Technical Report: CS-TR-2012-009, UTSA

Preference-Oriented Scheduling Framework and its Application to Fault-Tolerant Real-Time Systems (Extended Version)

Prepared on May 15, 2012; Updated on Oct. 15, 2012

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

Hakan Aydin
George Mason University
aydin@cs.gmu.edu

***Abstract*—In fault-tolerant systems, the primary and backup copies of different tasks can be scheduled together on one processor, where primary tasks should be executed *as soon as possible (ASAP)* and backup tasks *as late as possible (ALAP)* for better performance (e.g., energy efficiency). To address such mixed requirements, in this paper, we propose the concept of *preference-oriented execution* and study the corresponding scheduling algorithms. Specifically, we formally define the optimality of preference-oriented schedules and show that such schedules may not always exist for general periodic task sets. Then, we propose an *ASAP-Ensured Earliest Deadline (SEED)* scheduling algorithm, which guarantees to generate an *ASAP-optimal* schedule for any schedulable task set. Moreover, to incorporate the preference for ALAP tasks, we extend SEED and develop a *Preference-Oriented Earliest Deadline (POED)* scheduling heuristic. For a dual-processor fault-tolerant system, we illustrate how such algorithms can be exploited to improve the energy savings. We evaluate the proposed schedulers through extensive simulations. The results confirm the optimality of SEED for ASAP tasks. When compared to the well-known EDF scheduler, both SEED and POED can perform better in preference-oriented settings with reasonable overheads. Moreover, for a dual-processor fault-tolerant system, significant energy savings (up to 20%) can be obtained under POED when compared to the state-of-the-art standby-sparing scheme.**

***Index Terms*—Periodic Real-Time Tasks; Preference-Oriented Execution; Scheduling Algorithms; Optimality;**

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 [8]. 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 when it can be beneficial to execute tasks as late as possible. For instance, to provide better response time for soft aperiodic tasks, the *earliest deadline latest (EDL)* algorithm has been developed to postpone the execution of periodic tasks at their *latest* times provided that all the deadlines will be still met [4]. By delaying the executions 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 [4]. By its very nature, EDL is a non-work-conserving scheduling algorithm: with EDL, the processor may remain idle even though there are ready tasks. On the other hand, Davis and Wellings developed a fixed-priority based scheme, namely *dual-priority (DP)* scheduling [5]. Here, 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. In this way, the responsiveness of soft aperiodic jobs, which are executed at the medium-priority level by default, is improved.

As another example, consider fault-tolerant real-time system design. A common technique, based on the space redundancy approach, is to deploy multiprocessor systems and then schedule two copies of each real-time task (one *primary* copy and one *backup* copy) on two separate processors [2], [6], [9]. Since this approach can potentially consume significant system resources (e.g. CPU time and energy), it is beneficial to cancel the backup task as soon as the corresponding primary task completes successfully. As a result, the backup tasks should be executed *as late as possible (ALAP)* to give greater chance to complete the primary tasks prior to their dispatch time. In fact, EDL has been exploited in a recent study to schedule periodic backup tasks on the secondary processor to reduce the overlapped execution with their primary tasks for better energy savings [7]. However, for systems where primary and backup copies of different tasks are executed in a mixed manner on one processor (where the corresponding backup and primary tasks are on other processors), finding the effective schedule of such tasks remains an open problem [12].

Therefore, we believe there is a strong incentive to develop effective uniprocessor scheduling algorithms for periodic (primary and backup) tasks with different *preferences* on a single processor. To the best of our knowledge, such algorithms have not been well studied in the literature yet. Note that, the well-known scheduling algorithms generally treat all periodic tasks on one processor *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) without considering other system preferences on how to execute the tasks. Intuitively, 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 concern only how to schedule different task groups [10], [11]. However, the existing hierarchical scheduling frameworks consider mostly work-conserving algorithms (e.g. EDF and RM) at both parent and child scheduling components, and it is not obvious how they can be generalized to incorporate non-work-conserving algorithms such as EDL.

Focusing on single-level scheduling, for a set of periodic tasks running on a uniprocessor system where some tasks are preferably executed ASAP and others are preferably executed ALAP, we propose the concept of *preference-oriented execution* and study the corresponding scheduling algorithms. We first formally define the *optimality* of preference-oriented schedules for periodic tasks with their execution preferences being ASAP or ALAP. We show that, when the system utilization is strictly less than 100%, it may not be possible to optimally satisfy the ASAP and ALAP preference objectives at the same time. Therefore, the optimal preference-oriented schedule does not exist for every feasible task set.

Focusing on ASAP tasks, we propose an *ASAP-Ensured Earliest Deadline (SEED)* scheduling algorithm that explicitly takes execution preferences of tasks into consideration when making scheduling decisions. We prove that SEED is an *ASAP-optimal* scheduler that can always generate an *ASAP-optimal* schedule for any schedulable task set (i.e., when the system utilization does not exceed 100%). Moreover, to incorporate the preference of ALAP tasks, we extend SEED and study a *Preference-Oriented Earliest Deadline (POED)* scheduling heuristic. The application of SEED and POED is illustrated for a dual-processor fault-tolerant system to improve its energy efficiency. The simulation results show the superior performance of the proposed SEED and POED schedulers.

The remainder of this paper is organized as follows. Section II presents system models and some notations. In Section III, we formally define and analyze the optimality of different preference-oriented schedules. The SEED scheduling algorithm is proposed and analyzed in Section IV. The POED scheduler, as an extension of SEED, is addressed in Section V. Section VI illustrates the application of SEED/POED in fault-tolerant systems and Section VII presents the evaluation results. The paper concludes in Section VIII.

