Journal of Systems Architecture 000 (2016) 1–14

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

Preference-oriented fixed-priority scheduling for periodic real-time tasks

Rehana Begam[a], Qin Xia[b,*], Dakai Zhu[a,1], Hakan Aydin[c,2]

[a] Department of Computer Science, The University of Texas at San Antonio, San Antonio, TX 78249, USA
[b] School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an, Shaanxi, 710049, China
[c] Department of Computer Science, George Mason University, Fairfax, VA 22030, USA

A R T I C L E I N F O

Article history:
Received 6 May 2016
Revised 20 July 2016
Accepted 30 July 2016
Available online xxx

Keywords:
Real-time systems
Periodic tasks
Fixed-priority scheduling
Preference-oriented executions

A B S T R A C T

Traditionally, real-time scheduling algorithms prioritize tasks solely based on their timing parameters and cannot effectively handle tasks that have different *execution preferences*. In this paper, for a set of periodic real-time tasks running on a single processor, where some tasks are preferably executed *as soon as possible (ASAP)* and others *as late as possible (ALAP)*, we investigate *Preference-Oriented Fixed-Priority (POFP)* scheduling techniques. First, based on Audsley's *Optimal Priority Assignment (OPA)*, we study a *Preference Priority Assignment (PPA)* scheme that attempts to assign ALAP (ASAP) tasks lower (higher) priorities, whenever possible. Then, by considering the *non-work-conserving* strategy, we exploit the *promotion times* of ALAP tasks and devise an online dual-queue based POFP scheduling algorithm. Basically, with the objective of fulfilling the execution preferences of all tasks, the POFP scheduler retains ALAP tasks in the *delay queue* until their promotion times while putting ASAP tasks into the *ready queue* right after their arrivals. In addition, to further expedite (delay) the executions of ASAP (ALAP) tasks using system slack, runtime techniques based on dummy and wrapper tasks are investigated. The proposed schemes are evaluated through extensive simulations. The results show that, compared to the classical fixed-priority *Rate Monotonic Scheduling (RMS)* algorithm, the proposed priority assignment scheme and POFP scheduler can achieve significant improvement in terms of fulfilling the execution preferences of both ASAP and ALAP tasks, which can be further enhanced at runtime with the wrapper-task based slack management technique.

1. Introduction

Real-time systems, where application tasks generally have deadlines, have been studied for decades. For various task and system models, many classical real-time scheduling algorithms have been proposed by the research community. For example, for periodic real-time tasks running on a single processor system, *Rate Monotonic Scheduling (RMS)* and *Earliest Deadline First (EDF)* are the optimal schedulers for static and dynamic priority based preemptive scheduling algorithms, respectively [15]. With the exclusive focus on meeting tasks' timing constraints, most existing real-time scheduling algorithms prioritize and schedule tasks solely based on their timing parameters (e.g., deadlines and periods). Moreover, to complete the executions of tasks at their earliest possible time instants, these algorithms normally adopt the *work-conserving* strategy, which does not leave the processor idle when there are ready tasks for executions.

However, there are occasions where it can be beneficial to execute tasks at their latest times. For instance, when both *periodic* hard real-time tasks and *aperiodic* soft real-time tasks share the same computing platform, the executions of periodic hard real-time tasks can be maximally delayed in order to get better response times for aperiodic soft real-time tasks [6,7]. In addition, in fault-tolerant systems that adopt the *primary/backup* model, backup tasks should also be executed as late as possible to reduce the overlapped executions with their corresponding primary tasks (which are executed early on other processors) and thus system overheads to achieve cost efficient fault tolerance [12,17].

In contrast to the EDF and RMS schedulers [15] that execute tasks at their earliest possible times, the *Earliest Deadline Latest (EDL)* [6] and *Dual-Priority (DP)* schedulers [7] have been proposed to schedule periodic real-time tasks at their latest times. However, all these classical real-time schedulers treat **all** periodic tasks *uniformly* when assigning priorities to tasks and making scheduling

[*] Corresponding author.
E-mail addresses: qin.xia@mail.xjtu.edu.cn, xq_computer@sina.com (Q. Xia), dakai.zhu@utsa.edu (D. Zhu), aydin@cs.gmu.edu (H. Aydin).
[1] Member, IEEE
[2] Member, IEEE

http://dx.doi.org/10.1016/j.sysarc.2016.07.005
1383-7621/© 2016 Elsevier B.V. All rights reserved.

decisions. That is, these schedulers do *not* differentiate the execution preferences of tasks and thus **cannot** effectively handle co-running tasks that have different execution preferences.

