aware and migration-constrained (MAMC)* global early-release scheme. Our model assumes that the number of neighbor cores decreases with the closeness of the shared caches. For example, Figure 4 shows that each core has one core as its level-1 neighbor, two cores as its level-2 neighbors, four cores as its level-3 neighbors and so on. That is, in general, there are 2^{N-1} cores as a core's level-N neighbors and the overhead of migration is $\alpha^{N-1} \cdot X$. The MAMC global early-release scheme described below does not depend on this specific neighbor organization of cores and can be applied to any architecture of the cores.

Note that, once the multicore architecture is known and different levels of neighbors are identified for all cores, Algorithm 1 can be easily adapted to be a multicore-aware, cache-aware, and migration-constrained global early-release scheme for P-EDF. Specifically, a low-criticality task τ_i considers first the closest neighbors at lower levels for early-release (line 9), due to their low migration overheads for τ_i. For cores belonging to the same neighbor level, arbitrary orders can be used.

To incorporate the migration overheads into Algorithm 1, the amount of needed slack for global early-release S^G needs to be adjusted as $S^G = c_i^{LO} + 2 \cdot \alpha^{N-1} X$ for the level-N neighbor cores of C_x. For cases where the migration overheads are prohibitive, it is easy to limit task migrations between cores within a certain neighbor level.

VI. Evaluation and Discussion

In this section we evaluate the performance of the proposed global early-release technique for P-EDF on improving the service level of low-criticality tasks through extensive simulations. For comparison, in addition to the local early-release (that does not consider task migrations) for P-EDF, we also



a. Improvement w. FF; $m = 8$; b. Improvement w. WF; $m = 8$; c. Improvement w. WF; $m = 16$; d. Idle time w. FF; $m = 8$;

Fig. 5: Performance of the schemes with varying normalized system utilization for $prob(HI) = 0.5$; $prob(c_i^{LO}) = 0.9$;

implemented the Global EDF-VD (denoted as *GLO*), the state-of-the-art mixed-criticality multiprocessor scheduler [11]. For Global EDF-VD, whenever a high-criticality task τ_i takes more time than its C_i^{LO} and causes the system to enter high-level execution mode at runtime, it will discard all (current and future) low-criticality tasks until there is no active high-criticality task in the ready queue (at that time, the system can safely switch back to the low-level execution mode) [11].

E-MC tasks are generated following a similar approach in [11], as described in Section IV-B. In addition, the periods of tasks are generated between 50 and 200, following a uniform distribution. The high and low WCETs of tasks are obtained according to their utilizations. Considering the better schedulability of tasks under P-EDF, for low-criticality tasks, their maximum periods are set the same as their periods as discussed in Section IV-B. Moreover, we assume that each low-criticality task has 10 early-release points, which are uniformly distributed before its (maximum) period.

We use the average (execution) *frequency improvement* for low-criticality tasks as the performance metric [22]; the baseline is executing tasks according to their (maximum) periods with no early-release. For each data points, 100 task sets are generated and each task set runs to time $100,000$. Due to the close performance values for P-EDF with FF and BF, we omit the latter.

A. Effects of Normalized System Utilization

We evaluate the performance of the proposed local "*-L" and global "*-G" early-release schemes for P-EDF, and compare them against Global EDF-VD "GLO" for interesting values of normalized system utilization (i.e., $\frac{U^{bound}}{m} \in [0.2, 0.6]$). In Section IV-B, we showed that, when $\frac{U^{bound}}{m} > 0.6$, the schedulability of P-EDF decreases sharply with most of the partitioning heuristics.

We set $prob(HI) = 0.5$ (i.e., the numbers of low- and high-criticality tasks in a given task set are roughly the same) in this part of simulations. The effect of different task mixtures with varying $prob(HI)$ will be evaluated in next section. Moreover, considering the fact that high-criticality tasks rarely overrun and use their high-level WECTs, without being specified otherwise, we have the probability of these tasks taking their low-level WCETs as $prob(c^{LO}) = 0.9$.

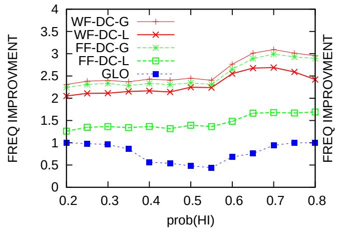
For 8-core systems (i.e., $m = 8$), Figure 5a shows the execution frequency improvements for low-criticality tasks, where the one corresponding to p_i^{max} (which actually equals to p_i in this work) is used as the baseline. To see the effects of different ordering of tasks, the figure contains only the results for P-EDF with FF heuristics in addition to that of Global EDF-VD. From the results, we can clearly see that the early-release techniques can significantly improve the execution frequencies (and thus the service levels) for low-criticality tasks under P-EDF.