II. Preliminaries

We consider a set of n periodic real-time tasks $\Psi = \{T_1, \dots, T_n\}$ 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.

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 [8].

In addition to its timing parameters, depending on how the system preferably executes its task instances, each task T_i in Ψ is assigned a *preference*, which can be either *as soon as possible (ASAP)* or *as late as possible (ALAP)* (and T_i is denoted as an ASAP or ALAP task, respectively). Hence, we can partition tasks into two subsets Ψ_S and Ψ_L (where $\Psi = \Psi_S \cup \Psi_L$), which contain the ASAP and ALAP tasks, respectively. Note that, it is not our objective to quantitatively measure how the system ASAP/ALAP preferences of *individual* tasks are fulfilled in a given schedule. Rather, the preference-optimality of a feasible schedule depends on the accumulated performance of all tasks in Ψ_S and/or Ψ_L, as defined in the next section.

Clearly, with a focus on dynamic priority scheduling algorithms, we can simply adopt the EDF algorithm to optimally schedule all tasks if Ψ_L is empty (i.e., no task has ALAP preference) [8]. Similarly, when $\Psi_S = \emptyset$ (i.e., no task has ASAP preference), all tasks in Ψ can be optimally scheduled with the EDL scheduler [4].

In this paper, we consider the cases where the tasks in Ψ have different system 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 $U \leq 1$ [4], [8]. However, as mentioned earlier, without taking the system preferences of executing tasks into consideration, neither of them can effectively address the preference requirements of the system for tasks in both Ψ_S and Ψ_L.

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 the relationship between these optimal schedules. 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 the 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, $\omega(\mathcal{S}, z) = 0$.

Based on the above notations, we present the following definitions regarding to the optimality of preference-oriented schedules.

*Definition 1 (***ASAP-optimal schedule***):* For a given set of periodic tasks with ASAP and ALAP preferences, a feasible schedule $\mathcal{S}^{opt}_{asap}$ is an ASAP-optimal schedule 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-optimal schedule***):* For a given set of periodic tasks with ASAP and ALAP preferences, a feasible schedule $\mathcal{S}^{opt}_{alap}$ is an ALAP-optimal schedule if, for any other feasible schedule $\mathcal{S}$, $\Omega(\mathcal{S}^{opt}_{alap}, t) \geq \Omega(\mathcal{S}, t)$ at any time t ($0 \leq t \leq LCM$).

*Definition 3 (***Optimal preference-oriented schedule***):* For a given set of periodic tasks with ASAP and ALAP preferences, a feasible schedule $\mathcal{S}^{opt}$ is an optimal preference-oriented schedule if, for any other feasible schedule $\mathcal{S}$, $\Delta(\mathcal{S}^{opt}, t) \geq \Delta(\mathcal{S}, t)$ <u>and</u> $\Omega(\mathcal{S}^{opt}, t) \geq \Omega(\mathcal{S}, t)$ at any time t ($0 \leq t \leq LCM$).

Observe that the optimal preference-oriented schedules are defined based on the accumulated executions of ASAP and/or ALAP tasks without distinguishing the execution orders of tasks within the same preference group. 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 effective 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 [8]. 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), then the execution of tasks with ALAP preference in Ψ_L has been also 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 an optimal preference-oriented schedule for the task set under consideration.

Proof: Suppose that for the task set under consideration, $U = 1$. Since the system is fully loaded, we know that 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, the total execution time for tasks with ALAP preference in Ψ_L within *LCM* in any feasible schedule 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:

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

That is, $\mathcal{S}^{opt}$ is an ALAP-optimal schedule as well (from Definition 2), which further indicates that $\mathcal{S}^{opt}$ is an optimal preference-oriented schedule for the task set (from Definition 3). This concludes our proof.

■

B. Discrepant Optimal Schedules: Non-Fully Loaded 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 the ASAP tasks in Ψ_S such idle intervals should appear as late as possible, while for the ALAP tasks in Ψ_L they should be appear as early as possible in a feasible schedule.

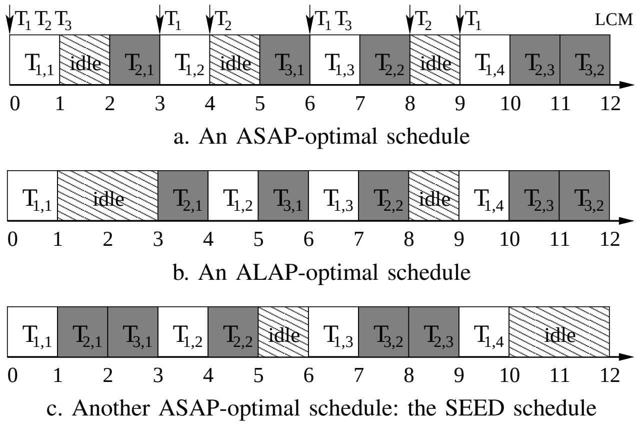


Fig. 1. An example task system with discrepant optimal ASAP and ALAP schedules; Here, the task set has three tasks: $T_1 = (1, 3)$, $T_2 = (1, 4)$ and $T_3 = (1, 6)$; and $\Psi_S = \{T_1\}$ and $\Psi_L = \{T_2, T_3\}$.

To illustrate the discrepancy between the ASAP-optimal and ALAP-optimal schedule requirements 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 is an ASAP task while T_2 and T_3 are both ALAP tasks. 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 $(1 - U) \cdot LCM = (1 - 0.75) \cdot 12 = 3$.