For a set of periodic real-time tasks running on a single processor that have different execution preferences, where some tasks are preferably executed *as soon as possible (ASAP)* while others *as late as possible (ALAP)*, we have recently studied an online *Preference-Oriented Earliest Deadline (POED)* scheduling algorithm [10]. Basically, different from the EDF/EDL schedulers that consider only the deadlines of tasks when making scheduling decisions, the POED scheduler takes the execution preferences of tasks into consideration and puts ASAP tasks in the central stage. That is, whenever there are ready ASAP tasks at a scheduling point, these tasks can be picked for executions even if they have later deadlines, provided that such executions do not cause deadline misses for other tasks in the future. The POED scheduler has been exploited in fault-tolerant systems to reduce run-time overhead and improve energy efficiency [11]. In addition, such schedulers can also be utilized in mixed-criticality systems to differentiate the executions of high-criticality and low-criticality tasks to better explore system slack at runtime [16].

Although POED was the first scheduler that addresses the different execution preferences of periodic tasks, it is a dynamic priority based scheduler. To the best of our knowledge, no fixed-priority based preference-oriented scheduling algorithm has been reported yet in the literature. Our preliminary study on the *Preference-Oriented Fixed-Priority (POFP)* scheduler was presented in [4], which is extended in this paper to address the topic more thoroughly. First, based on Audsley's *Optimal Priority Assignment (OPA)* algorithm [3], we study a *Preference Priority Assignment (PPA)* scheme for a set of periodic real-time tasks with ASAP or ALAP execution preferences to be executed on a single processor system under preemptive fixed-priority scheduling. With the intuition that low priority tasks are executed late at runtime, the basic idea of PPA is to favor ALAP tasks when assigning lower priorities (and thus ASAP tasks can implicitly have higher priorities) without sacrificing system schedulability. By taking tasks' execution preferences into consideration, the resulting PPA priorities can most likely delay the executions of ALAP tasks while executing ASAP tasks at earlier times.

Then, observing that there are normally idle intervals in fixed-priority schedules, we propose an online preemptive POFP scheduler that adopts the *non-work-conserving* scheduling strategy. Here, to exploit idle intervals and systematically delay (expedite) the executions of ALAP (ASAP) tasks, POFP utilizes a *dual-queue* based scheduling approach. ASAP tasks enter the *ready queue* immediately after they arrive. In contrast, an ALAP task is put into a *delay queue* at its arrival time and will wait there until its *promotion time*, which can be derived following the ideas in the dual-priority scheduling framework [7]. After that, the ALAP task is *promoted* to the ready queue. Such a dual-queue scheduling approach can effectively prevent an ALAP task from being executed before its promotion time, which can in turn give low priority ASAP tasks the opportunity to run at earlier times.

Moreover, we investigate runtime techniques to further delay (expedite) the executions of ALAP (ASAP) tasks by exploiting system slack, which can be expected at runtime as real-time tasks typically take a small fraction of their worst-case execution times [9]. In summary, the contributions of this work are as follows:

- A preference priority assignment (PPA) scheme that explicitly incorporates the ASAP and ALAP execution preferences of periodic real-time tasks is proposed;
- An online dual-queue based preference-oriented fixed-priority (POFP) scheduler is proposed [4], which is preemptive and non-work-conserving in nature to better serve the tasks' ASAP and ALAP execution preferences;
- Runtime techniques are also investigated to further delay (expedite) the executions of ALAP (ASAP) tasks by exploiting system slack at runtime;
- The proposed PPA scheme, POFP scheduler and runtime techniques are evaluated through extensive simulations with synthetic tasks, which are shown to be very effective to fulfill the tasks' ASAP and ALAP execution requirements, when compared to that of the *preference-oblivious* RMS scheduler.

The rest of this paper is organized as follows. We review the closely related work in Section 2. Section 3 presents system models and necessary preliminaries. The preference-oriented priority assignment (PPA) scheme is discussed in Section 4. In Section 5, the preference-oriented fixed-priority (POFP) scheduler is proposed. The runtime techniques are further addressed in Section 6. Section 7 discusses the evaluation results and Section 8 concludes the paper.