When the normalized system load is low (e.g., $\frac{U^{bound}}{m} < 0.3$), the schedulability condition for Global EDF can be satisfied and Global EDF-VD will behave the same as Global EDF without canceling tasks during runtime [11]. In contrast, due to task cancellation in Global EDF-VD, the execution frequency of low-criticality tasks can be severely affected, reduced to around 50% on 8-core systems. Note that no result is shown for Global EDF-VD when $\frac{U^{bound}}{m} > 0.4$ since most task sets are not schedulable; see Section IV-B.

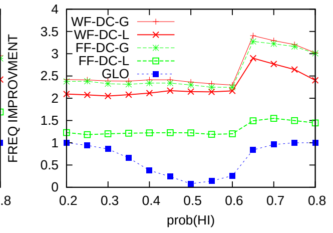
Moreover, for P-EDF, much better frequency improvements can be obtained for low-criticality tasks with the global early-release scheme when comparing to that of the local early-release scheme. The reason is that, by allowing low-criticality tasks to reclaim slack and migrating to other cores, the global early-release scheme provides more opportunities for such tasks to exploit slack time in the system and release their instances more frequently.

In addition, for the impact of different task orderings, DU for FF performs slightly better than DC when the local early-release is adopted. This is due to the uneven distribution of tasks among cores under FF with DC, where the "first" cores get most of the high-criticality tasks while other cores get all low-criticality tasks. For such cases, low-criticality tasks can still reclaim some slack that comes from the spare capacity on each core. However, the impact of different task orderings diminish when global early-release is adopted, because it allows tasks to migrate and reclaim slack on other cores.

Figure 5b shows the results for P-EDF with WF heuristics. For easy comparison, we show GLO and the global early-release under P-EDF with FF and DC. The performance difference between local and global early-release under P-EDF is much smaller because low- and high-criticality tasks are distributed among cores more evenly under WF (especially with DC ordering of tasks). Again, due to the ability of

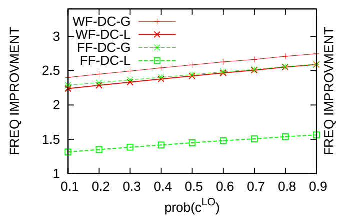


a. $m = 8$

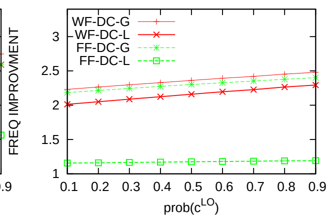


b. $m = 16$;

Fig. 6: Frequency improvement with varying $prob(HI)$ for $\frac{U^{bound}}{m} = 0.4$ and $prob(c^{LO}) = 0.9$.



a. $m = 8$;



b. $m = 16$;

Fig. 7: Frequency improvement with varying $prob(c^{LO})$ for $\frac{U^{bound}}{m} = 0.4$ and $prob(HI) = 0.5$.

reclaiming slack across cores with task migrations, for the global early-release scheme, the performance of P-EDF with WF for different task orderings is slightly better than that of FF heuristic.

For 16-core systems (i.e., $m = 16$), Figure 5c further shows the frequency improvement for low-criticality tasks under P-EDF with WF heuristics. The results follow the same trend as those for 8-core systems. However, notice the sharp decrease in performance for GLO, where most of low-criticality tasks may be cancelled when the normalized system utilization is 0.4. This is because systems with more cores have more high-criticality tasks for the same system utilization. Therefore, when tasks are executed under Global EDF-VD, it is more likely for the system to get into the high-level execution mode and canceling more low-criticality tasks.

Figure 5d further shows the percentage of idle time in 8-core systems under different schemes. Not surprisingly, there is more idle time under Global EDF-VD since there is no slack reclamation. Moreover, when low-criticality tasks are cancelled, there is even more idle time. For P-EDF, more slack time can be reclaimed in the global early-release scheme and there is less idle time in the system when compared to that of the local early-release scheme.

B. Effects of Task Mixtures with Varying $prob(HI)$

We evaluate the effect of task sets mix on the frequency of execution of low-criticality tasks (i.e., sensitivity analysis on $prob(HI)$). From last section, it can be seen that different orderings of tasks have quite limited impacts on the performance of P-EDF. Therefore, in the remaining experiments, we consider the DC ordering of tasks only, fixed $\frac{U^{bound}}{m} = 0.4$ and $prob(c^{LO}) = 0.9$. Figure 6ab show the frequency improvement of low-criticality tasks under P-EDF for systems with 8 and 16 cores, respectively.

We can see that, when there are more high-criticality tasks (i.e., larger $prob(HI)$), the performance of the local and global early-release schemes under P-EDF with different mapping heuristics increase slightly because more slack may be expected at runtime. Moreover, the global early-release scheme performs very closely with different mapping heuristics but always better than that of the local early-release scheme.

