Journal of Systems Architecture 61 (2015) 127–139

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Preference-oriented real-time scheduling and its application in fault-tolerant systems

Yifeng Guo [a], Hang Su [a], Dakai Zhu [a,*], Hakan Aydin [b]

[a] Department of Computer Science, The University of Texas at San Antonio, San Antonio, TX 78249, USA
[b] Department of Computer Science, George Mason University, Fairfax, VA 22030, USA

ARTICLE INFO

Article history:
Received 26 August 2014
Received in revised form 14 November 2014
Accepted 21 December 2014
Available online 3 January 2015

Keywords:
Periodic real-time tasks
Preference-oriented execution
Scheduling algorithms
Fault-tolerant systems

ABSTRACT

In this paper, 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 ALAP tasks' 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. The application of the POED scheduler in a dual-processor fault-tolerant system is further illustrated. We 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.

1. 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 [16]. 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 [8]. 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 [8]. By its very nature, EDL is a non-work-conserving scheduling algorithm: the processor may remain idle even though there are ready tasks. With the same objective, *dual-priority (DP)* was developed based on the phase delay technique [1] for fixed-priority rate-monotonic scheduling [9]. 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. Soft aperiodic tasks are executed at the medium-priority level to improve their responsiveness.

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 [4,12,22]. 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 [14].

However, 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 [4,12,22], the

* Corresponding author. Tel.: +1 210 458 7453; fax: +1 210 458 4437.
E-mail address: dakai.zhu@utsa.edu (D. Zhu).

http://dx.doi.org/10.1016/j.sysarc.2014.12.001
1383-7621/

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. 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). Hence, neither of them can effectively handle tasks with *different preferences*.

Intuitively, one may consider adopting the hierarchical scheduling approach [18,19] to solve such problems, where tasks with the same preference form a task group and the high-level scheduler would determine only how to schedule different task groups. However, the existing hierarchical scheduling frameworks consider mostly work-conserving algorithms (such as EDF and RM) at both parent and child scheduling components. It is not obvious how such framework can be generalized to non-work-conserving algorithms (such as EDL and DP) in order to comply with tasks' different execution preferences while guaranteeing their timing constraints.

Therefore, we believe that there is a strong incentive to develop effective uniprocessor scheduling algorithms for periodic tasks with different *execution preferences* (e.g., ASAP and ALAP). In addition to fault-tolerant systems, such algorithms can also be applied in mixed-criticality task systems [2], where high-criticality tasks can be given 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 [20].

However, to the best of our knowledge, such scheduling algorithms have not been well studied in the literature yet. In this work, we consider periodic tasks running on a uniprocessor system where some tasks are preferably executed ASAP while others ALAP. We study effective scheduling algorithms and illustrate their applications. Specifically, the main contributions of this paper are summarized as follows:

- The concept of *preference-oriented (PO)* execution is introduced for tasks with ASAP and ALAP preferences. Two types of *PO-optimal* schedules are defined, where their *harmonicity* and *discrepancies* for fully-loaded and under-utilized systems, respectively, are analyzed.
- An optimal *ASAP-Ensured Earliest Deadline (SEED)* scheduling algorithm, which takes the preference of ASAP tasks into consideration when making scheduling decisions, is proposed for fully-loaded systems.
- A generalized *Preference-Oriented Earliest Deadline (POED)* scheduler is also studied by extending SEED and explicitly managing system idle time, which can obtain a PO-optimal schedule for any schedulable task set.
- The application of the POED scheduler in dual-processor fault-tolerant systems to reduce execution overhead and thus improve system efficiency is further illustrated.
- Finally, we evaluate the proposed schedulers through extensive simulations. The results 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 EDF scheduler. Moreover, the execution overhead in dual-processor fault-tolerant systems can be significantly reduced under POED when compared to the state-of-the-art standby-sparing scheme.

The remainder of this paper is organized as follows. Section 2 reviews closely related work. Section 3 presents system models and some notations. In Section 4, we formally define and investigate the optimality of different preference-oriented schedules. The SEED scheduling algorithm is proposed and analyzed in Section 5. The generalized POED scheduler is addressed in Section 6 and Section 7 illustrates the application of POED in fault-tolerant systems. Section 8 presents the evaluation results and Section 9 concludes the paper.

2. Closely related work

In this section, we review closely related work on scheduling algorithms for periodic real-time tasks running on uniprocessor systems and techniques to reduce execution overhead in fault-tolerant systems. The *earliest-deadline-first (EDF)* and *rate monotonic (RM)* scheduling algorithms, which are well-known optimal schedulers for periodic tasks running on a uniprocessor system, have been studied in [16]. Here, EDF is a dynamic-priority scheduler that prioritizes and schedules tasks based on the deadlines of their current task instances. In comparison, RM is a fixed-priority scheduler that prioritizes tasks according to their periods where tasks with smaller periods have higher priorities. With the objective of meeting all tasks' deadlines, both EDF and RM adopt the *work conservation* strategy, which do not let the processor idle if there are ready tasks, and execute tasks as soon as possible.

For systems that have mixed workload with hard real-time periodic tasks and soft real-time aperiodic tasks, 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 [8]. To ensure that there is no deadline miss, EDL considers all instances of periodic tasks within the least common multiple (LCM) of their periods and generate an offline static schedule. For fixed-priority rate-monotonic scheduling, the *phase delay* technique was investigated where the arrival of tasks can be delayed for a certain offset without missing any deadline [1]. Based on this technique, the *dual-priority (DP)* scheme has been developed for rate-monotonic scheduling to improve the responsiveness of soft real-time aperiodic tasks [9].