2. Closely related work

In the past few decades, real-time systems have been studied extensively and numerous scheduling algorithms have been proposed for different task and system models. In this section, for periodic real-time tasks running on a single processor system, we review closely related scheduling algorithms.

The *Earliest-Deadline-First (EDF)* scheduling algorithm has been the well-known optimal scheduler based on the dynamic priority approach, where the instances (or jobs) of the same task may have different priorities [15]. Specifically, EDF prioritizes task instances (jobs) based on their absolute deadlines, where jobs with smaller deadlines have higher priorities (for jobs that have the same deadlines, the order of their priorities can be arbitrarily assigned without affecting tasks' schedulability). It has been shown that EDF can successfully schedule a set of periodic tasks on a single processor as long as the system utilization is no more than 1.0 [15].

In comparison, in the fixed-priority scheduling, priorities are assigned to tasks and the instances (jobs) of a task have the same priority of that task. For instance, as a classical optimal fixed-priority scheduler, the *Rate Monotonic Scheduling (RMS)* algorithm prioritizes tasks according to their periods, where tasks with smaller periods have higher priorities [15]. That is, if a set of periodic tasks can be feasibly scheduled with fixed-priority scheduling on a single processor, the task set can be successfully scheduled under RMS. Instead of prioritizing tasks solely based on their periods, in [3], Audsley studied an *Optimal Priority Assignment (OPA)* algorithm for periodic tasks under fixed-priority scheduling based on the concept of response time for tasks [1,13]. A comprehensive review on various priority assignment schemes for fixed-priority scheduling can be found in a recent report [8].

The well-known *work-conserving* schedulers EDF and RMS do not leave the processor idle if there are ready tasks for execution, where tasks are executed at their *earliest* times. However, for mixed workload with both hard real-time periodic tasks and soft real-time aperiodic tasks sharing a computing platform, it would be necessary for periodic tasks running at their *latest* times to provide better response times for soft aperiodic tasks. For such a purpose, the *earliest deadline latest (EDL)* algorithm has been developed [6]. By considering all instances of periodic tasks within the least common multiple (LCM) of their periods, EDL generates an offline static schedule to find out their latest execution times while ensuring that there is no deadline miss.

With the same objective, the *dual-priority (DP)* scheme has been developed for fixed-priority scheduling in order to improve the responsiveness of soft real-time aperiodic tasks [7]. The DP scheduler is based on the *phase delay* technique [2], where the arrivals of periodic tasks are delayed for a certain offset without missing any deadline.

For fault-tolerant systems that adopt the *Primary/Backup (PB)* technique, Unsal et al. studied an offline *Secondary Execution Time Shifting (SETS)* scheduling heuristic based on the EDF scheduler [17]. Here, to reduce the overlapped executions between backup and primary copies of the same task (and thus save system energy), SETS iteratively calculates the delayed release time for all backup instances within the *Least Common Multiple (LCM)* of tasks' periods.

As the most closely related work, for a set of periodic tasks running on a single processor system with either ASAP or ALAP execution preferences, we have studied a dynamic-priority based *Preference-Oriented Earliest Deadline (POED)* scheduling algorithm [10]. In this paper, we study an online *Preference-Oriented Fixed-Priority (POFP)* scheduling algorithm and related priority-assignment techniques, which are different from the dynamic-priority based POED scheduler [10]. Moreover, this work is also different from the classical real-time schedulers (such as EDF, RMS [15], EDL [6] and DP [7]), which cannot effectively handle the different execution preferences of tasks.

3. System models and preliminaries

In this section, the task and system models are first presented. Then, we review preliminaries related to fixed-priority scheduling, followed by the description of the problem to be studied in this work.

3.1. Task and system models

We consider a set of n independent periodic real-time tasks $\Psi = \{T_1, \ldots, T_n\}$ to be executed on a single processor system. The tasks do not share resources other than the processor. A task T_i is represented by a tuple (c_i, p_i, θ_i). Here, c_i is its worst-case execution time (WCET) and p_i is its period (i.e., inter-arrival time). We consider tasks with implicit deadlines where p_i is also the relative deadline of task T_i. The utilization of task T_i is defined as $u_i = \frac{c_i}{p_i}$ and the system utilization is further defined as $U = \sum_{T_i \in \Psi} u_i$.