By executing all task instances of T_1 right after their arrival times, an ASAP-optimal schedule $\mathcal{S}^{opt}_{asap}$ can be found as shown in Figure 1a. Note that, tasks T_2 and T_3's ALAP preference has also been considered in $\mathcal{S}^{opt}_{asap}$, where most of their task instances are delayed and executed right before their deadlines. However, $\mathcal{S}^{opt}_{asap}$ is <u>not</u> an ALAP-optimal schedule according to Definition 2, as it is possible to further delay the execution of task T_2's first task instance $T_{2,1}$ without causing any deadline miss.

By further delaying $T_{2,1}$'s execution to the moment right before its deadline, an ALAP-optimal schedule $\mathcal{S}^{opt}_{alap}$ can be found as shown in Figure 1b, where one additional idle time appears before $T_{2,1}$'s execution. Note that, *all* task instances of T_2 and T_3 have been maximally delayed where most are executed right before their deadlines in $\mathcal{S}^{opt}_{alap}$. Unfortunately, such delayed execution of $T_{2,1}$ causes some conflicts when T_1's second task instance $T_{1,2}$ arrives. Here, $T_{1,2}$ cannot be executed right after its arrival any more, as such execution would cause $T_{2,1}$ to miss its deadline.

Hence, we can see that, due to the conflicting requirements for idle intervals in a feasible schedule, there are generally some discrepancies between the ASAP-optimal and ALAP-optimal schedules of non-fully-loaded systems. That is, for such task systems, it may not exist an optimal preference-oriented schedule that is both ASAP and ALAP optimal, which is formally stated in the following proposition:

Proposition 1: For a set of ALAP and ASAP periodic tasks, if the system's utilization is strictly less than 1.0, there may **not** exist an preference-oriented schedule $\mathcal{S}^{opt}$ which is optimal simultaneously for both subsets of tasks with ASAP and ALAP preferences.

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.

Unfortunately, these two principles may be conflicting with each other and it may not always possible for a scheduler to simultaneously both at all times. In fact, the last example shows that a schedulable task set may not even have an optimal preference-oriented schedule when the system is not fully loaded. Therefore, in what follows, we first study an ASAP-optimal scheduling algorithm by primarily adhering to the ASAP scheduling principle (Section IV). Then, a preference-oriented scheduling heuristic is proposed, in an attempt to address both ASAP and ALAP scheduling principles simultaneously (Section V).

IV. An ASAP-Optimal Scheduling Algorithm

From previous discussions, we know that the optimality of a feasible schedule for a set of periodic tasks with ASAP and ALAP preferences depends only on the accumulated execution of tasks in Ψ_S and/or Ψ_L rather than the execution of individual tasks. Hence, focusing on dynamic priority based scheduling, in this section, we propose an *A<u>S</u>AP-<u>E</u>nsured <u>E</u>arliest <u>D</u>eadline (SEED)* scheduling algorithm that follows only the first ASAP scheduling principle. We show that SEED is an ASAP-optimal scheduler that can always generate an ASAP-optimal schedule for any schedulable task set with system utilization $U \leq 1$.

A. SEED Scheduling Algorithm

As with the EDF and EDL schedulers [4], [8], the SEED scheduling algorithm considers tasks' timing constraints: 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 *all* tasks *solely* based on their priorities, SEED also takes tasks' preferences into consideration and treats tasks with ASAP and ALAP preferences differently. In particular, to *fully* comply with the ASAP scheduling principle, SEED puts tasks with ASAP preference in the center stage when making scheduling decisions.

The main steps of SEED are summarized in Algorithm 1, which can be invoked on different occasions: a) the current

Algorithm 1 The SEED Scheduling Algorithm

```
1: //The invocation time of the algorithm is denoted as $t$.
2: Input: $\mathcal{Q}_S(t)$ and $\mathcal{Q}_L(t)$, which are ready queues for
   active ASAP and ALAP tasks at time $t$, respectively;
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($\mathcal{Q}_S(t)$); //$T_k$ has the earliest deadline
6:    else if ($\mathcal{Q}_L(t) != \emptyset$) then
7:       $T_k$ =Dequeue($\mathcal{Q}_L(t)$); //$T_k$ has the earliest deadline
8:    end if
9:    Execute $T_k$ or let CPU idle (if $\mathcal{Q}_S(t) = \mathcal{Q}_L(t) = \emptyset$);
10: else
11:    //$\mathcal{Q}_S(t)$'s header task $T_k$ has the earliest deadline $d_k$;
12:    Determine the look-ahead interval as $[t, d_k]$;
13:    Construct the look-ahead queue $\mathcal{Q}_{la}$, which may contain
       both current and future task instances;
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:       Execute highest priority task(s) from $\mathcal{Q}_L(t)$ in $[t, t']$;
17:    else
18:       $T_k$ =Dequeue($\mathcal{Q}_S(t)$; and execute $T_k$ in $[t, t']$;
19:    end if
20: 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:       Consider the free sections in $[t, d_i]$ in reverse order
         of their appearance, and mark them as "reserved",
         where the marked sections have length $c_i$;
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:         Consider the free sections in $[t, a_i]$ in reverse order
            of their appearance, and mark them as "reserved",
            where marked sections have length $(c_i - L)$;
15:      end if
16:   end if
17: end while
```

task completes; b) the current task is preempted; or c) a new task arrives. At any invocation time t, we use two ready task queues $\mathcal{Q}_S(t)$ and $\mathcal{Q}_L(t)$ to track active ASAP and ALAP tasks, respectively. Tasks in both queues are ordered in the decreasing order of their priorities. If only one of the queues is empty, then 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 4 to 9). When both queues are empty, there is no active task and CPU will idle (line 9).