The idea of delaying the execution of selected tasks has also been exploited in fault-tolerant systems [4,22]. As a common and effective fault-tolerance technique, the *primary/backup (PB)* approach normally schedules multiple copies (i.e., one as *primary* and others as *backup*) of a real-time task on different processors to tolerate a certain number of faults [17]. However, this technique can potentially consume significant system resources (e.g. CPU time and power). Thus, the backup copies are normally canceled as soon as their corresponding primary tasks complete successfully [5]. Hence, to reduce the execution overhead, backup tasks should be scheduled at their latest times to minimize the overlap with their corresponding primary tasks that run on different processors [4,22].

By dedicating one processor as the *spare* for backup tasks, Ejlali et al. studied a novel *Standby-Sparing (SS)* technique for dependent and aperiodic real-time tasks running on dual-processor systems with the goal of saving system energy consumption while tolerating a single permanent fault [10]. Based on the same idea of *separating* tasks on different processors, Haque et al. extended the standby-sparing technique to a more practical periodic task model based on the earliest deadline scheduling [14]. Here, to reduce the overlap between primary and backup copies of the same task, primary and backup tasks are scheduled according to EDF and EDL, respectively, on their dedicated processors [14]. Following this line of research, the fixed-priority (rate-monotonic priority) based standby-sparing scheme was studied in [15]. The generalized standby-sparing schemes for systems with more than two processors were investigated in [13].

Instead of dedicating a processor as the spare, it can be more efficient to allocate primary and backup copies of tasks in a mixed manner on both processors [12]. In this case, on each processor, the

primary tasks should be executed *as soon as possible (ASAP)* while the execution of backup tasks (whose primary tasks are on another processor) needs to be postponed *as late as possible (ALAP)*. Note that, the existing uniprocessor real-time scheduling algorithms normally schedule tasks *solely* based on their timing parameters. For instance, EDF and RM [16] schedule tasks at their earliest times while EDL [8] and DP [9] schedule them at their latest times. However, by treating all periodic tasks *uniformly*, none of the existing scheduling algorithms can effectively handle mixed tasks that have different *execution preferences*.

By considering both phase delay [1] and backup overloading [11,21] techniques, Bertossi et al. studied several schemes for fixed-priority rate-monotonic scheduling to improve system resource utilization and reduce the number of required processors to tolerate a given number of faults [4]. Here, backup tasks (whose primary tasks run on other processors) can be allocated in a mixed manner with other primary tasks on one processor (e.g., in the ARR1 scheme). However, the static nature of the phase delay technique [1] makes it not feasible for dynamic priority based scheduling (e.g., EDF). Based on EDF scheduling, Unsal et al. studied an offline *Secondary Execution Time Shifting (SETS)* heuristic which iteratively calculates the delayed release time for all backup task instances within the least common multiple (LCM) of tasks' periods to reduce the overlap between backup and primary tasks (and thus save system energy) [22].

In contrast to the scheduling algorithms in the existing work [4,22], we study the preference-oriented scheduling algorithms based on dynamic priority (i.e., deadline) of tasks running on uniprocessor systems. Specifically, by explicitly taking the execution preferences (ASAP/ALAP) of tasks into consideration when making scheduling decisions, we propose two online scheduling algorithms and show their optimality. We illustrate the application of the proposed schedulers in fault-tolerant systems and show that they can effectively reduce the overlap between primary and backup tasks.

3. 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 tasks' preferences, 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 $0 \leqslant t$ and $1 \leqslant i \leqslant n$. 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 a schedule in which no task instance misses its deadline [16].

We focus on 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 [16]. Similarly, when $\Psi_S = \emptyset$ (i.e., no task has ASAP preference), all tasks in Ψ can be optimally scheduled with the EDL algorithm [8].

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 $U \leqslant 1$ [8,16]. However, neither EDF nor EDL can effectively address the different preference requirements of various tasks.

4. PO-optimal 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 $(0 \leqslant t \leqslant LCM)$ 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 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 $\mathcal{S}$, $\Delta(\mathcal{S}^{opt}_{asap}, t) \geqslant \Delta(\mathcal{S}, t)$ at any time t $(0 \leqslant t \leqslant LCM)$.

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

As we show in Section 4.2, since ASAP and ALAP tasks may have conflicting demands in a schedule, in general it is not possible to find a feasible schedule which is both ASAP-Optimal and ALAP-

Optimal for a given task set. Therefore, we introduce the following *preference-oriented (PO) optimality* definitions, which capture the notions of how a schedule can be ASAP-Optimal while delaying the ALAP tasks as much as possible, or ALAP-optimal while executing the ASAP tasks as early as possible.

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

- $\Omega(\mathcal{S}^{opt}, t) \geqslant \Omega(\mathcal{S}^{opt}_{asap}, t)$ holds, where both $\mathcal{S}^{opt}$ and $\mathcal{S}^{opt}_{asap}$ are ASAP-optimal (denoted as **PO^S-optimal**); or,
- $\Delta(\mathcal{S}^{opt}, t) \geqslant \Delta(\mathcal{S}^{opt}_{alap}, t)$ 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 individual tasks with the same preference. That is, when determining the optimality of a feasible schedule, we can 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 existence of optimal schedules highly depends 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.

4.1. Harmonious PO-optimal schedules: $U = 1$

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 [16]. 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), this 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 ($0 \leqslant t \leqslant LCM$), 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 $\Delta(\mathcal{S}^{opt}, t) \geqslant \Delta(\mathcal{S}, t)$. Therefore, from the above equations, we can get:

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

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. □

4.2. Discrepant PO-optimal schedules: $U < 1$

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.

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 $T_1 = (1, 3), T_2 = (1, 4)$ 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 Fig. 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 Fig. 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.

