Technical Report: CS-TR-2013-007 (Updated version of CS-TR-2012-009), UTSA

Preference-Oriented Scheduling Framework for Periodic Real-Time Tasks

Prepared on May 15, 2013

Yifeng Guo, Hang Su, Dakai Zhu
University of Texas at San Antonio
San Antonio, TX, 78249
{yguo, hsu, dzhu}@cs.utsa.edu

Hakan Aydin
George Mason University
Fairfax, VA 22030
aydin@cs.gmu.edu

***Abstract*— We consider a set of real-time periodic tasks where some tasks are preferably executed *as soon as possible (ASAP)* and others *as late as possible (ALAP)* while still meeting their deadlines. After introducing the idea of *preference-oriented (PO) execution*, we formally define the concept of *PO-optimality*. For fully-loaded systems (with 100% utilization), we first propose a PO-optimal scheduler, namely *ASAP-ensured earliest deadline (SEED)*, by focusing on ASAP tasks where the optimality of tasks' ALAP preference is achieved *implicitly* due to the *harmonicity* of the PO-optimal schedules for such systems. Then, for under-utilized systems (with less than 100% utilization), we show the *discrepancies* between different PO-optimal schedules. By extending SEED, we propose a generalized *preference-oriented earliest deadline (POED)* scheduler that can obtain a PO-optimal schedule for any schedulable task set. We further evaluate the proposed PO-optimal schedulers through extensive simulations. The results show that, comparing to that of the well-known EDF scheduler, the scheduling overheads of SEED and POED are higher (but still manageable) due to the additional consideration of tasks' preferences. However, SEED and POED can achieve the preference-oriented execution objectives in a more successful way than EDF.**

I. Introduction

The real-time scheduling theory has been studied for decades and many well-known scheduling algorithms have been proposed for various task and system models. For instance, for a set of periodic tasks running on a uniprocessor system, the *rate monotonic (RM)* and *earliest-deadline-first (EDF)* scheduling policies are shown to be optimal for static and dynamic priority based preemptive scheduling algorithms, respectively [10]. With the main objective of meeting all the timing constraints, most existing scheduling algorithms (e.g., EDF and RM) prioritize and schedule tasks based only on their timing parameters (e.g., deadlines and periods). Moreover, these algorithms usually adopt the *work conservation* strategy (that is, the processor will not idle if there are tasks ready for execution) and execute tasks *as soon as possible (ASAP)*.

However, there are occasions where it can be beneficial to execute tasks *as late as possible (ALAP)*. For example, to provide better response time for soft aperiodic tasks, the *earliest deadline latest (EDL)* algorithm has been developed to execute periodic tasks at their *latest* times provided that all the deadlines will still be met [5]. By delaying the execution of all periodic tasks as much as possible, EDL has been shown to be optimal where no task will miss its deadline if the system utilization is no more than one [5]. By its very nature, EDL is a non-work-conserving scheduling algorithm: the processor may remain idle even though there are ready tasks. On the other hand, for fixed-priority based schemes, *dual-priority (DP)* scheduling was developed [6]. In that work, periodic tasks with hard deadlines start at lower priority levels and, to ensure that there is no deadline miss, their priorities are promoted to higher levels after a fixed time offset. Again, the main objective was to improve the responsiveness of soft aperiodic jobs, that are executed at the medium-priority level by default.

Such selectively delayed execution of tasks can be useful for fault-tolerant systems as well. For example, to minimize the overlap between the *primary* and *backup* tasks on different processors (and thus save energy), the execution of backup tasks should be delayed as much as possible [9], [14]. In fact, EDL has been exploited to schedule periodic backup tasks on the secondary processor to reduce the overlapped execution with their primary tasks for better energy savings [9].

Note that, the well-known scheduling algorithms generally treat all periodic tasks *uniformly*. They normally schedule tasks solely based on their timing parameters either at their earliest (e.g., with EDF and RM) or latest times (e.g., with EDL and DP). However, neither of them can effectively handle tasks with *different preferences*. For example, when backup tasks (whose primary tasks are on different processors) are scheduled with another set of primary periodic tasks in a mixed manner on one processor [8], [14], the execution of backup tasks needs to be postponed as much as possible while the primary tasks should be executed as soon as possible for better performance (our Appendix includes more detailed discussions). Moreover, for mixed-criticality task systems [1], [3], we may also give high-criticality tasks the preference of running early. This makes it possible to discover large amount of slack at earlier time, which could be further exploited to provide better service to low-criticality tasks [13].

Therefore, we believe that there is a strong incentive to develop effective scheduling algorithms for periodic tasks with different preferences. To the best of our knowledge, such algorithms have not been well studied in the literature yet. One may consider adopting the hierarchical scheduling approach to solve such problems, where tasks with the same preference form a task group and the high-level scheduler would deter-

mine only how to schedule different task groups [11], [12]. However, the existing hierarchical scheduling frameworks consider mostly work-conserving algorithms (such as EDF and RM) at both parent and child scheduling components, and it is not obvious how the framework can be generalized to non-work-conserving algorithms (such as EDL) and comply with tasks' different execution preferences. Hence, we focus on single-level scheduling algorithms in this work.

In this paper, for a set of periodic tasks running on a uniprocessor system where some tasks are preferably executed *as soon as possible (ASAP)* and others are preferably executed *as late as possible (ALAP)*, we introduce the concept of *preference-oriented execution* and propose corresponding optimal scheduling algorithms. Specifically, we first formally define *preference-oriented (PO) optimality* concept for periodic tasks with ASAP and ALAP preferences, and distinguish its two variants. We show the *harmonicity* of PO-optimal schedules for fully-loaded systems (with 100% utilization) where optimally attaining the ASAP preference of tasks implicitly indicates that the ALAP preference of other tasks is also optimally satisfied. However, for under-utilized systems (with utilization being less than 100%), there may exist *discrepancies* between different PO-optimal schedules.

Then, by taking tasks' ASAP preference into consideration when making scheduling decisions, we propose an optimal *ASAP-ensured earliest deadline (SEED)* scheduling algorithm for fully-loaded systems, where the optimality of tasks' ALAP preference is achieved *implicitly* due to the *harmonicity* of their PO-optimal schedules. Moreover, for under-utilized systems, by extending SEED and explicitly managing the spare capacity (i.e., idle time), we propose a generalized *preference-oriented earliest deadline (POED)* scheduling algorithm that can obtain a PO-optimal schedule for any schedulable task set. Finally, the evaluation results through extensive simulations show that, with manageable scheduling overheads (less than 35 microseconds per invocation for up to 100 tasks), the SEED and POED schedulers can obtain superior performance in terms of achieving tasks' preference objectives when comparing to that of the well-known EDF scheduler.

The remainder of this paper is organized as follows. Section II presents system models and preliminary notations. In Section III, we formally define the optimality of different preference-oriented schedules and show their harmonicity and discrepancies for fully-loaded and under-utilized systems, respectively. The optimal SEED scheduling algorithm for fully-loaded systems is proposed and analyzed in Section IV. The generalized POED scheduler is addressed in Section V. Section VI presents and discusses the evaluation results. Section VII concludes the paper.

II. Preliminaries

We consider a set of n periodic real-time tasks $\Psi = \{T_1, \ldots, T_n\}$ to be executed on a single processor system. Each task T_i is represented as a tuple (c_i, p_i), where c_i is its worst-case execution time (WCET) and p_i is its period. The utilization of a task T_i is defined as $u_i = \frac{c_i}{p_i}$. The system utilization of a given task set is the summation of all task utilizations: $U = \sum_{T_i \in \Psi} u_i$.

Tasks are assumed to have implicit deadlines. That is, the j^{th} task instance (or job) of T_i, denoted as $T_{i,j}$, arrives at time $(j-1) \cdot p_i$ and needs to complete its execution by its deadline at $j \cdot p_i$. Note that, a task has only one active task instance at any time. When there is no ambiguity, we use T_i to represent both the task and its current task instance.

In addition to its timing parameters, each task T_i in Ψ is assumed to have a *preference* to indicate how its task instances are ideally executed at run-time. The preference can be either *as soon as possible (ASAP)* or *as late as possible (ALAP)*. Hence, based on the preferences of tasks, we can partition them into two sets: Ψ_S and Ψ_L (where $\Psi = \Psi_S \cup \Psi_L$), which contain the tasks with ASAP and ALAP preferences, respectively.