Note that, when all active tasks have ALAP preference (line 6), then the processor should be preferably left idle if it is possible to do so without causing any (current and future) task to miss its deadline (i.e., the ALAP scheduling principle). However, as shown in the previous example (see Figure 1), such delayed execution of ALAP tasks can affect the early execution of future ASAP tasks and thus the ASAP optimality of SEED. Therefore, to guarantee its ASAP optimality and ensure that all ASAP tasks can be executed at their earliest time, SEED adopts the *work conserving* approach and does not fully enforce the ALAP scheduling principle: with SEED, the processor will not be idle if there are any active tasks.

The complicated case is when there are active ASAP and ALAP tasks (i.e., both $\mathcal{Q}_S(t)$ and $\mathcal{Q}_L(t)$ are not empty; line 10). Here, according to the ASAP scheduling principle, if possible, SEED should execute the header task T_k that has the highest priority in $\mathcal{Q}_S(t)$ even if its deadline is later than that of $\mathcal{Q}_L(t)$'s header task. 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 (line 12). 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 tasks that form a *look-ahead task 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:

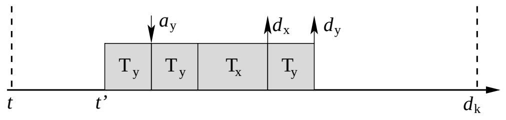
$$\Psi_{la}(t, d_k) = \{T_i | (T_i \in \mathcal{Q}_L(t) \vee a_i > t) \wedge d_i < d_k\} \tag{3}$$

where a_i is the arrival time of a future task instance T_i. That is, $\Psi_{la}(t, d_k)$ includes *active* ALAP tasks in $\mathcal{Q}_L(t)$ and *future* tasks (with $a_i > t$) 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.

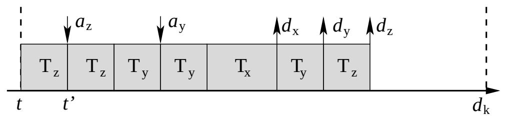
Then, for the look-ahead interval $[t, d_k]$, the *reserved* sections for task instances in $\Psi_{la}(t, d_k)$ are determined with the help of the function *Mark*$(t, d_k, \mathcal{Q}_{la})$ (line 14). There are two possibilities for the result as illustrated next in Figure 2. If the first section $[t, t']$ 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) may have 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 sole 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).

If T_i is an ALAP task, 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 $\mathcal{Q}_{la}$.



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.

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 higher priority, the algorithm first marks the section of size c_x right before its deadline d_x as *"reserved"* as shown in Figure 2a. Next, the execution time c_y of T_y is larger than the free section within $[a_y, d_y]$. In this case, we 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 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., there are $a_i > t$ 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).

We continue 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 uses 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.

For the example discussed earlier in Section III, following the steps in Algorithms 1 and 2, the SEED schedule can be found as shown in Figure 1c. Here, we can see that, all task instances of the ASAP task T_1 are also executed right after their arrival times. Therefore, the SEED schedule is also an ASAP-optimal schedule. However, the work-conserving property of SEED (that is critical to ensure its ASAP optimality) also forces it to execute the ALAP tasks T_2 and T_3 earlier, which makes the SEED schedule inferior to the ASAP-optimal schedule shown in Figure 1a. Unfortunately, as per previous discussions, it is not possible to optimally incorporate the preference requirement of ALAP tasks into the SEED scheduler. Hence, we focus on a preference-oriented heuristic scheduling algorithm in Section V.

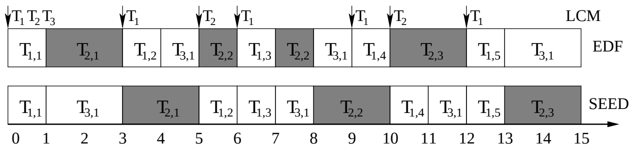
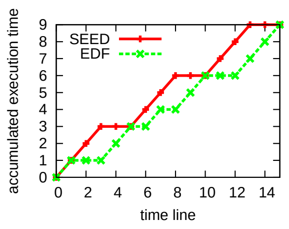
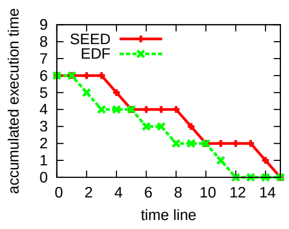


Fig. 3. The SEED and EDF schedules for a set of three tasks: $T_1 = (1, 3)$, $T_2 = (2, 5)$ and $T_3 = (4, 15)$; and $\Psi_S = \{T_1, T_3\}$ and $\Psi_L = \{T_2\}$.

To illustrate the optimality of SEED for fully loaded task systems, we consider another task set: $T_1 = (1, 3)$, $T_2 = (2, 5)$ and $T_3 = (4, 15)$, where $U = \frac{1}{3} + \frac{2}{5} + \frac{4}{15} = 1$. Suppose that $\Psi_S = \{T_1, T_3\}$ and $\Psi_L = \{T_2\}$. Figure 3b shows the SEED schedule of the tasks within $LCM = 15$. For comparison, the task set's EDF schedule is also shown in Figure 3a. From the figures, it can be easily seen that, compared to that of the EDF schedule, the execution of ASAP tasks T_1 and T_3 is moved to the front while the ALAP task T_2's execution has been pushed towards the end of the SEED schedule.