Each task T_i represents an infinite sequence of *task instances* (or *jobs*), where the jth instance of task T_i is denoted as $T_{i,j}$ $(j = 1, 2, \ldots)$. Tasks are assumed to be synchronous with the first instance of each task arriving at time 0. Hence, the task instance $T_{i,j}$ arrives at time $r_{i,j} = (j-1) \cdot p_i$ and has an absolute deadline as $d_{i,j} = r_{i,j} + d_i = j \cdot p_i$, where $[r_{i,j}, d_{i,j}]$ denotes the *active* window of $T_{i,j}$. Note that, there is only one active task instance for each task at any given time instant. In what follows, we use task T_i to refer to its current active instance when there is no ambiguity.

The *execution preference* of task T_i is denoted as θ_i, which can be either *as soon as possible (ASAP)* or *as late as possible (ALAP)* [10]. Based on their execution preferences, the tasks in Ψ can be partitioned into two subsets Ψ_S and Ψ_L (where $\Psi = \Psi_S \cup \Psi_L$), which contain the tasks with ASAP and ALAP preferences, respectively. Note that, when all tasks have ASAP (or ALAP) preference, they can be effectively scheduled by the RMS [15] (or Dual-Priority [7]) scheduler, respectively. Hence, in this work, we focus on task sets that have both ASAP and ALAP tasks, where the existing fixed-priority schedulers cannot handle them effectively.

3.2. FP scheduling and response time analysis

In fixed-priority (FP) scheduling, the key step is to assign each task T_i a static priority η_i, which will be utilized by all its instances. Here, the priorities assigned to tasks have to be different to ensure that tasks can be totally ordered [8]. That is, for any two tasks T_i and T_j $(i \neq j)$, either $\eta_i > \eta_j$ or $\eta_i < \eta_j$ holds, corresponding to the cases where T_i's priority is higher or lower than T_j's priority, respectively.

Once the priorities for all tasks are assigned, the set of tasks that have higher priorities than that of T_i can be defined as:

$$hp(T_i) = \{T_x | T_x \in \Psi \wedge \eta_i < \eta_x\} \tag{1}$$

In the classical preemptive FP scheduler (e.g., RMS [15]), a single ready queue is utilized to hold all active tasks at runtime. If there are active tasks and the ready queue is not empty, the one that has the highest priority will be chosen for execution. However, the execution of an active task T_i can be preempted when a high priority task $T_j \in hp(T_i)$ arrives. Under preemptive scheduling, the *response time* of T_i can be found iteratively through the following equation [1,13]:

$$R_i^{k+1} = \sum_{T_j \in hp(T_i)} \left\lceil \frac{R_i^k}{p_j} \right\rceil c_j + c_i \tag{2}$$

where $R_i^0 = c_i$. When the above equation converges with $R_i^{k+1} = R_i^k$, the response time of task T_i can be set accordingly as $R_i = R_i^k$.

Note that, for a given priority assignment, if $R_i \leq p_i$ (i.e., the response time does not exceed the relative deadline) for every task $T_i \in \Psi$, the task set can be successfully scheduled under the preemptive fixed-priority scheduling [1,13]. More formally, we can have the following lemma.

Lemma 1. *For the preemptive fixed-priority scheduling, a priority assignment for the tasks in* Ψ *is* feasible *if there are* $R_i \leq p_i \, \forall T_i$.

3.3. Problem description

Unlike tasks' deadlines that represent hard constraints of tasks, the execution preferences of tasks are rather *soft* requirements, which provide only guidelines on how early or late a task should be *preferably* executed. Based on the aggregated executions of ASAP or ALAP tasks within a certain interval, we have formally defined optimal schedules in terms of fulfilling the preference requirements of tasks [10]. However, it has been shown that, due to the conflicting demands of ASAP and ALAP tasks regarding the placement of idle times in a schedule, it may be impossible to find a feasible schedule that can optimally fulfill the preference requirements of both ASAP and ALAP tasks simultaneously [10].

Therefore, instead of aiming at finding optimal fixed-priority scheduling algorithms, in this work, we focus on investigating various techniques to improve the fulfillment of tasks' execution preferences. Specifically, for a set of periodic tasks running on a single processor under fixed-priority scheduling, we study *priority assignment, scheduling algorithms* and *runtime techniques* with the objective of better fulfilling tasks' execution preferences in the resulting schedule.

4. Preference priority assignment

In fixed-priority scheduling, the execution order of tasks (i.e., *when* tasks are executed) depends directly on their priorities. Therefore, in this section, we first study a *Preference Priority Assignment (PPA)* scheme, which considers tasks' execution preferences when assigning priorities to tasks.