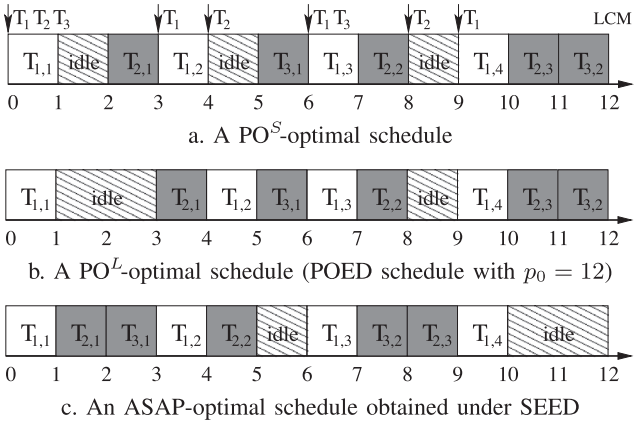


Fig. 1. An example task system with discrepant PO-optimal schedules; Here, $\Psi = \{T_1(1, 3), T_2(1, 4), T_3(1, 6)\}$; $\Psi_S = \{T_1\}$ and $\Psi_L = \{T_2, T_3\}$.

Note that, the schedule in Fig. 1a is not 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 Fig. 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 $[2,5)$ in two 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 their preferences.

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 as well as their conflicting requirements for the idle times in feasible schedules. This observation is formally presented as the remark below.

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.

5. An ASAP-optimal scheduler

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 should 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 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 an optimal preference-oriented scheduling algorithm, namely *ASAP-Ensured Earliest Deadline (SEED)*, for fully-loaded systems. In Section 6, 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.

5.1. SEED scheduling algorithm

To ensure that all ASAP tasks run as early as possible, SEED puts tasks with ASAP preference in the center stage when making scheduling decisions instead of scheduling the tasks *solely* based on their deadlines. That is, even if an ASAP task instance has later deadline than an ALAP task instance, SEED may schedule the ASAP task instance first if it is possible to delay the execution of the ALAP task. Therefore, to fully comply with the ASAP scheduling principle, the main steps of SEED are summarized in Algorithm 1.

Algorithm 1. The SEED Scheduling Algorithm

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

Here, SEED 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.

Recall that, from the definitions in Section 4, 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, at any time t, SEED is invoked after newly arrived tasks (if any) are added to their corresponding queues, 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, to facilitate the discussion later (Section 5.4) 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, 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 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 first execute the highest priority ASAP 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_l, T_k can be executed immediately (line 21). Otherwise, if we want to execute T_k by delaying the execution of T_l, it may delay potentially not only T_l but also other active ALAP tasks, and transitively other (ALAP or ASAP) task instances that arrive in the future with deadlines earlier than d_k. Therefore, 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.

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

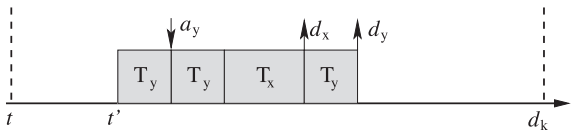
```
1: Input: [t, d_k], the look-ahead interval; Q_la, the queue of task
   instances in Ψ_la(t, d_k) with decreasing priority order;
2: while (Q_la ≠ ∅) do
3:   T_i = Dequeue(Q_la);//T_i has the highest priority with d_i
4:   //Suppose the (remaining) execution time of T_i is c_i;
5:   if (T_i ∈ Ψ_L) 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 ⩽ 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
```

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:

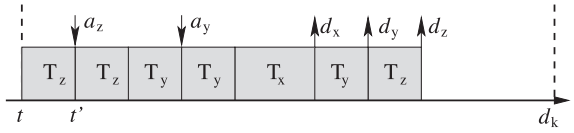
$$\Psi_{la}(t, d_k) = \{T_{i,j} | (T_{i,j} \in \mathcal{Q}_L(t) \vee a_{i,j} > t) \wedge d_{i,j} < d_k\} \quad (3)$$

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 all the task instances in $\Psi_{la}(t, d_k)$, the length of the CPU time that must be *reserved* before their respective deadlines is determined in the function $Mark(t, d_k, \mathcal{Q}_{la})$ (line 14). There are two possibilities for the result as illustrated in Fig. 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



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.

(line 16). Otherwise, if there is a *"free"* section starting at time t, T_k is deemed to be safe to run at that time (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' (with the help of a timer) to meet other tasks' deadlines.

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).

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 Fig. 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 Fig. 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 Fig. 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.

5.2. 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 $U \leqslant 1$, 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, when all active tasks have the same preference, the task with the earliest deadline will be executed next (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 ASAP task with the earliest deadline. 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 observation:

Observation 1. 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 2. *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 Observation 1, 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 $(r_k \leqslant t \leqslant d_k)$. 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 Observation 1, 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 2 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 $U \leqslant 1$, the SEED scheduler can successfully schedule all tasks without missing any deadline.*

Proof. Suppose that a task T_k arrives at time r_k and misses its deadline at d_k. From Lemma 2, we know that there is no task with a deadline later than d_k can be executed within $[r_k, d_k]$, which is defined as the *problematic interval*.

Let t_0 denote the last processor idle time before d_k. Note that, there must exist tasks 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 tasks that arrive no earlier than t_0 and have deadlines no later than d_k [3], is more than $(d_k - t_0)$, which contradicts with the condition of $U \leqslant 1$.

Moreover, there must exist tasks 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 tasks, 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 task with a deadline later than d_k that has been executed before r_k. This contradicts with our earlier findings that there must exist tasks 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 $U \leqslant 1$, the generated schedule under SEED is an ASAP-optimal schedule.*

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 $\Delta(\mathcal{S}, t) \geqslant \Delta(\mathcal{S}_{seed}, t)$ $(0 \leqslant t \leqslant LCM)$. Moreover, there must exist at least one interval during which ASAP tasks are executed in $\mathcal{S}$ but not in $\mathcal{S}_{seed}$. Assume $[t_1, t_2]$ $(0 \leqslant t_1 < t_2 \leqslant LCM)$ 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 fully-loaded systems with $U = 1$, SEED is essentially an optimal preference-oriented scheduler. Thus, we have the following theorem.