For Global EDF-VD, it is quite interesting to see that there is no task cancellation at low or high values of $prob(HI)$. The reason is that, when there are many more high- (or low-) criticality tasks in a task set, the schedulability condition of Global EDF can be satisfied and Global EDF-VD will act as Global EDF that will not cancel any task at runtime. However, when the numbers of high- (or low-) criticality tasks get close to each other (i.e., towards $prob(HI) = 0.5$), Global EDF-VD will rely on the virtual deadlines of high-criticality tasks and low-criticality tasks will be cancelled if necessary [11].

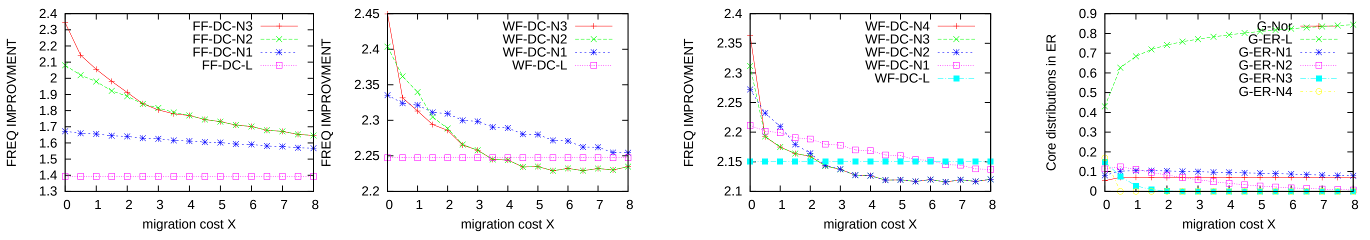
C. Effects of High-Criticality Tasks' Runtime Behaviors

With fixed $\frac{U^{bound}}{m} = 0.4$ and $prob(HI) = 0.5$, Figures 7a and 7b further show the performance of P-EDF with both local and global early-release schemes as the runtime behaviors of high-criticality tasks change (i.e., by varying $prob(c^{LO})$) for systems with 8 and 16 cores, respectively. Intuitively, when $prob(c^{LO})$ increases, more high-criticality tasks will take their low WCETs and more slack can be expected. With $\frac{U^{bound}}{m} = 0.4$ and $prob(HI) = 0.5$, the amount of slack generated by high-criticality tasks is quite limited and most slack reclaimed by low-criticality tasks actually comes from the spare capacity (60% on average) on each core. Therefore, P-EDF performs slightly better as $prob(c^{LO})$ increases.

D. Effects of Migration Overheads and Constraints

In this section, we further evaluate the impact of migration overheads on the performance of the global early-release scheme with various migration constraints under P-EDF. The overhead for migrating a task instance between different cores has been discussed in Section V-D. We fix $\alpha = 5$ and vary the value of X from 0 to 8 (i.e., from no overhead to high overhead). For comparison, the local early-release scheme (denoted as "*-L") that does not consider task migration is also included. Moreover, "*-Nk" denotes the scheme that allows tasks to migrate to no beyond its host's level-k neighbor cores.

For systems with 8 cores, Figure 8a first shows the performance of P-EDF with FF when different migration constraints are considered. With the model of neighbor cores considered in Section V-D, "*-N3" essentially represents the scheme that allows tasks to migrate globally. We can see that, when tasks have the opportunity to migrate to higher levels of neighbor cores and thus better chance for early release, the performance of P-EDF generally becomes better. However, such benefits decrease quickly as the migration overhead increases (recall



a. Freq. for FF; $m = 8$; b. Freq. for WF; $m = 8$; c. Freq. for WF; $m = 16$; d. Migration dist.; $m = 16$;

Fig. 8: The impacts of migration overheads on the global early-release; $U_{bound}/m = 0.4$, $prob(HI) = 0.5$, $prob(c^{LO}) = 0.9$;

the exponential increased overhead for migrating to higher level of neighbor cores).

In addition, when the WF task mapping heuristic is adopted, Figure 8b shows that the schemes that enable task migrations to higher levels of neighbor cores can even perform worse due to higher overhead costs. Similar results can be found for systems with 16 cores as shown in Figure 8c. When the migration overhead is high, the local early-release can performs better than the global peers.

To further investigate the behaviors of migrating tasks, Figure 8d shows where the low-criticality tasks are executed when the global early-release scheme is considered for P-EDF with WF in 16-core systems. G-Nor denotes the portion of low-criticality tasks that are released normally; G-ER-L denotes the ones released early on their host cores; G-ER-Nk denotes those released early on their host's level-k neighbor cores. As expected, when the migration overhead gets larger, the majority (more than 75%) of the task instance of low-criticality tasks will be released normally or early and executed on their host cores and the percentage of task instances executed on other cores is quite low.