In general, when a task arrives and becomes ready for execution, the higher priority it has, the earlier it can be executed, and vice versa. Thus, intuitively, we should assign higher and lower priorities to ASAP and ALAP tasks, respectively, to satisfy their execution preferences. However, such a priority assignment has to consider the schedulability of tasks to avoid deadline misses. That is, for tasks with mixed ASAP and ALAP execution preferences, an effective priority assignment needs to *simultaneously* consider both the schedulability of tasks and their execution preferences.

As mentioned earlier, an *Optimal Priority Assignment (OPA)* algorithm has been proposed in [3] based on the response times of tasks [1,13]. Note that, the response time of a task T_i depends only on $hp(T_i)$, the set of tasks that have higher priorities, rather than their relative priority order (see Eq. (2) in the last section). Based on this property of task's response time, the basic idea of OPA is as follows: by assuming that other unassigned tasks are given higher priorities, OPA iteratively finds a task that can be feasibly assigned the next lowest priority level (where its response time is no more than its period). Once OPA successfully assigns a priority for each task, a feasible priority assignment is found. Otherwise, if there is no task that can be feasibly assigned the next lowest priority level in one iteration, the task set is deemed to be *unschedulable* under the fixed-priority scheduling [3].

4.1. The PPA algorithm

Without considering the execution preferences of tasks, OPA could assign an ASAP task the lowest priority during an iteration when it is possible to assign this lowest priority to an ALAP task. By extending the idea of OPA and incorporating the execution preferences of tasks, we propose a *Preference Priority Assignment (PPA)* algorithm, where its major steps are shown in Algorithm 1.

The key idea behind PPA is to **favor ALAP tasks for lower priorities** (which in turn can leave higher priorities for ASAP tasks). The PPA algorithm starts with the lowest priority and all tasks in a *un-assigned* task set Ψ^{un} (line 3). To find an *eligible* task to assign the next lowest priority level, PPA first checks the ALAP tasks that have not been assigned a priority yet with the helper function *findEligibleTask()*. If such a task T_x exists, it is assigned the lowest priority and is removed from Ψ^{un} (lines 6 to 8). Then, the algorithm continues for the next lowest priority if Ψ^{un} still contains un-assigned tasks (line 9).

In case there is no ALAP task that is eligible to take the lowest priority, PPA checks ASAP tasks with the same helper function *findEligibleTask()* (line 11). Similarly, if an eligible ASAP task is found, it takes the lowest priority and is removed from Ψ^{un} and PPA continues for the next lowest priority (lines 12 to 14). However, if there is no eligible ASAP task to take the lowest priority either, PPA fails to find a feasible priority assignment for the tasks

Algorithm 1 : The PPA algorithm.

```
1: Input: $\Psi$, $\Psi_S$, $\Psi_L$ and $(c_i, p_i, \theta_i)$ $\forall T_i \in \Psi$;
2: Output: FAIL or $\{\eta_i | \forall T_i \in \Psi\}$;
3: $\eta^{next} = 1$; $\Psi^{un} = \Psi$; //initialization
4: while ($\Psi^{un} \neq \emptyset$) do
5:    $T_x = findEligibleTask(\Psi^{un}, \Psi_L)$;//check ALAP tasks
6:    if ($T_x \neq NULL$) then
7:       $\eta_x = \eta^{next}$; //task $T_x$ has the lowest priority
8:       $\Psi^{un} = \Psi^{un} - \{T_x\}$; $\eta^{next}++$;
9:       continue; //continue with the next lowest priority
10:   end if
11:   $T_x = findEligibleTask(\Psi^{un}, \Psi_S)$;//check ASAP tasks
12:   if ($T_x \neq NULL$) then
13:      $\eta_x = \eta^{next}$; //task $T_x$ has the lowest priority
14:      $\Psi^{un} = \Psi^{un} - \{T_x\}$; $\eta^{next}++$;
15:   else
16:      return FAIL; //no task can have the lowest priority
17:   end if
18: end while
19: return $\{\eta_i\}$; //a feasible priority assignment is found
```