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

5.3. The improved SEED algorithm and 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 for the first 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 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 $T_x \in \mathcal{Q}_{la}$ such that there is no free section between t and T_x's deadline d_x. Define the *accumulated workload* for task instances in $\mathcal{Q}_{la}$ that has to be done before a given deadline D as:

$$W(D, \mathcal{Q}_{la}) = \sum_{T_i \in \mathcal{Q}_{la} \wedge d_i \leqslant D} c_i^{rem} \quad (4)$$

where c_i^{rem} denotes the remaining execution time of T_i. That is, we have $W(d_x, \mathcal{Q}_{la}) = d_x - t$. Otherwise, the first section is a "free" section and its size can be found as

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

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.

5.4. SEED for under-utilized systems

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 4, following the steps in Algorithms 1 and 2, its SEED schedule can be found as shown in Fig. 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 comply with the ASAP scheduling principle) also forces it to execute the ALAP tasks T_2 and T_3 at an earlier time. Such early executions of T_2 and T_3 make the resulting SEED schedule inferior to the PO^S-optimal schedule as shown in Fig. 1a.

6. Generalized POED scheduler

The preceding discussion shows that, despite the sophistication of the SEED algorithm, we need a more general framework to explicitly delay the ALAP tasks in under-utilized systems, by judiciously letting the processor idle. 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. POED 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.

However, unlike (genuine) ASAP tasks, the purpose of the dummy task is to simply introduce idle times into the schedule periodically and thus delay the execution of ALAP tasks. Therefore, to comply with the ASAP scheduling principle, the idle times introduced by the dummy task should not block (or delay) the execution of other active ASAP tasks even if the deadline of the current dummy task instance is earlier than those of the active ASAP tasks.

From another perspective, we can consider the idle times as *system slack*, which can be borrowed 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 [23]. 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, an idle interval will appear when the slack is consumed.