A *schedule* of tasks essentially shows *when* to execute *which* task. We consider discrete-time schedules. More formally, a schedule $\mathcal{S}$ is defined as:

$$\mathcal{S} : t \rightarrow T_i$$

where $0 \leq t$ and $1 \leq i \leq n$. If a task T_i is executed in time slot $[t, t+1)$ in the schedule $\mathcal{S}$, we have $\mathcal{S}(t) = T_i$. Furthermore, a *feasible* schedule is defined as the one where no task instance misses its deadline [10].

We focus on with dynamic priority-based scheduling algorithms in this work. Note that, if Ψ_L is empty (i.e., no task has ALAP preference), we can simply adopt the EDF scheduler to optimally execute all ASAP tasks [10]. Similarly, when $\Psi_S = \emptyset$ (i.e., no task has ASAP preference), all tasks in Ψ can be optimally scheduled with the EDL algorithm [5].

In this paper, we consider the cases where Ψ consists of tasks with different preferences (i.e., both Ψ_S and Ψ_L are non-empty). For such cases, both EDF and EDL can still *feasibly* schedule the tasks in Ψ as long as $U \leq 1$ [5], [10]. However, without taking tasks' preferences into consideration, neither of them can effectively address the different preference requirements of various tasks.

III. Optimal Preference-Oriented Schedules

Before discussing the proposed scheduling algorithms for tasks with ASAP and ALAP preferences, in this section, we first formally define the optimality of different preference-oriented schedules and investigate their relationships. Considering the periodicity of the problem, we focus on the schedule of tasks within the *LCM (least common multiple)* of their periods. Intuitively, in an optimal preference-oriented schedule, **a.) tasks with ASAP preference should be executed before the ones with ALAP preference whenever possible;** and **b.) the execution of ALAP tasks should be delayed as much as possible without causing any deadline miss.**

To quantify the early execution of ASAP tasks in Ψ_S in a feasible schedule $\mathcal{S}$, the *accumulated ASAP execution* at any time t ($0 \leq t \leq LCM$) is defined as the total amount of

execution time of ASAP tasks from time 0 to time t in the schedule $\mathcal{S}$, which is denoted as $\Delta(\mathcal{S}, t)$. Formally, we have

$$\Delta(\mathcal{S}, t) = \sum_{z=0}^{t} \delta(\mathcal{S}, z) \tag{1}$$

where $\delta(\mathcal{S}, z) = 1$ if $\mathcal{S}(z) = T_i$ and $T_i \in \Psi_S$; otherwise, $\delta(\mathcal{S}, z) = 0$.

Similarly, the *accumulated ALAP execution* of tasks in Ψ_L is defined as the total amount of execution time of Ψ_L's tasks from time t to LCM in a feasible schedule $\mathcal{S}$ and is denoted as $\Omega(\mathcal{S}, t)$. Formally,

$$\Omega(\mathcal{S}, t) = \sum_{z=t}^{LCM-1} \omega(\mathcal{S}, z) \tag{2}$$

where $\omega(\mathcal{S}, z) = 1$ if $\mathcal{S}(z) = T_i$ and $T_i \in \Psi_L$; otherwise, $\omega(\mathcal{S}, z) = 0$.

When *only* ASAP or ALAP tasks are of interest in a given task set, we first define the ASAP and ALAP optimalities of a schedule based on the above notations.

*Definition 1 (***ASAP-optimality):** A feasible schedule $\mathcal{S}^{opt}_{asap}$ is ASAP-optimal if, for any other feasible schedule $\mathcal{S}$, $\Delta(\mathcal{S}^{opt}_{asap}, t) \geq \Delta(\mathcal{S}, t)$ at any time t ($0 \leq t \leq LCM$).

*Definition 2 (***ALAP-optimality):** A feasible schedule $\mathcal{S}^{opt}_{alap}$ is ALAP-optimal if, for any other feasible schedule $\mathcal{S}$, $\Omega(\mathcal{S}^{opt}_{alap}, t) \geq \Omega(\mathcal{S}, t)$ at any time t ($0 \leq t \leq LCM$).

Since it is possible to have conflicting demands from ASAP and ALAP tasks (as discussed below), when considering the preference requirements of all tasks, we define the *preference-oriented (PO) optimality* of a schedule as follows.

*Definition 3 (***PO-optimality):** A feasible schedule $\mathcal{S}^{opt}$ is PO-optimal if, at any time t ($0 \leq t \leq LCM$),

- a) $\Omega(\mathcal{S}^{opt}, t) \geq \Omega(\mathcal{S}^{opt}_{asap}, t)$ holds, where both $\mathcal{S}^{opt}$ and $\mathcal{S}^{opt}_{asap}$ are ASAP-optimal (denoted as **PO^S-optimal**); or
- b) $\Delta(\mathcal{S}^{opt}, t) \geq \Delta(\mathcal{S}^{opt}_{alap}, t)$ holds, where both $\mathcal{S}^{opt}$ and $\mathcal{S}^{opt}_{alap}$ are ALAP-optimal (denoted as **PO^L-optimal**).

Note that PO-optimal schedules are defined based on the accumulated executions of ASAP and ALAP tasks without distinguishing the execution orders of tasks with the same preference. That is, when determining the optimality of a feasible schedule, we essentially divide the schedule into a sequence of ASAP and ALAP execution sections. Hence, provided that there is no deadline miss, switching the execution order of some task instances with the same preference in their execution sections will not affect the optimality of a feasible schedule. Therefore, as shown later, more than one optimal schedule may exist for a set of periodic tasks with ASAP and ALAP preferences.

Moreover, the optimal schedules highly depend on the system utilization of a given task set. In what follows, we investigate the relationship between different optimal schedules of tasks with ASAP and ALAP preferences based on system utilization. This investigation gives a *foundation* for the preference-oriented execution framework and provides insightful guidelines to develop optimal preference-oriented schedulers as shown later.

A. Harmonious Optimal Schedules: Fully-Loaded Systems

When the system utilization of a task set is $U = 1$, we know that the processor will be fully loaded and there is no idle time in any feasible schedule [10]. Therefore, if a feasible schedule $\mathcal{S}$ is an ASAP-optimal schedule (i.e., the execution of tasks with ASAP preference in Ψ_S is performed at their earliest possible time), it also implies that the execution of tasks with ALAP preference in Ψ_L has been maximally delayed at any time instance. Therefore, the feasible schedule $\mathcal{S}$ is an ALAP-optimal schedule as well. More formally, we can have the following lemma.

Lemma 1: For a set of periodic tasks with ASAP and ALAP preferences where the system utilization is $U = 1$, if a feasible schedule $\mathcal{S}^{opt}$ is an ASAP-optimal schedule, it is also an ALAP-optimal schedule. That is, $\mathcal{S}^{opt}$ is both PO^S-optimal and PO^L-optimal. Hence, $\mathcal{S}^{opt}$ is a PO-optimal schedule for the task set under consideration.

Proof: When the system utilization $U = 1$, we know that the system is fully loaded and there is no idle time in the schedule $\mathcal{S}^{opt}$. Therefore, for any time t ($0 \leq t \leq LCM$), the overall execution time for tasks in Ψ_L from time 0 to t in the schedule $\mathcal{S}^{opt}$ can be found as $(t - \Delta(\mathcal{S}^{opt}, t))$, where $\Delta(\mathcal{S}^{opt}, t)$ represents the accumulated execution time for tasks in Ψ_S from time 0 to t.

Note that, for a given task set, the total execution time for tasks with ALAP preference in Ψ_L within a LCM is fixed, which can be denoted as t^{total}_{alap}. Thus, the accumulated execution time for ALAP tasks in Ψ_L from time t to LCM in any feasible schedule $\mathcal{S}$ can be found as:

$$\Omega(\mathcal{S}, t) = t^{total}_{alap} - (t - \Delta(\mathcal{S}, t))$$

Since $\mathcal{S}^{opt}$ is also a feasible schedule, we have:

$$\Omega(\mathcal{S}^{opt}, t) = t^{total}_{alap} - (t - \Delta(\mathcal{S}^{opt}, t))$$

As $\mathcal{S}^{opt}$ is an ASAP-optimal schedule, from Definition 1, for any feasible schedule $\mathcal{S}$, we have $\Delta(\mathcal{S}^{opt}, t) \geq \Delta(\mathcal{S}, t)$. Therefore, from the above equations, we can get:

$$\Omega(\mathcal{S}^{opt}, t) \geq t^{total}_{alap} - (t - \Delta(\mathcal{S}, t)) = \Omega(\mathcal{S}, t)$$

From Definition 2, we know that $\mathcal{S}^{opt}$ is also an ALAP-optimal schedule. Therefore, from Definition 3, $\mathcal{S}^{opt}$ is a PO-optimal (essentially both PO^S-optimal and PO^L-optimal) schedule for the task set under consideration. This concludes the proof. ■

B. Discrepant Optimal Schedules: Under-Utilized Systems

For task sets with system utilization $U < 1$, the processor will not be fully loaded and there will be idle intervals in any feasible schedule. However, the conflicting requirements of ASAP and ALAP tasks make the *distribution* of these intervals an intriguing problem. Intuitively, for ASAP tasks in Ψ_S, such idle intervals should appear as late as possible; whereas for ALAP tasks in Ψ_L, they should appear as early as possible in a feasible schedule.