(a) ASAP tasks: $\Delta(\mathcal{S}, t)$



(b) ALAP tasks: $\Omega(\mathcal{S}, t)$

Fig. 4. The accumulated execution times in EDF and PO-ED schedules

Such observations can be further confirmed from the accumulated execution times for ASAP and ALAP tasks, respectively, in the SEED and EDF schedules for the example as shown in Figure 4. Here, we can see that the accumulated execution time $\Delta(\mathcal{S}, t)$ and $\Omega(\mathcal{S}, t)$ for ASAP and ALAP tasks, respectively, in the SEED schedule are always better than or at least equal to those in the EDF schedule at any time t. As proved next, SEED is an ASAP-optimal scheduler and, for a fully loaded task set with $U = 1$, the SEED schedule is essentially an optimal preference-oriented schedule.

B. Optimality of the SEED Scheduler

In this section, we provide formal analysis and proof for the optimality of our 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 SEED is an ASAP-optimal scheduler (i.e., for any schedulable task set, SEED will generate an ASAP-optimal schedule). This further implies that SEED can can generate optimal preference-oriented schedules for fully loaded task systems.

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 [1], 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. Complexity of the SEED Scheduler

From Algorithms 1 and 2, we can see that the complexity of the SEED scheduler comes mainly from determining the reserved sections of the look-ahead interval when there are both active ASAP and ALAP tasks. 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 $n \cdot \lfloor \frac{p_{max}}{p_{min}} \rfloor$. Note that, each task instance in $\mathcal{Q}_{la}$ needs to search for free sections in the look-ahead interval. Therefore, the complexity of SEED at each scheduling point can be found as $\mathbf{O}\left(n \cdot \frac{p_{max}^2}{p_{min}}\right)$.

V. Preference-Oriented Scheduling Heuristic

Although SEED is an ASAP-optimal scheduler, it does not fully enforce the ALAP scheduling principle with the adopted work-conserving approach. As shown later in the evaluations, such an approach can lead to dramatically low preference values for ALAP tasks for non-fully loaded task systems. Considering the fact that it is impossible to optimally address the preference requirement of both ASAP and ALAP tasks simultaneously, by extending the central ideas of the SEED scheduler, we further study a *preference-oriented earliest deadline (POED)* scheduling heuristic.

In POED, to avoid executing ALAP tasks early when there are only active ALAP tasks, *slack time* (i.e., idle intervals) in the system has to be managed *explicitly* for appropriate delayed execution of ALAP tasks without missing any task's deadline. For such a purpose, for any task set Ψ with $U < 1$, a *dummy* periodic task T_0 with $u_0 = (1 - U)$ is added into the system. Here, the augmented task set $\Psi' = (\Psi \cup \{T_0\})$ will have the total utilization $U' = 1$ and and the system becomes fully loaded. However, the sole objective of the dummy task (whose *actual* execution time will be always 0 at run-time) is to introduce slack into the system periodically. It is easy to see that the period of task T_0 controls the amount and the frequency of the slack introduced into the system. For simplicity and efficiency, we assume that $p_0 = p_{min}$ and $c_0 = p_0 \cdot u_0$, where p_{min} is the minimum period of real tasks.

A. Wrapper-Tasks for Slack Management

To efficiently manage slack and enable appropriate scheduling of idle intervals at runtime, we adopt the *wrapper-task* mechanism studied in our previous work [13]. Essentially, a wrapper-task WT represents a piece of slack with two parameters (c, d), where the size c denotes the amount of the slack and the deadline d equals to that of the task giving rise to this slack. At any time t, wrapper-tasks are kept in a separate wrapper-task queue $\mathcal{Q}_{WT}(t)$ with increasing order of their deadlines (i.e., wrapper-tasks with smaller deadlines are in the front of $\mathcal{Q}_{WT}(t)$).

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 [13], 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.

B. The POED Scheduling Algorithm

With the newly added dummy task, the major steps of POED are summarized in Algorithm 3. Essentially, when making scheduling decisions, POED tries to follow both ASAP and ALAP scheduling principles by considering first active ASAP tasks, then the wrapper-tasks (that represent the static slack) and finally, the active ALAP tasks.

Whenever there are active ASAP tasks, POED tries to execute them in the first place (lines 13 to 19) 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 16). Otherwise, POED tries to let the processor idle by executing wrapper-tasks and consuming slack if possible (lines 20 to 26). Finally, when there are only active ALAP tasks, they are executed in the order of their priorities (line 28).

Note that, when executing the ASAP tasks or wrapper-tasks (slack) early while delaying the execution of ALAP tasks, POED adopts the same steps and thus has the same complexity as those of SEED. Moreover, the wrapper-task mechanism does not introduce additional workload into the system [13]. Therefore, following a similar reasoning, POED can be shown to guarantee all the timing constraints.