VII. Conclusions

In existing mixed-criticality scheduling framework, low-criticality tasks will be cancelled to provide necessary computation capacity for high-criticality tasks in the worst case. To address such service interruption problem for low-criticality tasks, we have studied the *Elastic Mixed-Criticality (E-MC)* task model, where the minimum service requirements of low-criticality tasks are guaranteed by their maximum periods.

In this paper, we address the problem of scheduling *Elastic Mixed-Criticality (E-MC)* tasks in multicore systems. We investigate and empirically evaluate the schedulability of partitioned-EDF (P-EDF) by considering various task-to-core mapping heuristics. Then, *with* and *without* task migrations being considered, we propose both *global* and *local* early-release (ER) schemes to improve the service levels of low-criticality tasks at runtime. In addition, we explore the *multicore-aware and migration constrained* global-ER schemes by considering varied migration overheads between different cores. Our simulation results show that, compared to the state-of-the-art Global EDF-VD scheduler, P-EDF with various partitioning heuristics can lead to many more schedulable E-MC task sets. Moreover, our proposed global and local ER schemes can significantly improve the service levels of low-criticality tasks, while Global EDF-VD may severely and negatively affect them by canceling most of their task instances at runtime (especially for systems with more cores). Furthermore, by allowing task migrations, global-ER schemes can dramatically improve low-criticality tasks' service levels for partitionings that have *unbalanced* high- and low-criticality tasks on the cores.

References

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

[2] S.K. 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 *Proc. of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[3] S.K. Baruah, V. Bonifaci, G. DAngelo, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. Mixed-criticality scheduling of sporadic task systems. *Algorithms–ESA*, pages 555–566, 2011.

[4] S.K. Baruah, H. Li, and L. Stougie. Mixed-criticality scheduling: improved resource augmentation results. In *Proc. of the ICSA International Conference on Computers and their Applications (CATA)*, 2010.

[5] S.K. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Proc. of the IEEE 16th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010.

[6] S.K. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Proc. of the 20th Euromicro Conference on Real-Time Systems*, pages 147–155, 2008.

[7] 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 *Proc. of the 24th IEEE Real-Time Systems Symposium (RTSS)*, 2003.

[8] M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Trans. On Software Engineering*, 15(12):1497–1505, 1989.

[9] P. Ekberg and W. Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Proc. of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[10] N. Guan, P. Ekberg, M. Stigge, and W. Yi. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *Proc. of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, pages 13–23, 2011.

[11] H.Li and S.K. Baruah. Global mixed-criticality scheduling on multiprocessors. In *Proc. of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[12] O.R. Kelly, H. Aydin, and B. Zhao. On partitioned scheduling of fixed-priority mixed-criticality task sets. In *Proc. of the 8th IEEE International Conference on Embedded Software and Systems (ICESS)*, 2011.

[13] A. Kumar, R. Huggahalli, and S. Makineni. Characterization of direct cache access on multicore systems and 10gbe. In *Proc. of the IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, pages 341–352, 2009.

[14] K. Lakshmanan, D. De Niz, and R.R. Rajkumar. Mixed-criticality task synchronization in zero-slack scheduling. In *Proc. of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 47–56, 2011.

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

[16] H. Li and S.K. Baruah. Load-based schedulability analysis of certifiable mixed-criticality systems. In *Proc. of the 10th ACM international conference on Embedded software*, pages 99–108, 2010.

[17] M.S. Mollison, J.P. Erickson, J.H. Anderson, S.K. Baruah, and J.A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Proc. of the 10th IEEE International Conference on Computer and Information Technology (CIT)*, pages 1864–1871, 2010.

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

[19] F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp. In *Proc. of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[20] K.G. Shin and C.L. Meissner. Adaptation and graceful degradation of control system performance by task reallocation and period adjustment. In *Proc. of the Euromicro Conference on Real-Time Systems*, 1999.

[21] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Mixed critical earliest deadline first. Technical report, Technical Report TR-2012-22, Verimag Research Report, 2012.

[22] H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *Proc. of the Design, Automation and Test in Europe (DATE)*, 2013.

[23] J. J. Treibig, G. Hager, and G. Wellein. Multi-core architectures: Complexities of performance prediction and the impact of cache topology. Technical report, Regionales Rechenzentrum Erlangen, Friedrich-Alex ander Universiat Erlangen-urnberg, Erlangen, Germany, Oct. 2009.

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

[25] F. Zhang, K. Szwaykowska, W. Wolf, and V. Mooney. Task scheduling for control oriented requirements for cyber-physical systems. In *Proc. of the 29th IEEE Real-Time Systems Symposium (RTSS)*, 2008.

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