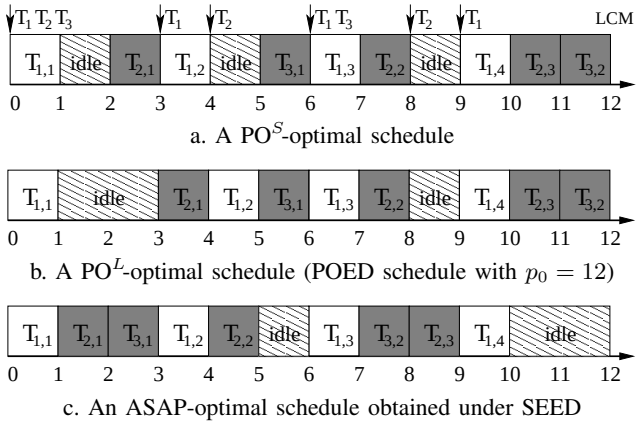


Fig. 1. An example task system with discrepant PO-optimal schedules; Here, $\Psi = \{T_1(1,3),\ T_2(1,4),\ T_3(1,6)\}$; $\Psi_S = \{T_1\}$ and $\Psi_L = \{T_2, T_3\}$.

To illustrate the *discrepancies* between PO^S-optimal and PO^L-optimal schedules for task systems with $U < 1$, we consider an example task set with three tasks, where $T_1 = (1,3)$, $T_2 = (1,4)$ and $T_3 = (1,6)$. Here, task T_1 has ASAP preference while T_2 and T_3 have ALAP preference. That is, $\Psi_S = \{T_1\}$ and $\Psi_L = \{T_2, T_3\}$. It can be easily found that the system utilization is $U = 0.75$ and the least common multiple of all tasks' periods is $LCM = 12$. Therefore, for any feasible schedule within LCM, the amount of idle time can be found as $(1-U) \cdot LCM = (1 - 0.75) \cdot 12 = 3$.

First, for the schedule in Figure 1a, we can see that all instances of the ASAP task T_1 are executed *right after* their arrival times. That is, it is an ASAP-optimal schedule. Moreover, for all possible executions of the ALAP tasks T_2 and T_3 in ASAP-optimal schedules, the one as shown in Figure 1a has been *maximally* delayed with most of T_2 and T_3's instances are executed right before their deadlines. It turns out that it is actually a PO^S-optimal schedule.

Note that, the schedule in Figure 1a is <u>not</u> ALAP-optimal. By further delaying the execution of task T_2's first instance $T_{2,1}$ for one more unit, we can obtain another feasible schedule as shown in Figure 1b, which turns out to be another PO-optimal (specifically, PO^L-optimal) schedule.

Here, we can see that there are *discrepancies* with the execution of ASAP and ALAP tasks during the interval $[2, 5)$ between different PO-optimal schedules. Such discrepancies come from the *conflicting* demands from the ASAP task T_1 and ALAP task T_2, where both of their active instances at time 3 ideally should be executed in time slot $[3, 4)$ to optimally satisfy preferences, respectively.

Therefore, for under-utilized systems, it is possible to have discrepant PO-optimal schedules due to the conflicting demands of ASAP and ALAP tasks for their executions and thus their conflicting requirements for the idle times in feasible schedules. We now state this observation.

Remark 1: For a set of periodic tasks with ASAP and ALAP preferences, if the system is under-utilized with $U < 1$, there may exist *discrepancies* between the execution of ASAP and ALAP tasks in different PO-optimal (i.e., PO^S-optimal and PO^L-optimal) schedules.

IV. An Optimal Scheduler for Systems with $U = 1$

Intuitively, when designing preference-oriented scheduling algorithms, there are two basic principles to address the preference requirements of ASAP and ALAP tasks, respectively.

- **P1 (ASAP Scheduling Principle):** at any time t, if there are ready ASAP tasks in Ψ_S, the scheduler should not let the processor idle – however, it may have to first execute some ALAP tasks in Ψ_L to meet their deadlines.
- **P2 (ALAP Scheduling Principle):** at any time t, if all ready tasks belong to Ψ_L, the scheduler should not execute these tasks and let the processor stay idle if it is possible to do so without causing any deadline miss for current and future task instances.

These two principles can have conflicts at run time (from Remark 1) and a scheduler may have to choose to favor one over the other. However, for fully-loaded systems, we know that their PO-optimal schedules are harmonious (see Lemma 1). Hence, if the ASAP scheduling principle is *fully* complied with when scheduling tasks in such systems, it means that the ALAP scheduling principle is (implicitly) respected as well. Therefore, by focusing on ASAP tasks and adhering to the first principle, we first propose in this section an optimal preference-oriented scheduling algorithm, namely *A<u>S</u>AP-<u>E</u>nsured <u>E</u>arliest <u>D</u>eadline (SEED)*, for fully-loaded systems. In Section V, by explicitly taking both ASAP and ALAP scheduling principles into consideration, a generalized preference-oriented scheduler is devised, which can obtain a PO-optimal schedule for any schedulable task set.

A. SEED Scheduling Algorithm

SEED is a dynamic priority scheduling algorithm where tasks with earlier deadlines have in general higher priorities (when there is a tie, the task with smaller index has higher priority). However, instead of scheduling the tasks *solely* based on their deadlines, SEED puts tasks with ASAP preference in the center stage when making scheduling decisions to *fully* comply with the ASAP scheduling principle.

The main steps of SEED are summarized in Algorithm 1, which can be invoked on different occasions: a) a new task arrives; b) the current task completes or is preempted. At any invocation time t, we use two ready queues $\mathcal{Q}_S(t)$ and $\mathcal{Q}_L(t)$ to manage active ASAP and ALAP tasks, respectively.

Algorithm 1 The SEED Scheduling Algorithm

1: //The invocation time of the algorithm is denoted as t.
2: **Input:** $\mathcal{Q}_S(t)$ and $\mathcal{Q}_L(t)$;
3: **if** ($\mathcal{Q}_S(t) == \emptyset$ OR $\mathcal{Q}_L(t) == \emptyset$) **then**
4: **if** ($\mathcal{Q}_S(t)! = \emptyset$) **then**
5: T_k =*Dequeue*($\mathcal{Q}_S(t)$) and execute T_k;
6: **else if** ($\mathcal{Q}_L(t)! = \emptyset$) **then**
7: T_l =*Dequeue*($\mathcal{Q}_L(t)$) and execute T_l;
8: **else**
9: Let CPU idle; //$\mathcal{Q}_S(t) = \mathcal{Q}_L(t) = \emptyset$;
10: **end if**
11: **else if** ($d_k > d_l$) **then**
12: //T_k =*Header*($\mathcal{Q}_S(t)$) and T_l =*Header*($\mathcal{Q}_L(t)$);
13: Construct the look-ahead queue $\mathcal{Q}_{la}$ for interval $[t, d_k]$;
14: *Mark*$(t, d_k, \mathcal{Q}_{la})$;//determine reserved sections in $[t, d_k]$;
//Suppose the first *"reserved"/"free"* section ends at t';
15: **if** ($[t, t']$ is marked as *"reserved"*) **then**
16: T_l =*Dequeue*($\mathcal{Q}_L(t)$) and execute T_l;
17: **else**
18: T_k =*Dequeue*($\mathcal{Q}_S(t)$) and execute T_k;
19: **end if**
20: **else**
21: T_k =*Dequeue*($\mathcal{Q}_S(t)$) and execute T_k;
22: **end if**