Algorithm 3 The POED Scheduling Algorithm at time t

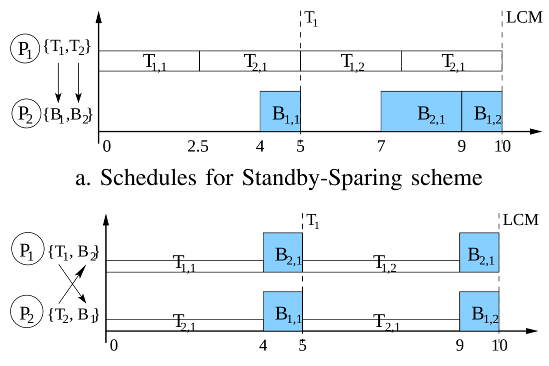
```
1:  **Input:** $\mathcal{Q}_S(t)$, $\mathcal{Q}_L(t)$ and $\mathcal{Q}_{WT}(t)$;
2:  **if** (CPU idle or wrapped-execution occurs in $[t^l, t]$) **then**
3:      *RemoveSlack*($t - t^l$);//$t^l$ is previous scheduling time
4:      **if** (The execution of an ASAP task $T_k$ is wrapped) **then**
5: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

VI. Application to Fault-Tolerant Systems

In this section, we illustrate the application of SEED/POED algorithms in fault-tolerant systems through a concrete example. Here, we consider a dual-processor system with two periodic tasks where $T_1 = (1, 5)$ and $T_2 = (2, 10)$. To tolerate a permanent fault, two copies (i.e., *primary* and *backup*) for each task have to be scheduled on different processors. Once a task's primary copy completes successfully, its backup copy can be cancelled. Therefore, for energy efficiency, we should postpone the execution of a task's backup copy as much as possible and minimize the overlapped execution with its primary copy running on another processor [12].

Specifically, for dual-processor fault-tolerant systems, the *Standby-Sparing* scheme has been recently studied [7]. For the example, as shown in Figure 5a, the primary copies of all tasks are executed at a scaled frequency on one processor under EDF, while all backup copies are scheduled on the secondary processor under EDL for energy savings. Note that, to address the negative effects of voltage scaling on system reliability, the executions of tasks' backup copies are not scaled and the slack time on the secondary processor is wasted [7].

To efficiently utilize the slack time on both processors, we can partition the primary copies of tasks among the processors and allocate the corresponding backup copies on another processor. As shown in Figure 5b, each processor will have a mixed set of primary and backup copies of different tasks. Here, the SEED scheduler can execute the scaled primary and non-scaled backup copies of tasks in ASAP and ALAP fashion, respectively, to improve the energy efficiency. Specifically, with the power model and parameters adopted in [7], the SEED-based scheme can save 20% more energy in this example comparing to that of the Standby-Sparing scheme.



b. Schedules for SEED-based scheme

Fig. 5. A dual-processor fault-tolerant system with two tasks: $T_1 = (1, 5)$ and $T_2 = (2, 10)$.

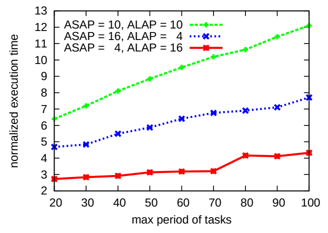
VII. Simulations and Evaluations

In this section, we evaluate the performance of our proposed SEED and POED schedulers through extensive simulations. First, comparing to the well-known EDF scheduler, we first evaluate their scheduling overheads and how well they can achieve the preference requirements of tasks. Moreover, focusing on dual-processor fault-tolerant systems, the energy efficiency of both SEED and POED schedulers is evaluated and compared to the Standby-Sparing scheme.

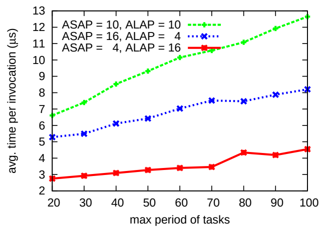
We consider synthetic task sets with 10 or 20 tasks, where the utilization of each task is generated using the *UUniFast* scheme proposed in [3]. The period of each task is uniformly distributed in the range of $[p_{min}, p_{max}]$, where the default values are $p_{min} = 10$ and $p_{max} = 100$. We vary the system load (i.e., utilization U), the number and workload of ASAP and ALAP tasks, as well as the maximum period of tasks p_{max} during the simulations. Each data point in the figures corresponds to the average result of 1000 synthetic task sets. All experiments were conducted on a Linux box with an Intel Xeon E5507 ($2.0GHz$) processor and 32 GB of memory.

A. Scheduling Overhead

First, we evaluate the scheduling overhead of the SEED scheduler. Here we consider task sets with 20 tasks and system utilization as $U = 1$. Recall that the complexity of SEED at every scheduling point is $\mathbf{O}\left(n \cdot \frac{p_{max}^2}{p_{min}}\right)$, where n is the number of tasks in the task set (see Section IV-B). That is, the scheduling overhead of SEED depends heavily on p_{max} (the maximum period of tasks).



a. normalized time overhead



b. Average time per invocation

Fig. 6. Scheduling overhead of SEED over EDF for 20 tasks

By varying p_{max} (with fixed $p_{min} = 10$), Figure 6a first shows the normalized overall execution time of SEED for generating the schedule within LCM, where the execution time of EDF is used as the baseline. The implemented EDF scheduler has a linear complexity of $\mathbf{O}(n)$, which does not depend on p_{max}. As expected, when p_{max} becomes larger, the scheduling overhead of SEED increases due to larger look-ahead intervals and increased number of task instances in the look-ahead intervals. With 20 tasks in a task set, SEED can take up to 12 times longer to generate the schedule within LCM when compared to that of EDF.