More details about wrapper-tasks can be found in [23], 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
```

6.1. POED scheduling algorithm

With the help of the two operations for wrapper-tasks, the major steps of POED for scheduling the augmented task set with the newly added dummy task 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 wrapper-tasks (to let the processor idle) and finally, active ALAP tasks.

Specifically, at any invocation time t, the slack time is first properly managed through the two wrapper-task operations. Here, if the processor is idle during last interval and the slack time is actually consumed, the corresponding wrapper-tasks are removed with the *RemoveSlack()* operation (line 4). Otherwise, if wrapped-execution occurs during last interval, the slack time is actually pushed forward by having a later deadline through the *AddSlack()* operation (line 6) [23]. Moreover, whenever a new instance of the dummy task arrives, its execution time is converted to slack time immediately with the *AddSlack()* operation as well (line 10).

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. Note that wrapper-tasks also compete for processor. Therefore, when the highest priority ASAP task has a later deadline than that of the wrapper-task, the processor will be allocated to the wrapper-task if the first section in the look-ahead interval is "free", which will wrap the execution of the ASAP task; otherwise, if the ASAP task has the earliest deadline, it is executed normally (line 17). For the case where there is no "free" section at the beginning of the look-ahead interval, urgent execution of active ALAP tasks will be performed to meet their deadlines (line 19).

When there is no active real ASAP task, POED will try to let the processor idle and delay the execution of ALAP tasks if possible. Recall that the dummy task has ASAP preference, which will be inherited by the wrapper-tasks. Therefore, for the wrapper-task with the earliest deadline, similar to the handling of other real ASAP tasks, a look-ahead interval is checked by constructing the corresponding look-ahead task instance set (line 22). If there is a "free" section at the beginning of the look-ahead interval (which means that all active ALAP tasks can be delayed), the processor will idle by consuming the slack time represented by the wrapper task (line 24). Otherwise, urgent execution of ALAP tasks is performed (line 26). Finally, when there are only active ALAP tasks, they are executed in the order of their priorities (line 29).

6.2. Analysis of the POED scheduler

Note that, POED can also be applied to task sets with full system utilization (i.e., $U = 1$). With the dummy task having $u_0 = 0$, there is no idle time and wrapped execution. Here, POED will reduce to SEED, which can optimally schedule all tasks without missing any deadline as shown in Section 5.

For under-utilized task sets where $U < 1$, the execution time of the dummy task is converted to slack at run-time, which is further represented by and managed through wrapper-tasks. However, from the discussions in [23], we know that such a wrapper-task based slack management mechanism does not introduce additional workload into the system. Moreover, from Algorithm 3, we can see that there are two possibilities for the execution of such wrapper-tasks: (a) consuming the corresponding slack time to let the processor idle; or (b) wrapping the execution of an ASAP task to push forward the corresponding slack to a later time.

When the slack time is actually consumed, it indicates that there is no active ASAP task and the execution time of active ALAP tasks (and future tasks) can be guaranteed before their corresponding deadlines. That is, the idling of the processor consumes the execution time of a wrapper-task and will not affect the timeliness of any real task. For the case of wrapped execution, the allocated time for the wrapper-task is actually *lent* to an ASAP task (that has a later deadline), which will return it at a later time (i.e., the slack is pushed forward). From another point of view, the idling of the processor (which supposes to happen at an earlier time) and the execution of the ASAP task (which supposes to happen at a later time) are essentially switched in the schedule to better fulfill our scheduling objectives. Therefore, following the similar reasonings as those for SEED and in [23], we have:

Theorem 4. *For any task set Ψ with $U < 1$ that is augmented with a dummy task T_0 (where $u_0 = 1 - U$), there is no deadline miss when the tasks are scheduled under POED.*

From Algorithm 3, we can also see that, when processing a wrapper-task to delay the execution of ALAP tasks, POED adopts the same steps as in SEED to scrutinize the look-ahead interval with the corresponding task instance set. Therefore, POED has the same complexity as that of SEED, which can be given as $\mathbf{O}\left(n \cdot \frac{p'_{max}}{p_{min}}\right)$ where $p'_{max} = \max\{p_{max}, p_0\}$. That is, the period of the dummy task can have a significant impact on the scheduling overhead of POED.

Intuitively, having a smaller period for the dummy task can reduce POED's scheduling overhead with a shorter look-ahead interval when a corresponding wrapper-task has a higher priority than those of the active ASAP tasks. However, when there is no active ASAP task and POED needs to decide the idle time of the processor, the smaller period can limit the amount of available slack (i.e., idle time) within the shorter look-ahead intervals. Thus the idling of the processor may not be able to delay the execution of ALAP tasks to the maximum extent, which results in a sub-optimal schedule.

On the other hand, if the dummy task's period is set as $p_0 = LCM$, we can always find the longest idle time for the processor whenever there is no active ASAP task, which can maximally delay the execution of ALAP tasks. That is, the resulting POED schedule is ALAP-optimal. Moreover, with the wrapper-tasks and wrapped executions, we know that POED will not let the processor idle whenever there are active ASAP tasks (see Algorithm 3). Hence, the resulting POED schedule is essentially PO^L-optimal when $p_0 = LCM$. In fact, the example in Fig. 1b is such a POED schedule. More formally, we have the following theorem.

Theorem 5. *For any task set Ψ with $U < 1$ that is augmented with a dummy task T_0 (where $u_0 = 1 - U$), the POED schedule is PO^L-optimal when the dummy task's period is $p_0 = LCM$.*