Algorithm 2 The function *Mark*$(t, d_k, \mathcal{Q}_{la})$

1: **Input:** $[t, d_k]$, the look-ahead interval; $\mathcal{Q}_{la}$, the queue of task instances in $\Psi_{la}(t, d_k)$ with decreasing priority order;
2: **while** ($\mathcal{Q}_{la} \neq \emptyset$) **do**
3: T_i = *Dequeue*($\mathcal{Q}_{la}$);//T_i has the highest priority with d_i
4: //Suppose the (remaining) execution time of T_i is c_i;
5: **if** ($T_i \in \Psi_L$) **then**
6: For the free sections in $[t, d_i]$, in the reverse order of their appearance, mark them as *"reserved"*, where the marked sections have the length of c_i;
7: **else**
8: //Suppose T_i arrives at time a_i (after time t); and
9: //the total length of free sections in $[a_i, d_i]$ is L;
10: **if** ($c_i \leq L$) **then**
11: Mark the free sections in $[a_i, d_i]$ as *"reserved"*, where the marked sections have the length of c_i;
12: **else**
13: Mark *all* free sections in $[a_i, d_i]$ as *"reserved"*;
14: For the free sections in $[t, a_i]$, in the reverse order of their appearance, mark them as *"reserved"*, where marked sections have the length of $(c_i - L)$;
15: **end if**
16: **end if**
17: **end while**

Recall that, from the definitions in Section III, the optimality of a feasible schedule for a given set of periodic tasks with ASAP and ALAP preferences depends on only the accumulated executions of such tasks rather than when each individual task is executed. Therefore, tasks in both queues are ordered and processed in the decreasing order of their priorities. We assume that SEED is invoked after newly arrived tasks are added to their corresponding queues at any time t (which is not shown for brevity).

For fully-loaded systems, it is not possible to have both ready queues be empty when SEED is invoked. However, for the discussion later on applying SEED to under-utilized systems, such a case is included (line 9) when there is no active task and CPU should be idle. If there is only one empty ready queue, it means that all active tasks have either ASAP or ALAP preference and there is no conflicting requirement at time t. For such cases, the active task with the earliest deadline is executed (lines 5 and 7).

The complicated case comes when there are both active ASAP and ALAP tasks. Here, according to the ASAP scheduling principle, SEED should execute first the highest priority task T_k in $\mathcal{Q}_S(t)$ whenever possible. If T_k's deadline is no later than that of $\mathcal{Q}_L(t)$'s header task, T_k can be executed immediately (line 21). Otherwise, to find out whether T_k can be executed at time t without causing any deadline miss, as the **centerpiece** of the SEED scheduler, the handling of this special case has the following steps.

First, we determine the *look-ahead interval* as $[t, d_k]$, where d_k is T_k's current deadline. Note that, to meet its deadline, the (remaining) execution of T_k has to be performed within the interval $[t, d_k]$. Moreover, at/after time t, only the task instances (including the future arrivals) that have higher priorities than T_k may execute before d_k and affect T_k's execution. As the second step, we find these task instances that form a *look-ahead set* $\Psi_{la}(t, d_k)$; and, in the order of their priorities, put them into a look-ahead queue $\mathcal{Q}_{la}$ (line 13). More formally, $\Psi_{la}(t, d_k)$ is defined as:

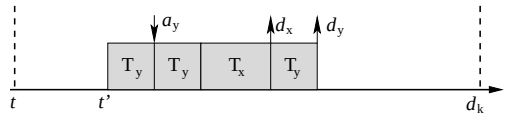
$$\Psi_{la}(t, d_k) = \{T_{i,j} | (T_{i,j} \in \mathcal{Q}_L(t) \vee a_{i,j} > t) \wedge d_{i,j} < d_k\} \quad (3)$$

where $a_{i,j}$ is the arrival time of a future task instance $T_{i,j}$. That is, $\Psi_{la}(t, d_k)$ includes both the *active* ALAP tasks in $\mathcal{Q}_L(t)$ and *future* task instances that have *earlier* deadlines than d_k. Essentially, $\Psi_{la}(t, d_k)$ contains all task instances that can prevent T_k from being executed immediately at time t.

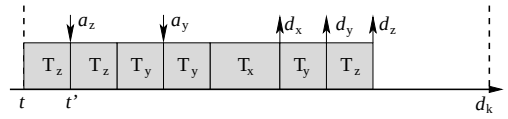
Then, for the look-ahead interval $[t, d_k]$, the *reserved* sections for task instances in $\Psi_{la}(t, d_k)$ are determined with the help of the function *Mark*$(t, d_k, \mathcal{Q}_{la})$ (line 14). There are two possibilities for the result as illustrated in Figure 2. If the first section $[t, t']$ is marked as *"reserved"*, it means that some active ALAP tasks have to be executed immediately to avoid deadline misses (line 16). Otherwise, T_k can be executed right away at time t (line 18). Note that, the execution of T_k: a) may complete or be preempted due to the arrival of a new task instance before time t'; or b) has to stop at time t' due to the timing constraints of other task instances.

Algorithm 2 further details the steps of *Mark*$(t, d_k, \mathcal{Q}_{la})$. Again, the objective of this function is to determine whether it is possible to execute task T_k at time t. Thus, we just need to find out the location of the reserved sections rather than to generate the schedule for the task instances in $\mathcal{Q}_{la}$

within the interval $[t, d_k]$. Therefore, in decreasing order of their priorities, the task instances in $\mathcal{Q}_{la}$ are handled one at a time as discussed below (lines 2 and 3).



a. $\mathcal{Q}_{la} = \{T_x, T_y\}$, where the first section is *"free"*;



b.$\mathcal{Q}_{la} = \{T_x, T_y, T_z\}$ and the first section is *"reserved"*;

Fig. 2. The marking of the look-ahead interval.

If T_i is an ALAP task instance, in the *backward* order, we mark the free sections before d_i as *"reserved"*. Here, a free section may be divided into pieces and the total length of the marked sections should equal to T_i's (remaining) execution time c_i (line 6). If T_i is a future task instance, the backward marking process may use free sections before its arrival time a_i. Note that this does not mean that we need to execute a task instance before its arrival, but merely indicates that the marked sections before a_i have to be reserved for the task instances in $\mathcal{Q}_{la}$.

As an example, suppose that there are two ALAP task instances in $\mathcal{Q}_{la}$, where T_x has an earlier deadline and T_y is a future task instance that arrives at time $a_y (> t)$. Since T_x has a higher priority, it first marks the section of size c_x right before its deadline d_x as *"reserved"* as shown in Figure 2a. Then, for T_y, its execution time c_y is larger than the free section within $[a_y, d_y]$. In this case, it will first mark the free section within $[a_y, d_y]$ and then part of the free section before a_y as *"reserved"*. As there is no other task instance in $\mathcal{Q}_{la}$, the first section $[t, t']$ is left as *"free"*. Therefore, even though its deadline d_k is later than that of the ALAP task T_x, T_k can be executed right away at time t (up to the time point t').

If the next task instance T_i in $\mathcal{Q}_{la}$ is an ASAP task, it must arrive after time t and have its deadline before d_k (i.e., $a_i > t$ and $d_i < d_k$). For the free sections within $[a_i, d_i]$, if their overall size L is no smaller than T_i's execution time c_i, we mark them as *"reserved"* in the *forward* order such that the marked sections have the total length of c_i (line 11). Otherwise, all the free sections within $[a_i, d_i]$ will be marked as *"reserved"* (line 13). Then, similar to the handling of ALAP tasks, the free sections before a_i will be reserved in the backward order for the amount of $c_i - L$ (line 14).

Continuing with the example in Figure 2a, suppose that there is one more ASAP task instance (T_z) in $\mathcal{Q}_{la}$, where $d_y < d_z$. As the free section within $[a_z, d_z]$ is not large enough, it turns out that T_z marks all free sections before d_z as shown in Figure 2b, where the first section $[t, a_z]$ is *"reserved"*. That is, to guarantee that there is no deadline miss for the task instances in $\mathcal{Q}_{la}$, we have to execute T_x (and even T_y) immediately at time t. However, such urgent execution will be preempted when a new task T_z arrives at the nearest future time a_z by re-invoking the SEED scheduler.

B. Optimality of the SEED Scheduler

In this section, we provide formal analysis and proof for the optimality of the SEED scheduling algorithm. Specifically, we first show that, for any schedulable task set with system utilization $U \leq 1$, the SEED scheduler can successfully schedule all tasks and guarantee that there is no deadline miss. Then, we prove that, for any schedulable task set, SEED will generate an ASAP-optimal schedule. This further implies that, for fully-loaded task systems, SEED is essentially an optimal preference-oriented scheduler.