Moreover, interestingly, different numbers of ASAP and ALAP tasks in the system also have an important effect on the execution time of the SEED scheduler. When there is an equal number (10) of ASAP and ALAP tasks the scheduling overhead is higher than other unbalanced cases. It is because the probability of having both active ASAP and ALAP task instances at each scheduling point is higher for such cases, which require checking the states of the look-ahead intervals.

Furthermore, Figure 6b shows the average execution time per scheduling point for the SEED scheduler. Here, we can see that, although the scheduling overhead of SEED scheduler is much higher than that of EDF, the overhead at each scheduling point is still manageable with about only 10 microseconds, compared to the execution time of real-time tasks that normally takes a few or tens milliseconds.

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, which are time varying values and hard to evaluate at run-time. To quantitatively evaluate the performance of different schedulers on the fulfillment of tasks' preferences, we define a new performance metric, denoted as *preference value* (PV) for any feasible schedule. For a periodic task instance $T_{i,j}$ that arrives at time r with a deadline d, the earliest and latest times for it to start its execution are $st_{min} = r$ and $st_{max} = d - c_i$, respectively, where c_i is the WCET of T_i. 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:

$$PV_{i,j} = \begin{cases} \frac{ft_{max}-ft}{ft_{max}-ft_{min}} & \text{if } T_i \in \Psi_S; \\ \frac{st-st_{min}}{st_{max}-st_{min}} & \text{if } T_i \in \Psi_L. \end{cases} \quad (4)$$

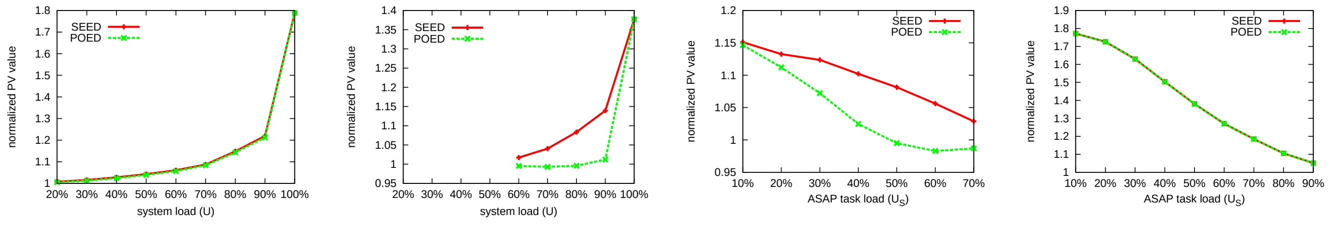
which has the value within the range of $[0, 1]$. Here, a larger value of $PV_{i,j}$ indicates that $T_{i,j}$'s preference has been served better. Moreover, for a given schedule of a task set, the preference value of a task is defined as the average preference value of all its task instances. In what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

Considering that SEED is an ASAP-optimal scheduler, we start with the average preference values for ASAP tasks where the results for cases with 10 tasks per task set are shown in Figure 7. First, by varying the system loads (i.e., U), Figures 7ab show the achieved preference values for ASAP tasks when the loads of ASAP tasks are $U_S = 0.1$ and $U_S = 0.5$, respectively. From the results, we can see that, although SEED always achieves better preference values for ASAP tasks as an ASAP-optimal scheduler, the improvement on the preference values of ASAP tasks over that of EDF is very marginal when the system load is low (e.g., $U \leq 0.4$). The reason is that, under very low system loads, almost all task instances can be executed right after their arrival times even under EDF. However, as system load increases, better preference values for ASAP tasks can be achieved under SEED as more task instances of ASAP tasks can be executed and finish earlier under SEED comparing to the case of EDF.

When the load of ASAP tasks (U_S) varies, Figures 7cd further show the achieved preference values for ASAP tasks for system utilization of $U = 0.8$ and $U = 1.0$, respectively. We can see that the improvement of achieved preference values for ASAP tasks decreases as the load of ASAP tasks increases. This is because in general it is harder for SEED to complete larger number of ASAP tasks earlier.

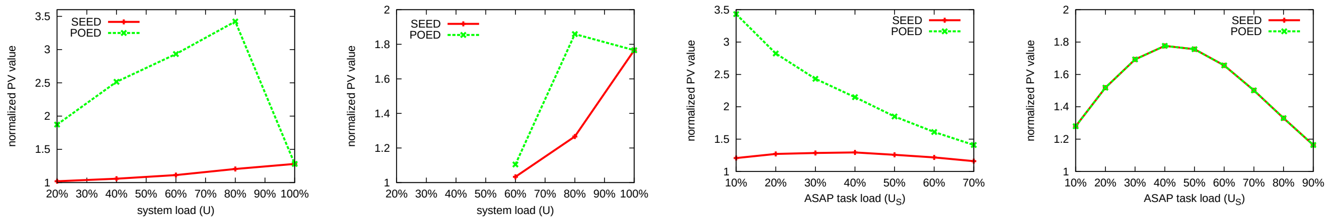
When the system utilization $U < 1$, POED achieves up to 10% less preference values for ASAP tasks compared to SEED. For fully loaded systems (with $U = 1$), there is no slack in the system and POED essentially reduces to SEED (Figure 7d). Interestingly, we see that the achieved preference values for ASAP tasks under POED can be even less than that of EDF (as shown in Figure 7c). The reason could be that, by leaving the processor idle as early as possible, the delayed execution of ALAP tasks under POED can prevent ASAP tasks from completing early, especially when there are many such tasks in the systems.

For the same settings, Figure 8 further shows the achieved preference values for all (including both ASAP and ALAP) tasks. Here, we can see that, by properly managing the idle intervals, POED can achieve much better (up to three-fold) overall preference values for all tasks when compared to SEED. Similarly, when the load of ASAP tasks becomes higher, it becomes more difficult for both SEED and POED to finish such tasks earlier (and thus delay the execution of ALAP tasks), which leads to less improvement for the achieved



a. $U_S = 0.1$ b. $U_S = 0.5$ c. $U = 0.8$ d. $U = 1.0$

Fig. 7. Normalized preference values of ASAP tasks under SEED and POED; 10 tasks per task set.



a. $U_S = 0.1$ b. $U_S = 0.5$ c. $U = 0.8$ d. $U = 1.0$

Fig. 8. Normalized preference values for all tasks under SEED and POED; 10 tasks per task set.

preference values. Moreover, as before, when $U = 1$ SEED and POED converge and the best improvement is obtained when the loads of ASAP and ALAP tasks are balanced.

C. Energy-Efficiency of SEED and POED-based Schemes