Proof. We first show that the resulting POED schedule $\mathcal{S}^{POED}$ is ALAP-optimal when $p_0 = LCM$.

Suppose that $\mathcal{S}^{POED}$ is **not** ALAP-optimal. Without loss of generality, we assume that a different feasible schedule $\mathcal{S}^{opt}_{alap}$ is ALAP-optimal. From Definition 2, we know that

$$\Omega(\mathcal{S}^{opt}_{alap}, t) \geqslant \Omega(\mathcal{S}^{POED}, t); \quad \forall t (0 \leqslant t \leqslant LCM)$$

Moreover, there must exist time t $(0 < t < LCM)$ such that $\Omega(\mathcal{S}^{opt}_{alap}, t) > \Omega(\mathcal{S}^{POED}, t)$. That is, there must exist time intervals during which ALAP tasks are executed in one schedule but not the other.

Suppose that $[t_1, t_2]$ $(0 < t_1 < t_2 < LCM)$ is the earliest such interval. It must be the case that, during the interval $[t_1, t_2]$, ALAP tasks are executed in the schedule $\mathcal{S}^{POED}$ but not in $\mathcal{S}^{opt}_{alap}$ (where the corresponding ALAP execution is performed at a later time). That is,

before time t_1, both schedules execute ALAP tasks (may not be the same) during exactly the same time intervals, which are denoted as *concurrent ALAP-intervals*.

In what follows, we define a set of task instances based on their executions in the POED schedule $\mathcal{S}^{POED}$ and show that there must be deadline miss for these task instances in the schedule $\mathcal{S}^{opt}_{alap}$, which contradicts with our assumption.

From Algorithm 3, we know that the execution of ALAP tasks during the interval $[t_1, t_2]$ under POED is *urgent*, which must be demanded by some task instances in the look-ahead interval to meet their timing constraints. Suppose that the earliest deadline of these task instances is d_z ($t_2 \leqslant d_z$; see Fig. 2b for an example). For the tasks in Ψ, define the set of their task instances that have deadlines no later than d_z as $\mathcal{Q}(\Psi, d_z)$. From Eqs. (4) and (5), we know that the remaining workload for the task instances in $\mathcal{Q}(\Psi, d_z)$ under POED at time t_1 is *exactly* $(d_z - t_1)$, which leaves no "free" section and causes urgent execution.

Note that, for the active ASAP task instances at time t_1 under POED, their deadlines should be later than d_z and they are not in the set $\mathcal{Q}(\Psi, d_z)$. That is, if there is any remaining ASAP workload for task instances in $\mathcal{Q}(\Psi, d_z)$ at time t_1 under POED, the ASAP task instances must arrive after t_1.

Moreover, POED follows the earliest deadline first principle and executes task instances in the non-decreasing order of their deadlines. Therefore, during the concurrent ALAP-intervals before time t_1, the executed workload for the ALAP task instances in $\mathcal{Q}(\Psi, d_z)$ under POED is no less than that in the schedule $\mathcal{S}^{opt}_{alap}$. Hence, for the schedule $\mathcal{S}^{opt}_{alap}$ at time t_1, the sum of the remaining ALAP workload for task instances in $\mathcal{Q}(\Psi, d_z)$ and the workload for future arrival ASAP task instances (if any) in $\mathcal{Q}(\Psi, d_z)$ is no less than $(d_z - t_1)$.

However, in our assumption, the processor is either idle or executing ASAP tasks in the schedule $\mathcal{S}^{opt}_{alap}$ during the interval $[t_1, t_2]$. In either case, there must exist an interval within $[t_1, t_2]$ during which the processor does not execute the remaining ALAP task instances or the future arrival ASAP task instances in $\mathcal{Q}(\Psi, d_z)$. That is, it is not possible for such task instances in $\mathcal{Q}(\Psi, d_z)$ to complete their executions within the interval $[t_1, d_z]$ in $\mathcal{S}^{opt}_{alap}$ and deadline misses will occur, which contradicts with our assumption that $\mathcal{S}^{opt}_{alap}$ is a feasible schedule. Hence, $\mathcal{S}^{POED}$ is ALAP-optimal.

Next, we show that $\mathcal{S}^{POED}$ is essentially a PO^L-optimal schedule. From Algorithm 3, once ASAP tasks arrive, they are executed at the earliest possible time with the wrapped-execution under POED and the processor can only be idle when there is no active ASAP tasks. Therefore, for any other ALAP-optimal schedule $\mathcal{S}^{opt}_{alap}$, the accumulated execution for ASAP tasks at any time t in $\mathcal{S}^{POED}$ will not be less than that of $\mathcal{S}^{opt}_{alap}$. From Definition 3, we know that $\mathcal{S}^{POED}$ is PO^L-optimal, which concludes our proof. □

7. An application of POED scheduler

Fault tolerance has been the traditional research topic for real-time systems, where tasks have to complete their executions successfully no later than their deadlines even in the presence of various faults [5,7,11]. In this section, as an example application of the POED scheduler, we illustrate how it can be applied in fault-tolerant systems to reduce execution overhead and thus improve system efficiency. Specifically, we consider a set of periodic real-time tasks running on a dual-processor system. To tolerate a single fault, a simple and well-studied approach is *hot-standby*, which runs two copies of the same task *concurrently* and *simultaneously* on the processors [17]. However, such an approach can be quite costly with 100% execution overhead.

In this work, we focus on an efficient *primary/backup (PB)* technique [5]. Here, with the objective of tolerating a single (permanent) fault, each (*primary*) task will have a *backup* task with the same timing parameters where the primary and backup of the same task have to be scheduled on different processors. To reduce the overlap between the primary and backup of the same task, the backup task should be scheduled at the latest time [14,22].

7.1. An example

As a concrete example, we consider two periodic tasks $T_1 = (3, 5)$ and $T_2 = (3, 10)$ that run on a dual-processor system. In our recently studied *Standby-Sparing (SS)* technique [14], all primary tasks are scheduled on a (primary) processor, while backup tasks are executed on another (*spare*) processor, as shown in Fig. 3a. Here, to reduce the overlapped executions of primary and backup copies of the same task, primary tasks are scheduled according to EDF on the primary processor while backup tasks are delayed as much as possible according to EDL on the spare processor [14].

Suppose that there is no failure during the execution of the tasks in the above example. The corresponding SS schedule within the LCM is shown in Fig. 3a, where the canceled (partial) backup (or primary) copies are marked with "X". Here, we can see that, there are five (5) units of overlapped executions. Compared to the hot-standby scheme [17] that would need nine (9) units of overlapped executions within one LCM, the execution overhead under the Standby-Sparing technique can be reduced to $\frac{5}{9} = 56\%$.