From Algorithm 1, we can see that SEED follows the earliest deadline first (EDF) principle when scheduling tasks with the same preference. Specifically, where all active tasks have the same preference, the task with the earliest deadline will be executed (lines 5 and 7 for ASAP and ALAP tasks, respectively). For cases where active tasks have different preferences, the look-ahead interval is determined by an earliest deadline ASAP task. Therefore, if the initial part of the look-ahead interval is *"free"*, the earliest deadline ASAP task is executed (line 18); otherwise, if the initial part is *"reserved"*, the earliest deadline ALAP task will be executed (line 16). Hence, we can have the following lemma:

Lemma 2: At any time t, the SEED scheduler executes tasks with the same preference according to the earliest deadline first (EDF) principle. That is, whenever SEED executes an ASAP (or ALAP) task, the task should have the earliest deadline among all active ASAP (or ALAP) tasks.

Hence, before a task T_k completes its execution, no other task with the *same* preference but a later deadline can be executed within the interval $[r_k, d_k]$, where r_k and d_k are T_k's arrival time and deadline, respectively. From Algorithm 1, we can further get the following lemma:

Lemma 3: Suppose that a task T_k misses its deadline at time d_k, no task that has a deadline later than d_k can be executed within $[r_k, d_k]$ under SEED.

Proof: If T_k is an ASAP task, from Lemma 2, we know that no ASAP task with a deadline later than d_k can be executed within $[r_k, d_k]$. Moreover, from Algorithm 1, we know that no ALAP task with a deadline later than d_k will be in the look-ahead task queue $\mathcal{Q}_{la}$ when SEED is invoked at time t, where ($r_k \leq t \leq d_k$). Therefore, no task with a deadline later than d_k can be executed within $[r_k, d_k]$ when T_k is an ASAP task.

When T_k is an ALAP task, from Lemma 2, we know that no ALAP task with a deadline later than d_k can be executed within $[r_k, d_k]$. Moreover, from Algorithms 1 and 2, we know that the execution of any ASAP task with a deadline later than d_k within $[r_k, d_k]$ would indicate that enough time has been *reserved* for task T_k before d_k, which contradicts with

our assumption that T_k misses its deadline. Therefore, no task with a deadline later than d_k can be executed within $[r_k, d_k]$ when T_k is an ALAP task.

To conclude, if a task T_k misses its deadline at time d_k, no task (regardless of its preference) that has a deadline later than d_k can be executed within $[r_k, d_k]$ under SEED.

■

From Lemma 3 and Algorithms 1 and 2, we can get the following theorem regarding to the schedulability of tasks under SEED:

Theorem 1: For a set of periodic tasks with ASAP and ALAP preferences where $U \leq 1$, the SEED scheduler can successfully schedule all tasks without missing any deadline.

Proof: Suppose that a job J_k arrives at time r_k and misses its deadline at d_k. From Lemma 3, we know that there is no job with a deadline later than d_k can be executed within the interval $[r_k, d_k]$, which is defined as the *problematic interval*.

Let t_0 denote the last processor idle time before d_k. Note that, there must exist jobs with deadlines later than d_k that are executed before r_k. Otherwise, we can find that the *processor demand* in $[t_0, d_k]$, defined as the sum of the computation times of all jobs that arrive no earlier than t_0 and have deadlines no later than d_k [2], is more than $(d_k - t_0)$, which contradicts with the condition of $U \leq 1$.

Moreover, there must exist jobs that arrives before r_k with deadlines earlier than d_k and are executed in $[r_k, d_k]$ (otherwise, there will be a contradiction for the processor demand within the interval $[r_k, d_k]$). Suppose r_0 is the earliest arrival time of such jobs, we can extend backward our problematic interval to be $[r_0, d_k]$.

Following the above steps, we can finally extend our problematic interval to be $[t_0, d_k]$, which indicates that there is no job with a deadline later than d_k that has been executed before r_k. This contradicts with our earlier findings that there must exist jobs with deadlines later than d_k that are executed before r_k, and thus concludes the proof.

■

Theorem 2: For a set of periodic tasks with ASAP and ALAP preferences where the system utilization is $U \leq 1$, the generated schedule under SEED is an ASAP-optimal schedule and SEED is an ASAP-optimal scheduler.

Proof: Suppose that the schedule $\mathcal{S}_{seed}$ obtained under SEED for the tasks being considered is **not** an ASAP-optimal schedule. There must exist another feasible schedule $\mathcal{S}$ such that $\Delta(\mathcal{S}, t) \geq \Delta(\mathcal{S}_{seed}, t)$ $(0 \leq t \leq LCM)$. Moreover, there must exist at least one interval during which ASAP tasks are executed in $\mathcal{S}$ but not in $\mathcal{S}_{seed}$. Assume $[t_1, t_2]$ $(0 \leq t_1 < t_2 \leq LCM)$ is the first of such intervals. That is, during the interval $[0, t_1]$, $\mathcal{S}$ and $\mathcal{S}_{seed}$ must execute ASAP tasks for the same amount and at the same time.

As there are active ASAP tasks during $[t_1, t_2]$, from Algorithm 1, we know that SEED must have executed ALAP tasks during $[t_1, t_2]$ and such ALAP tasks (which form a set Φ) have to be executed during $[t_1, t_2]$ to meet their deadlines. Since SEED is a work-conserving scheduler and it executes ALAP tasks in the order of their deadlines, the total amount of execution time for ALAP tasks in Φ during $[0, t_1]$ in the schedule $\mathcal{S}$ will be no more than that of $\mathcal{S}_{seed}$. Therefore, such ALAP tasks in Φ have to be executed during $[t_1, t_2]$ in the schedule $\mathcal{S}$ as well to meet their deadlines, which contradicts with our assumption and thus concludes the proof.

■

From Theorem 2 and Lemma 1, for systems with $U = 1$, SEED is essentially an optimal preference-oriented scheduler. Thus, we have the following theorem.

Theorem 3: For a fully loaded set of periodic tasks with ASAP and ALAP preferences where $U = 1$, SEED is the optimal preference-oriented scheduler and the generated SEED schedule is an optimal preference-oriented schedule.

C. Improved SEED Algorithm and Its Complexity

From Algorithms 1 and 2, we can see that the most complex case happens when the deadline of the highest priority ASAP task is later than that of the highest priority ALAP task. To determine whether it is possible to first execute the ASAP task and comply with the ASAP scheduling principle, SEED needs to consider all (active and future) task instances within the look-ahead interval. However, the *Mark()* function in Algorithm 2 is computationally costly by requiring every task instance in $\mathcal{Q}_{la}$ to search through the look-ahead interval and mark all corresponding reserved sections.

Note that, except the first (free/reserved) section, SEED does not need the detailed information about other sections within the look-ahead interval. Essentially, the only information that SEED needs is how much time (if any) it can use to *safely* execute the the highest priority ASAP task T_k at the invocation time t without causing any deadline miss in the future.

From the discussion of Algorithm 2, we know that, when the first section is "reserved", it indicates there is no available time for task T_k at time t. In this case, there must exist at least one task instance $T_x \in \mathcal{Q}_{la}$ for which there is no free section between t and T_x's deadline d_x. That is, the *accumulated workload* $W(d_x, \mathcal{Q}_{la})$ for task instances in $\mathcal{Q}_{la}$ that has to be done before d_x is $(d_x - t)$, where

$$W(d_x, \mathcal{Q}_{la}) = \sum_{T_i \in \mathcal{Q}_{la} \wedge d_i \leq d_x} c_i \tag{4}$$

Otherwise, the size of the first "free" section can be found as

$$t^{free} = \min\{(d_x - t) - W(d_x, \mathcal{Q}_{la}) | \forall T_x \in \mathcal{Q}_{la}\} \tag{5}$$

Therefore, based on the above two equations, the process of determining the status of the first section can be simplified. Here, $t^{free} = 0$ indicates the first section is "reserved", while $t^{free} > 0$ represents the size of the first "free" section.

Suppose that the minimum and maximum periods of tasks are p_{min} and p_{max}, respectively. In the worst case, the look-ahead interval can be as large as p_{max}. Moreover, the worst case number of task instances in $\mathcal{Q}_{la}$ can be found as $n \cdot \frac{p_{max}}{p_{min}}$. Hence, by checking the accumulated workload that has to be done before the deadline of each task instance in $\mathcal{Q}_{la}$, t^{free}

can be found in $\mathbf{O}\left(n \cdot \frac{p_{max}}{p_{min}}\right)$, which is also the complexity of the SEED scheduler.