In this section, we focus on a dual-processor fault-tolerant system and explore the energy-savings yielded by the preference-oriented framework. After the tasks are generated randomly as before, the primary and backup copies are executed in ASAP and ALAP fashion, respectively, as described in Section VI. To calculate energy consumption, we adopt the same power model and parameters as in [7] and assume that there are four normalized frequency levels (which are $\{1.0, 0.8, 0.6, 0.4\}$) for the processors.

For comparison, we have implemented the following schemes: the *basic Standby-Sparing (Basic-SS)* that does not scale down tasks' primary copies; the Standby-Sparing with static scaled frequency for tasks' primary copies (denoted as *SS-SPM*)[7]; the two techniques that statically scale the frequency for tasks' primary copies under SEED and POED schedulers, denoted as *SEED-SPM* and *POED-SPM*, respectively; finally, *POED-DPM* that exploits online slack to further reduce the processing frequency of primary copies of tasks. All schemes cancel backup copies' executions once the corresponding primary copy completes.

Figure 9 shows the normalized energy consumption under different schemes with varying system loads, where the one under *Basic-SS* is used as the baseline. Here, all tasks take their WCETs at runtime and each data point is the average result of 100 task sets. In general, the normalized energy consumption increases with the system utilization, since the scaled frequency for primary tasks gets larger.

However, the energy consumption under *SEED-SPM* is significantly higher than that of *SS-SPM* and even higher than *Basic-SS* at high system utilization. The reason comes from

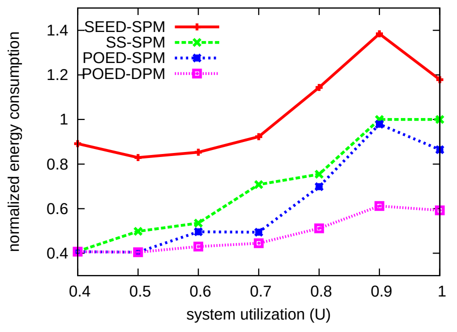


Fig. 9. Normalized energy consumption *vs.* static load; 10 tasks per task set

the fact that *SEED-SPM* focuses only on the early execution of primary copies of tasks and does not attempt to delay the execution of backup copies, which have been postponed at their latest times under both *Basic-SS* and *SS-SPM*. In contrast, when the execution of such backup copies is postponed through slack management under *POED-SPM*, better energy savings can be obtained. Note that, when system utilization $U = 0.9$, the static scaled frequency for primary copies of tasks will be 1.0 due to discrete frequency limitation and the energy savings are limited under *POED-SPM*. Moreover, if online slack is utilized to scale down the processing frequency of primary copies further, significantly more energy (up to 40%) can be saved under *POED-DPM*.

VIII. Conclusions

In this paper, we introduced the concept of *preference-oriented execution* for real-time tasks, where some tasks are ideally to be executed *as soon as possible (ASAP)*, while others *as late as possible (ALAP)*. After defining optimal preference-oriented schedules, we showed that such schedules do not always exist. We proposed an *ASAP-Ensured Earliest*

Deadline (SEED) scheduling algorithm, which guarantees to generate an ASAP-optimal schedule for any schedulable task set (with system utilization being no more than 100%) by explicitly taking the preference of tasks into consideration when making scheduling decisions. Moreover, a *Preference-Oriented Earliest Deadline (POED)* heuristic is studied to incorporate the preference requirements of ALAP tasks. We further illustrate how such preference-oriented schedulers can be applied to fault-tolerant systems for energy efficiency.

The proposed schedulers are evaluated through extensive simulations. Although the scheduling overhead of SEED is much higher than that of EDF, such overhead is still manageable (about 10 microseconds for 20 tasks at each scheduling point). The results confirm the ASAP optimality of SEED scheduler. Moreover, POED can achieve significantly better (up to three-fold) preference values by considering the preference requirements of both ASAP and ALAP tasks. The results also show that, comparing to the existing standby-sparing technique, more energy savings can be obtained, especially under the POED-based schemes.

References

[1] 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.

[2] A. A. Bertossi, L. V. Mancini, and A. Menapace. Scheduling hard-real-time tasks with backup phasing delay. In *Proceedings of the 10th IEEE international symposium on Distributed Simulation and Real-Time Applications*, pages 107–118, Washington, DC, USA, 2006. IEEE Computer Society.

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

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

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

[6] S. Ghosh, R. Melhem, and D. Mossé. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *Parallel and Distributed Systems, IEEE Transactions on*, 8(3):272 –284, March 1997.

[7] 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.

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

[9] G. Manimaran and C. S. R. Murthy. A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. *Parallel and Distributed Systems, IEEE Transactions on*, 9(11):1137 –1152, nov 1998.

[10] 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.

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

[12] 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.

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