To further reduce the overlapped execution, instead of dedicating a processor as the spare, we can schedule the primary and backup copies of tasks in a mixed manner. For example, we can schedule the primary task T_1 and backup task B_2 on the first processor while backup task B_1 and primary task T_2 on the second processor, as shown in Fig. 3b. Here, the mixed task set on each processor are scheduled with the POED scheduler, where primary and backup tasks are assigned ASAP and ALAP preferences, respectively. In this case, when there is no failure during tasks' execution, there are only 3 units of overlapped execution. It leads to the execution overhead as $\frac{3}{9} = 33\%$, a 23% reduction on the overhead compared to that of the Standby-Sparing technique.

By reducing the execution overhead, we can reduce the energy consumption of the system and thus improve its energy efficiency. In particular, by exploiting the commonly used energy saving techniques, such as dynamic voltage scaling (DVS) and dynamic power management (DPM), we have studied an energy-efficient fault-tolerance scheme based on the POED scheduler in our recent work [12].

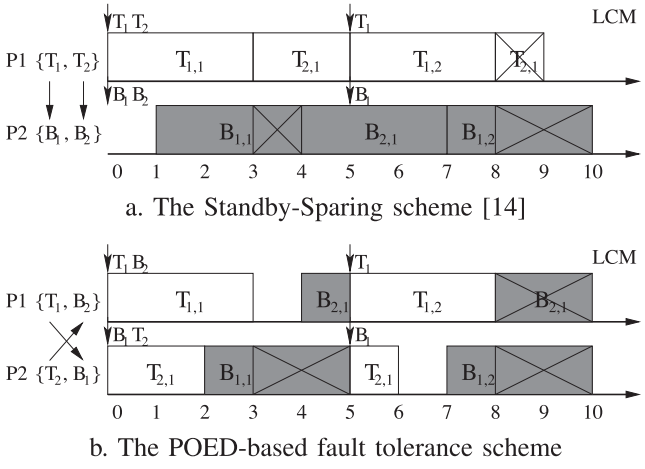


Fig. 3. An example with two tasks: $T_1 = (3, 5)$ and $T_2 = (3, 10)$ on a dual-processor system.

7.2. POED-based fault-tolerance technique

As shown in the example, to reduce the execution overhead for periodic real-time tasks tolerating a single fault in a dual-processor system, both processors will adopt the POED scheduler to execute the mixed tasks allocated to them. Specifically, the basic steps of the POED-based fault-tolerance technique can be summarized as follows (which are similar to [12]):

- **Step 1:** Partition primary tasks to the processors and mark them as *ASAP* tasks; (The worst-fit decreasing (WFD) heuristic is adopted to balance primary workload).
- **Step 2:** For each primary task, allocate its backup task to the processor other than its own processor and mark all the backup tasks as *ALAP* tasks.
- **Step 3:** On each processor, schedule the mixed set of primary tasks (with ASAP preference) and backup tasks (with ALAP preference) by the POED algorithm.

At runtime, the same as in other primary/backup schemes, the backup task on one processor will be canceled as soon as its corresponding primary task completes successfully on another processor. Moreover, the canceled backup tasks can generate slack time, which can be exploited to execute future primary tasks at earlier time. Such early execution of primary tasks can further cause more cancellation of backup tasks. More detailed discussions on cancellation of backup tasks at runtime under POED-based scheme can be found in [12]. The performance of the POED-based technique on reducing execution overhead is evaluated through simulations and discussed in Section 8.3.

8. 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 [6]. 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.

8.1. Scheduling overhead of SEED and POED

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 $n = 20$, Fig. 4a 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 (2.0 GHz) 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 5.5 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 Fig. 4b, 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 $p_{max} = 100$, Fig. 4c 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.

Fig. 4d 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 less ASAP tasks, it is more likely to have the look-ahead interval to be p_0, where the overhead is generally higher than other cases.

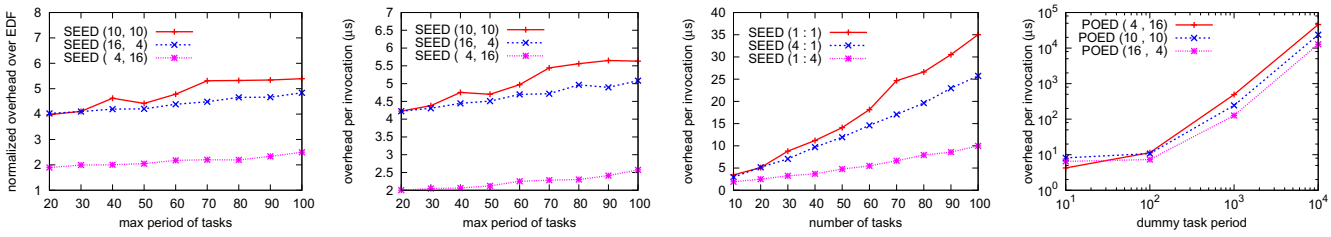
8.2. Fulfillment of preference requirements

In Section 4, the optimality of a schedule for tasks with ASAP and ALAP preferences has been defined based on the accumulated executions of those tasks over varying time intervals, 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:

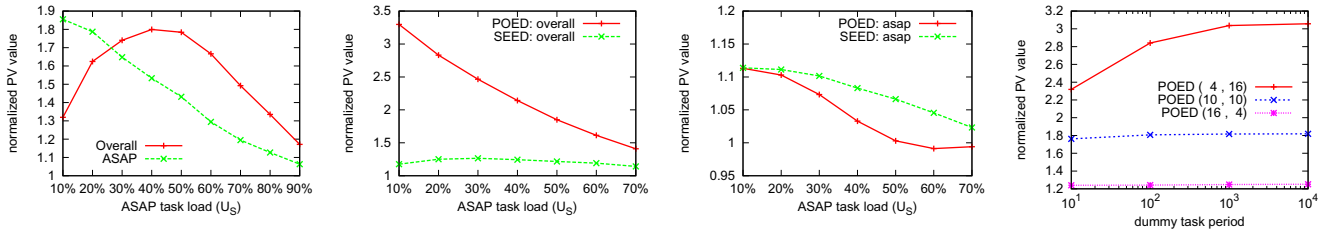
$$PV_{i,j} = \begin{cases} \frac{ft_{max} - ft}{ft_{max} - ft_{min}} & \text{if } T_i \in \Psi_S; \\ \frac{st - st_{min}}{st_{max} - st_{min}} & \text{if } T_i \in \Psi_L. \end{cases} \quad (6)$$

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



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

Fig. 4. Scheduling overheads of SEED and POED vs. EDF with $U = 1.0$ and 0.8, 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. 5. Normalized preference values achieved for tasks under SEED and POED ($p_{min} = 10, p_{max} = 100$ and $n = 20$).

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., $U = 1$), Fig. 5a 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 the 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 $U = 0.8$, Fig. 5b 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.

Fig. 5c 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 future ASAP tasks from executing early, especially when most tasks have ASAP preference.

For the case of $U = 0.8$, Fig. 5d shows the achieved PVs for all tasks with varying period (p_0) of the dummy task. 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. Moreover, having $p_0 = 10^2$ (i.e., p_{max}) is sufficient to achieve good PVs for tasks without incurring prohibitive scheduling overhead.

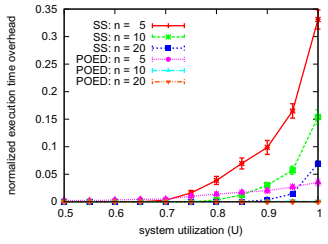


Fig. 6. Overheads for different fault-tolerance schemes.

8.3. Reduction of execution overhead with POED

Fig. 6 shows the normalized execution overhead for the Standby-Sparing and POED-based schemes under different system loads with different numbers of tasks per task set. Here, the execution overhead of the hot-standby scheme is used as the baseline [17]. From the results, we can see that, when the system load is low (i.e., $U \leqslant 0.7$), the normalized execution overhead is close to 0. The reason is that the primary copies of all task can complete before their corresponding backup copies start and almost all backup copies can be canceled under both schemes. However, when the system load is high (e.g., $U \geqslant 0.95$), the overhead of the POED-based scheme can be substantially lower than that of Standby-Sparing, especially for the cases with only a few tasks. The reason is that, the locations of the backup copies of tasks are fixed according to EDL in the Standby-Sparing scheme. However, as discussed previously, canceled backup copies generate slack, which can be exploited at runtime in the POED-based scheme to execute primary tasks at earlier times and further delay/cancel more backup tasks, which in turn leads to much reduced execution overhead.