D. SEED for Under-Utilized Systems

Although SEED can generate a PO-optimal schedule for any fully-loaded task system, for the ones with $U < 1$, the schedules obtained under SEED are only ASAP-optimal (see Theorem 2). From Algorithm 1, we can see that the processor will not be idle under SEED if there is any active (ASAP or ALAP) task. That is, SEED adopts the *work-conserving* approach, which conflicts with the ALAP scheduling principle when a task system is not fully-loaded.

For the example task set discussed earlier in Section III, following the steps in Algorithms 1 and 2, its SEED schedule can be found as shown in Figure 1c. Here, we can see that, all instances of the ASAP task T_1 are also executed right after their arrival times (i.e., the SEED schedule is an ASAP-optimal schedule). However, the work-conserving property of SEED (which is critical to ensure its ASAP optimality) also forces it to execute the ALAP tasks T_2 and T_3 earlier. Such early executions of T_2 and T_3 make the resulting SEED schedule inferior to the PO^S-optimal schedule shown in Figure 1a.

V. Preference-Oriented Scheduling Algorithm

Therefore, to comply with the ALAP scheduling principle, we need to judiciously let the processor idle for under-utilized systems even if there are active ALAP tasks. By extending the central ideas of SEED and explicitly taking the ALAP scheduling principle into consideration, in this section, we propose a generalized *Preference-Oriented Earliest Deadline (POED)* scheduling algorithm, which can obtain a PO-optimal schedule for any schedulable task system as shown later.

Here, to manage the idle times and appropriately delay the execution of ALAP tasks without causing any deadline miss, we augment a under-utilized task set Ψ (i.e., $U < 1$) with a *dummy* task T_0 that has period p_0 and utilization as $u_0 = (1-U)$. That is, after the augmentation, we have the task set as $\Psi = (\Psi \cup \{T_0\})$ with $U = 1$. Moreover, in order to postpone the execution of ALAP tasks with the help of dummy task T_0, we assume that T_0 has ASAP preference.

Note that, different from other *real* ASAP tasks, the objective of the dummy task is to (periodically) introduce idle times into the schedule and thus to delay the execution of ALAP tasks. Therefore, when there are active real ASAP tasks, the idle times introduced by the dummy task should not be inserted to comply with the ASAP scheduling principle even if the current dummy task instance has an earlier deadline.

From another perspective, we can consider the idle times as *system slack*, which can be borrowed (reclaimed) by the real ASAP tasks for early executions. To systematically manage system slack (i.e., idle times) and enable appropriate scheduling of such idle intervals at runtime, we adopt the *wrapper-task* mechanism studied in our previous work [15].

Essentially, a wrapper-task WT represents a piece of slack with two parameters (c, d), where size c denotes the amount of slack and deadline d equals to that of the task giving rise to this slack. For the dummy task T_0, there is no real workload and its execution time will be converted to slack whenever it arrives. At any time t, wrapper-tasks are kept in a separate wrapper-task queue $\mathcal{Q}_{WT}(t)$ with increasing order of their deadlines. At runtime, wrapper-tasks compete for the processor with other active tasks based on their priorities (i.e., deadlines).

When a wrapper-task has the earliest deadline, it actually wraps the execution of the highest priority ASAP task (if any) by lending its allocated processor time to the ASAP task and pushing forward the slack; if there is no active ASAP task, the slack is consumed and an idle interval appears. The detailed discussions of wrapper-tasks can be found in [15], and we list below two basic operations that are used in this work:

- *AddSlack*(c, d): create a wrapper-task WT with parameters (c, d) and add it to $\mathcal{Q}_{WT}(t)$. Here, all wrapper-tasks represent slack with different deadlines. Therefore, WT may need to merge with an existing wrapper-task in $\mathcal{Q}_{WT}(t)$ if they have the same deadline;
- *RemoveSlack*(c): remove wrapper-tasks from the front of $\mathcal{Q}_{WT}(t)$ with accumulated size of c. The last one may be partially removed by adjusting its remaining size.

Algorithm 3 The POED Scheduling Algorithm

```
1: //The invocation time of the algorithm is denoted as t.
2: Input: Q_S(t), Q_L(t) and Q_WT(t);
3: if (CPU idle or wrapped-execution occurs in [t^l, t]) then
4:    RemoveSlack(t − t^l);//t^l is previous scheduling time
5:    if (The execution of an ASAP task T_k is wrapped) then
6:       AddSlack(t − t^l, d_k);//push forward the slack
7:    end if
8: end if
9: if (new dummy task arrives at time t) then
10:   AddSlack(c_0, t + p_0);//add new slack
11: end if
12: //suppose that T_k, T_j and WT_x are the header tasks of
13: //Q_S(t), Q_L(t) and Q_WT(t), respectively
14: if (Q_S(t)! = ∅) then
15:   Determine/mark look-ahead interval: [t, min(d_x, d_k)];
16:   if (the first interval [t, t'] is marked "free") then
17:      Execute T_k in [t, t'];//wrapped execution if d_x < d_k
18:   else
19:      Execute T_j in [t, t'];//urgent execution of ALAP tasks
20:   end if
21: else if (Q_WT(t)! = ∅) then
22:   Determine/mark look-ahead interval: [t, d_x];
23:   if (the first interval [t, t'] is marked "free") then
24:      Processor idles in [t, t'];//idle interval appears
25:   else
26:      Execute T_j in [t, t'];//urgent execution of ALAP tasks
27:   end if
28: else
29:   Execute T_j normally;//only ALAP tasks are active
30: end if
```

A. The POED Scheduling Algorithm

With the newly added dummy task, the major steps of POED are summarized in Algorithm 3. Basically, when making scheduling decisions, POED aims at following both ASAP and ALAP scheduling principles by considering first active ASAP tasks, then the wrapper-tasks (representing the idle times) and finally, the active ALAP tasks.

Whenever there are active ASAP tasks, POED tries to execute them in the first place (lines 14 to 20) by following the same steps as in SEED. Recall that wrapper-tasks also compete for processor and may wrap the execution of an ASAP task when a wrapper-task has the highest priority (line 17). Otherwise, if possible, POED will let the processor idle by executing wrapper-tasks and consuming slack (lines 21 to 27). Recall that, the dummy task has ASAP preference, which will be inherited by the wrapper-tasks. Therefore, similar to the handling other real ASAP tasks, a look-ahead interval needs to be checked with the corresponding look-ahead task instance set. Finally, when there are only active ALAP tasks, they are executed in the order of their priorities (line 29).

B. Analysis of the POED Scheduler

Note that, POED can also schedule task systems with $U = 1$. Here, with $u_0 = 0$ for the dummy task, there is no idle time (i.e., slack) and POED will reduce to SEED. Moreover, the adopted wrapper-task mechanism for managing the idle times (slack) under POED does not introduce additional workload into the system [15]. Therefore, following the similar reasonings as those for SEED, we can get that there is no deadline miss under POED for any schedulable (and augmented) task set (with $U \leq 1$).

In addition, to execute the ASAP tasks or wrapper-tasks (idle times) at earlier times and delay the execution of ALAP tasks, POED adopts the same steps as in SEED. Therefore, it has the same complexity as that of SEED, which is $\mathbf{O}\left(n \cdot \frac{p'_{max}}{p_{min}}\right)$ where $p'_{max} = \max\{p_{max}, p_0\}$.

That is, the dummy task's period can have a significant impact on the scheduling overhead of POED. Clearly, selecting smaller periods for the dummy task (when $p_0 > p_{max}$) can reduce POED's scheduling overhead. However, with smaller dummy task's periods, the look-ahead interal will be limited and there may not be enough available idle times to maximally delay the execution of ALAP tasks, where the result schedule can be neither ASAP-optimal nor ALAP-optimal.

In contrast, if the dummy task's period is set as $p_0 = LCM$, we can have the longest look-ahead interval, and the scheduling overhead can be significant. However, in this case, we can always find the longest idle time for the processor whenever there is no active ASAP task (otherwise, a contradiction can be easily found based on Equation 5).

Hence, when $p_0 = LCM$, the execution of all ALAP tasks can be maximally postponed at any time under POED, which results in an ALAP-optimal schedule for any schedulable task set (with $U \leq 1$). Moreover, from Algorithm 3, we know that POED also follows ASAP scheduling principle and will not let the processor idle whenever there are active ASAP tasks. Therefore, the POED schedule is essentially PO^L-optimal when $p_0 = LCM$. In fact, the example in Figure 1b is such a POED schedule.

VI. Evaluations and Discussions

To evaluate the scheduling overhead and how well tasks' preference requirements are achieved, we have implemented the proposed SEED and POED scheduling algorithms and developed a discrete event simulator using C++. For comparison, the well-known EDF scheduler is also implemented.

We consider synthetic task sets with up to 100 tasks, where the utilization of each task is generated using the *UUniFast* scheme proposed in [4]. The period of each task is uniformly distributed in the range of $[p_{min}, p_{max}]$. Each data point in the figures corresponds to the average result of 100 task sets.

A. Scheduling Overhead

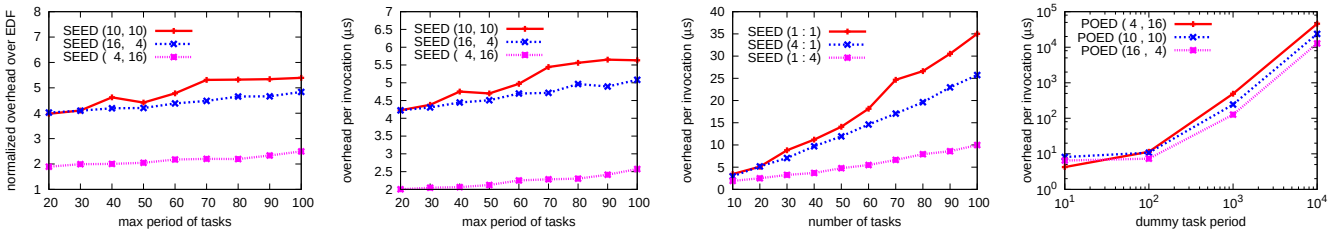
Recall that the complexity of SEED is $\mathbf{O}\left(n \cdot \frac{p_{max}}{p_{min}}\right)$, which depends on both the number of tasks in a task set and tasks' periods. Here, we fix $p_{min} = 10$. With $U = 1$ and $n = 20$, Figure 3a first shows the normalized scheduling overhead of SEED when varying p_{max}. The overhead of EDF is used as the baseline, which depends only on the number of tasks. The two numbers in the labels represent the numbers of ASAP and ALAP tasks, respectively. All experiments were conducted on a Linux box with an Intel Xeon E5507 ($2.0GHz$) processor.

Not surprisingly, when p_{max} becomes larger, the normalized overhead of SEED increases due to larger look-ahead intervals and more task instances in such intervals. With 20 tasks per task set, the overhead of SEED can be up to 6 times of that of EDF when $p_{max} = 100$. The actual scheduling overhead of SEED at each invocation with varying p_{max} are further shown in Figure 3b, which is less than 6 microseconds.

Interestingly, different mixes of ASAP and ALAP tasks can affect SEED's scheduling overhead as well. When the numbers of ASAP and ALAP tasks are equal, the scheduling overhead is much higher than other unbalanced cases. The reason is that, the probability of having both active ASAP and ALAP task instances at each scheduling point is higher for such cases, which require examining the look-ahead intervals.

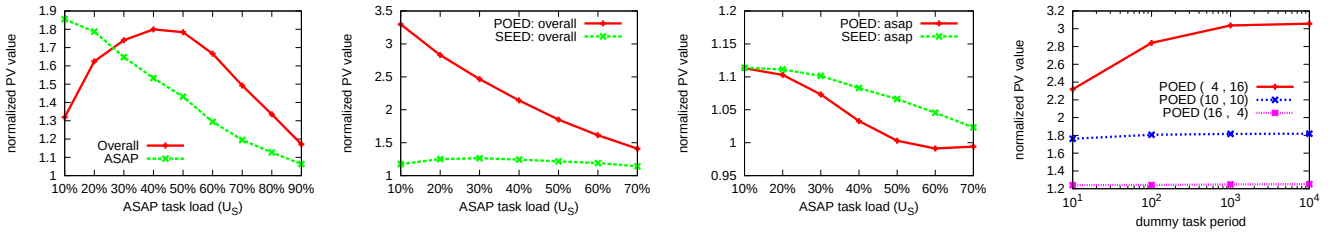
When $p_{max} = 100$, Figure 3c shows scheduling overhead of SEED with varying number of tasks, where the two numbers in the labels represent ratio of ASAP over ALAP tasks. As expected, the overhead increases when there are more tasks. However, the overhead is manageable with less than 35 microseconds per invocation for up to 100 tasks.

Figure 3d further shows the overhead of POED with varying p_0 for systems with $U = 0.8$ and $n = 20$. When p_0 increases and becomes much larger than p_{max}, POED's overhead can become prohibitive. Moreover, when there are more ALAP tasks, it is more likely to have the look-ahead interval to be p_0, where the overhead is generally higher than other cases.



a. normalized overhead of SEED b. SEED overhead vs. p_{max} c. SEED overhead vs. task number d. POED overhead vs. p_0

Fig. 3. Scheduling overheads of SEED and POED with comparison to that of EDF; $U = 1.0$ and 0.8 for SEED and POED, respectively.



a. SEED; $U = 1.0$. b. Overall PV; $U = 0.8$ c. ASAP PV; $U = 0.8$ d. Overall PVs vs. p_0; $U = 0.8$

Fig. 4. Normalized preference values achieved for tasks under SEED and POED ($p_{min} = 10$, $p_{max} = 100$ and $n = 20$)

B. Fulfillment of Preference Requirements

In Section III, the optimality of a schedule for tasks with ASAP and ALAP preferences has been defined based on the accumulated executions of those tasks over sliding time windows, which is difficult to evaluate. To effectively evaluate the performance of different schedulers, we define a new performance metric, denoted as *preference value* (PV) for a periodic task schedule. For a task instance $T_{i,j}$ that arrives at time r with a deadline d, the earliest and latest times to start execution are $st_{min} = r$ and $st_{max} = d - c_i$, respectively, where c_i is T_i's WCET. Similarly, its earliest and latest finish times are $ft_{min} = r + c_i$ and $ft_{max} = d$, respectively.

Suppose that $T_{i,j}$ starts and completes its execution at time st and ft, respectively. According to the preference of task T_i, the preference value for $T_{i,j}$ is defined as:

$$PV_{i,j} = \begin{cases} \frac{ft_{max} - ft}{ft_{max} - ft_{min}} & \text{if } T_i \in \Psi_S; \\ \frac{st - st_{min}}{st_{max} - st_{min}} & \text{if } T_i \in \Psi_L. \end{cases} \quad (6)$$

which has the value within the range of $[0, 1]$. Here, a larger value of $PV_{i,j}$ indicates that $T_{i,j}$'s preference has been served better. Moreover, for a given schedule of a task set, the preference value of a task is defined as the average preference value of all its task instances. In what follows, we report the normalized preference values achieved for tasks under SEED and POED with that of EDF as the baseline.

For fully-loaded systems (i.e., $U = 1$), Figure 4a shows the achieved preference values for all and ASAP tasks with varying ASAP task loads (U_S), which are labeled as *"Overall"* and *"ASAP"*, respectively. There are 20 tasks per task set (i.e., $n = 20$) and the number of ASAP tasks is proportional to ASAP loads. For the overall PVs of all tasks, SEED performs best when there are roughly equal numbers of ASAP and ALAP tasks (i.e., $U_S = 40\%$). This is because it is more likely to have both active ASAP and ALAP tasks at run time where SEED can better address their preferences through the look-ahead intervals.

Note that, if there are only ASAP or ALAP tasks in a task set, SEED essentially reduces to EDF. Therefore, when there are only a few ($U_S = 10\%$) or more ($U_S = 90\%$) ASAP tasks, SEED performs more closely to EDF as the results show. Moreover, if only ASAP tasks are of interest, their achieved PVs with SEED decrease with increasing number of tasks.

For under-utilized systems with $U = 0.8$, Figure 4b shows the achieved PVs for all tasks under POED (where $p_0 = 10$) and SEED. By considering both ASAP and ALAP scheduling principles, POED achieves much better PVs than SEED that focuses on only the ASAP scheduling principle. When task sets contain mostly ALAP tasks with only a few ASAP tasks (i.e., $U_S = 0.1$), POED can achieve close to (more than) 3 times PVs when compared to that of SEED (EDF) since both SEED and EDF are work-conserving schedulers that have conflicts with the ALAP scheduling principle. When there are more ASAP tasks (i.e., larger U_S), the performance of POED gets closer to that of SEED.