9. Conclusions

In this paper, we introduced the concept of *preference-oriented (PO) execution*, where some tasks are ideally to be executed *as soon as possible (ASAP)*, while others *as late as possible (ALAP)*. We define different types of *PO-optimal* schedules and show their *harmonicity* and *discrepancies* for fully-loaded and under-utilized task systems, respectively. Then, for 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. We further illustrate how such preference-oriented schedulers can be applied to fault-tolerant systems to reduce execution overhead and improve system efficiency.

The proposed schedulers are evaluated through extensive simulations. The 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. For dual-processor fault-tolerant systems, the results further show that the POED-based technique can significantly reduce the execution

overhead (especially at high system loads) when comparing to the existing standby-sparing technique.

In our future work, we will study the extension of the preference-oriented scheduling framework to multiprocessor systems.

References

[1] N. Audsley, K. Tindell, A. Burns. The end of line for static cyclic scheduling? In Real-Time Systems, 1993. Proceedings., Fifth Euromicro Workshop on, pages 36 –41, Jun. 1993.

[2] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, L. Stougie, Scheduling real-time mixed-criticality jobs, IEEE Trans. Comput. (2011).

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

[4] A.A. Bertossi, L.V. Mancini, 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, IEEE Computer Society, Washington, DC, USA, 2006, pp. 107–118.

[5] A.A. Bertossi, L.V. Mancini, F. Rossini, Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems, IEEE Trans. Parallel Distrib. Syst. 10 (9) (1999) 934–945.

[6] E. Bini, G.C. Buttazzo. Biasing effects in schedulability measures, in: Proceedings of the Euromicro Conference on Real-Time Systems, 2004.

[7] A. Burns, R. Davis, S. Punnekkat, Feasibility analysis of fault-tolerant real-time task sets, in: Proceedings of the Eighth Euromicro Workshop on Real-Time Systems, 1996, June 1996, pp. 29–33.

[8] H. Chetto, M. Chetto, Some results of the earliest deadline scheduling algorithm, IEEE Trans. Softw. Eng. 15 (1989) 1261–1269.

[9] R. Davis, A. Wellings, Dual priority scheduling, in: Proceedings of the IEEE Real-Time Systems Symposium, 1995, pp. 100–109.

[10] A. Ejlali, B.M. Al-Hashimi, P. Eles, A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems, in: Proceedings of the IEEE/ACM Int'l Conference on Hardware/Software Codesign and System Synthesis, New York, NY, USA, 2009, pp. 193–202.

[11] S. Ghosh, R. Melhem, D. Mossé, Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems, IEEE Trans. Parallel Distrib. Syst. 8 (3) (March 1997) 272–284.

[12] Y. Guo, D. Zhu, H. Aydin, Efficient power management schemes for dual-processor fault-tolerant systems, in: Proceedings of the First Workshop on Highly-Reliable Power-Efficient Embedded Designs (HARSH), in Conjunction with HPCA, Feb. 2013.

[13] Y. Guo, D. Zhu, H. Aydin, Generalized standby-sparing techniques for energy-efficient fault tolerance in multiprocessor real-time systems, in: Proceedings of the IEEE Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Aug. 2013.

[14] M. Haque, H. Aydin, D. Zhu, Energy-aware standby-sparing technique for periodic real-time applications, in: Proceedings of the IEEE International Conference on Computer Design (ICCD), 2011.

[15] M.A. Haque, H. Aydin, D. Zhu, Energy management of standby-sparing systems for fixed-priority real-time workloads, in: Proceedings of the Second Int'l Green Computing Conference (IGCC), Jun. 2013.

[16] C.L. Liu, J. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, J. ACM 20 (January 1973) 46–61.

[17] D.K. Pradhan (Ed.), Fault-Tolerant Computer System Design, Prentice-Hall Inc, Upper Saddle River, NJ, USA, 1996.

[18] J. Regehr, A. Reid, K. Webb, M. Parker, J. Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis, in: Proceedings of the IEEE Real-Time Systems Symposium, 2003, pp. 25–36.

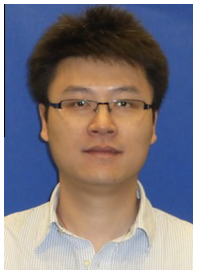
[19] I. Shin, I. Lee, Compositional real-time scheduling framework with periodic model, ACM Trans. Embed. Comput. Syst. 7 (3) (2008).

[20] H. Su, D. Zhu, An elastic mixed-criticality task model and its scheduling algorithm, in: Proceedings of the Design, Automation and Test in Europe (DATE), Mar. 2013.

[21] W. Sun, Y. Zhang, C. Yu, X. Defago, Y. Inoguchi, Hybrid overloading and stochastic analysis for redundant real-time multiprocessor systems, in: 26th IEEE International Symposium on Reliable Distributed Systems, 2007. SRDS 2007, oct. 2007, pp. 265–274.

[22] O.S. Unsal, I. Koren, C.M. Krishna, Towards energy-aware software-based fault tolerance in real-time systems, in: Proceedings of the Int'l Symposium on Low Power Electronics and Design, 2002, pp. 124–129.

[23] D. Zhu, H. Aydin, Reliability-aware energy management for periodic real-time tasks, IEEE Trans. Comput. 58 (10) (2009) 1382–1397.



Yifeng Guo received the PhD degree in Computer Science from the University of Texas at San Antonio in 2013. He currently works at NetApp Inc. His research interests include real-time systems, power management and fault tolerance.

Hang Su is currently a PhD Candidate in the Department of Computer Science at the University of Texas at San Antonio. His research interests include real-time systems, cyberphysical systems and mixed-criticality scheduling.

Dakai Zhu received the PhD degree in Computer Science from University of Pittsburgh in 2004. He is currently an Associate Professor in the Department of Computer Science at the University of Texas at San Antonio. His research is in the general area of real-time systems. He was a recipient of the US National Science Foundation (NSF) Faculty Early Career Development (CAREER) Award in 2010. He is a member of the IEEE and the IEEE Computer Society.

Hakan Aydin received the PhD degree in computer science from the University of Pittsburgh in 2001. He is currently an associate professor in the Computer Science Department at George Mason University. He was a recipient of the US National Science Foundation (NSF) Faculty Early Career Development (CAREER) Award in 2006. His research interests include real-time systems, low-power computing, and fault tolerance. He is a member of the IEEE.