Figure 4c further shows the achieved PVs for only ASAP tasks under both SEED and POED when $U = 0.8$. Clearly, SEED performs better here as it puts ASAP tasks in the center stage when making scheduling decisions. More interestingly, we can see that POED can perform even worse than that of EDF. The reason could be that, by forcing the processor to be idle at earlier times, the delayed execution of ALAP tasks under POED can prevent ASAP tasks from executing early, especially when most tasks have ASAP preference.

For the case of $U = 0.8$, Figure 4d shows the achieved PVs for all tasks with varying period (p_0) of the dummy task. Again, the two numbers in the labels represent the number of ASAP and ALAP tasks, respectively. Here, we can see that,

having larger p_0 has very limited improvement on the overall achieved PVs under POED, except for the cases with more ALAP tasks where it is more likely to have the dummy task's period as the look-ahead interval.

VII. Conclusions

In this work, we studied novel scheduling algorithms for a set of periodic tasks with ASAP and ALAP tasks running on a single processor system. We introduced the concept of *preference-oriented (PO) execution* and identified different types of *PO-optimal* schedules. For fully-loaded systems, we showed the *harmonicity* of different PO-optimal schedules while there can be *discrepancies* between them for under-utilized systems.

Then, focusing on fully-loaded systems, we proposed and analyzed an optimal preference-oriented scheduling algorithm (SEED) that explicitly takes the preference of tasks into consideration when making scheduling decisions. Moreover, by taking the idle times in the schedules of under-utilized systems into consideration, we proposed a generalized *preference-oriented earliest deadline (POED)* scheduling algorithm that can generate a PO-optimal schedule for any schedulable task set. The evaluation results show that, with manageable scheduling overheads (less than 35 microseconds per invocation for up to 100 tasks), SEED and POED can achieve significantly better (up to three-fold) preference values when compared to that of EDF.

References

[1] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Trans. on Computers*, 2011.

[2] S. Baruah, R. Howell, and L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor. *Real-Time Systems*, 22(4):301–324, 1990.

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

[4] E. Bini and G.C. Buttazzo. Biasing effects in schedulability measures. In *Proc. of the Euromicro Conf. on Real-Time Systems*, 2004.

[5] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Trans. Softw. Eng.*, 15:1261–1269, 1989.

[6] R. Davis and A. Wellings. Dual priority scheduling. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 100 –109, 1995.

[7] Y. Guo, H. Su, D. Zhu, and H. Aydin. Preference-oriented execution and scheduling algorithms for real-time tasks (extended version). Technical report, Dept. of Computer Science, Univ. of Texas at San Antonio, 2013. available at http://www.cs.utsa.edu/~dzhu/papers/poed-tr.pdf.

[8] Y. Guo, D. Zhu, and H. Aydin. Efficient power management schemes for dual-processor fault-tolerant systems. In *Proc. of the First Workshop on Highly-Reliable Power-Efficient Embedded Designs (HARSH), in conjunction with HPCA*, Feb. 2013.

[9] M. Haque, H. Aydin, and D. Zhu. Energy-aware standby-sparing technique for periodic real-time applications. In *Proc. of the IEEE International Conference on Computer Design (ICCD)*, 2011.

[10] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973.

[11] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 25 – 36, 2003.

[12] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM Trans. Embed. Comput. Syst.*, 7(3), 2008.

[13] 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)*, Mar. 2013.

[14] O. S. Unsal, I. Koren, and C. M. Krishna. Towards energy-aware software-based fault tolerance in real-time systems. In *Proc. of the Int'l Symp. on Low Power Electronics and Design*, pages 124–129, 2002.

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

Appendix
An Application of the POED Scheduler

Reliability and fault tolerance techniques have been the traditional research focus for computing systems, where systems may fail due to various reasons. In this section, we consider a set of period real-time tasks running on a dual-processor system and investigate efficient techniques to tolerate a single permanent fault.

A simple and well-studied approach for fault tolerance would be *hot-standby*, where two copies of the same task run *concurrently* and *simultaneously* on two processors. However, such an approach has the 100% execution overhead and is quite costly. There have been several studies [14] on reducing such overheads, including the *Standby-Sparing (SS)* technique in our recent work [9].

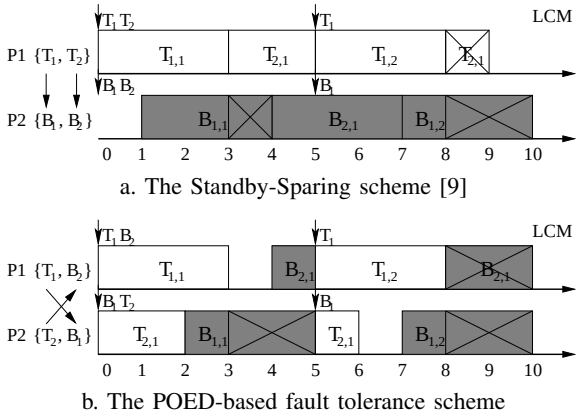


Fig. 5. An example with two tasks: $T_1 = (3, 5)$ and $T_2 = (3, 10)$.

A. An Example

Before formally presenting the POED-based fault tolerance technique, we first look at a concrete example. Consider two periodic tasks $T_1 = (3, 5)$ and $T_2 = (3, 10)$ to run on a dual-processor system. Each (*primary*) task will have a corresponding *backup* task with the same timing parameters. To tolerate a single permanent fault, the primary and backup copies of the same task should be executed on different processors.

In the Standby-Sparing technique [9], all primary tasks run on a (primary) processor, while backup tasks run on another (*spare*) processor, as shown in Figure 5a. To reduce the overlapped executions of primary and backup copies of the same task (and thus the overhead), primary tasks are scheduled according to EDF while backup tasks are delayed as much as possible according to EDL. Note that, when there is no failure during a task's execution, its delayed backup copy will be cancelled (e.g., to save energy [9], [14]).

Suppose that there is no failure during the execution of the tasks in the above example. Then the corresponding schedule within the LCM is shown in Figure 5a, where the cancelled (partial) backup (or primary) copies are marked with "X". Here, we can see that, there are five (5) units of overlapped or wasted executions. With nine (9) units of total workload of the two tasks within a LCM, the execution overhead under the Standby-Sparing technique is $\frac{5}{9} = 56\%$.

Instead of dedicating a processor as the spare, we can schedule the primary and backup copies of tasks in a mixed manner. For the example, we can schedule the primary copy of task T_1 and backup copy of task B_2 on the first processor and so on, as shown in Figure 5b. Here, the mixed task set on each processor are scheduled with the POED scheduler, where primary and backup copies of tasks have ASAP and ALAP preferences, respectively. In this case, when there is no failure during tasks' execution, there are only 3 units of overlapped or wasted execution. This leads to the execution overhead as $\frac{3}{9} = 33\%$, a 23% reduction compared to that of the Standby-Sparing technique.

B. POED-based Fault-Tolerance and Evaluations

We have studied the POED-based fault-tolerance scheme (for energy efficiency) in our recent workshop paper [8]. The basic steps can be summarized as follows:

- **Step 1:** Map primary copies of tasks to the two processors (e.g., according to the worst-fit decreasing heuristic) and mark them as *ASAP* tasks;
- **Step 2:** Allocate the backup copy of each task to the other processor tasks' and mark all the backup tasks as *ALAP* tasks.

Once the primary and backup copies of tasks are allocated to processors, POED can be applied on each processor to reduce their overlapped executions.

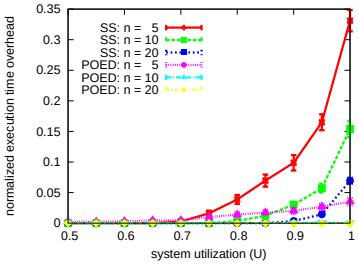


Fig. 6. Execution overheads for POED-based and Standby-Sparing schemes.

Figure 6 shows the execution overheads for the POED-based and Standby-Sparing schemes under different system loads with different numbers of tasks per task set. Here, we can see that, when the system load is low (i.e., $U \leq 0.7$), almost all backup copies can be cancelled under both schemes and, the execution overhead is close to 0. However, when the system load is high (e.g., $U \geq 0.95$), the overhead of the POED-based scheme can be substantially lower than that of Standby-Sparing, especially for cases with only a few tasks. The reason is that, the locations of the backup copies of tasks are fixed according to EDL with the Standby-Sparing scheme. However, in the POED-based scheme, cancelled backup copies generate slack, which is exploited at run time to further delay the execution of future backup tasks. More detailed results and discussions can be found in [8].