Algorithm 2 : *findEligibleTask*(Ψ^{un}, Ψ^{target}) function.

```
1: $\Psi^{eligible} = \emptyset$; //eligible tasks for next lowest priority
2: for ($\forall T_x \in (\Psi^{un} \cap \Psi^{target})$) do
3:    $hp(T_x) = \Psi^{un} - \{T_x\}$;//assume higher priority task set
4:    Find $R_x$ from Equation 2;
5:    Add $T_x$ to $\Psi^{eligible}$ if there is $R_x \leq p_x$;
6: end for
7: if ($\Psi^{eligible} \neq \emptyset$) then
8:    $\forall T_x \in \Psi^{eligible}$, find $T_k$ with the largest $(p_k - R_k)$;
9:    return $T_k$; //return an eligible unassigned task
10: else
11:   return NULL; //no eligible unassigned task in $\Psi^{target}$
12: end if
```

(line 16). Once Ψ^{un} becomes empty and all tasks have been assigned their priorities, PPA returns the feasible priority assignment $\{\eta_i | \forall T_i \in \Psi\}$.

The helper function *findEligibleTask()* is further detailed in Algorithm 2, which takes the un-assigned task set Ψ^{un} and a target task group Ψ^{target} (which can be either Ψ_S or Ψ_L) as parameters. For each un-assigned task T_x in the target task group (line 2), its response time can be found using Eq. (2) by assuming that all other un-assigned tasks have higher priorities (lines 3 and 4). The task T_x is *eligible* (to take the lowest priority) if its response time is no more than its period (i.e., $R_x \leq p_x$), which is also its relative deadline.

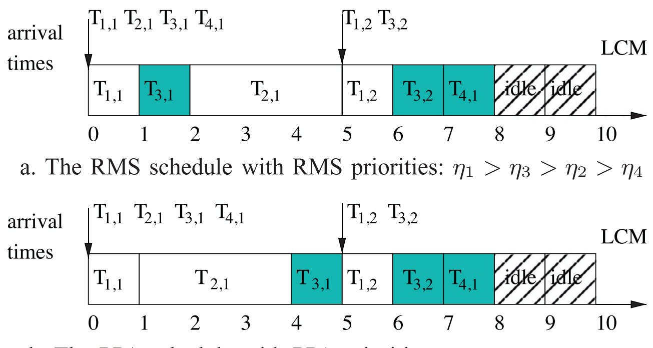
Note that, *any* eligible task could be returned to take the lowest priority. Here, PPA adopts the heuristic that returns the task with the largest value of $(p_k - R_k)$ when there are more than one eligible un-assigned tasks in the target task group (lines 8 and 9). This allows other tasks to have higher priorities with reduced response times, which can potentially balance the value of $(p_k - R_k)$ for the tasks. In case the target task group does not have any eligible un-assigned task, the function returns *NULL* (line 11).

We would like to point out that, as a variation of OPA [3], PPA is also an *optimal* priority assignment algorithm. That is, PPA can find a feasible priority assignment for the tasks in fixed-priority scheduling *if and only if* such a priority assignment exists. From Algorithm 1, we can see that the response times of un-assigned tasks *monotonically* decrease after each iteration since there will be fewer tasks in their high-priority task sets. Hence, when there are more than one eligible tasks in an iteration, the selection of *any* of them for the next lowest priority does not affect the eligibility of other tasks in the next iteration. Therefore, the separate consideration of ALAP and ASAP tasks in PPA does not affect its optimality for finding a feasible priority assignment.

4.2. An example: PPA vs. RMS

We illustrate the advantages of PPA on improving the fulfillment of tasks' execution preferences through a concrete example. The example task set has four tasks: $T_1(1, 5)$, $T_2(3, 10)$, $T_3(1, 5)$ and $T_4(1, 10)$, where tasks T_1 and T_2 have ASAP preference and T_3 and T_4 have ALAP preference (i.e., $\Psi_S = \{T_1, T_2\}$ and $\Psi_L = \{T_3, T_4\}$).

For comparison, we first consider the well-known RMS scheduler [15], which is preference-oblivious and assigns priorities to tasks solely based on their periods. More specifically, tasks with larger periods are assigned lower priorities, and vice versa. When there are more than one tasks that have the same period, tie can be broken arbitrarily without affecting the schedulability of tasks [15]. Here, we assume that RMS assigns a higher priority to the task with smaller index. Hence, for the example task set, tasks' RMS priorities can be found as $\eta_1 > \eta_3 > \eta_2 > \eta_4$. Fig. 1a



a. The RMS schedule with RMS priorities: $\eta_1 > \eta_3 > \eta_2 > \eta_4$

b. The PPA schedule with PPA priorities: $\eta_1 > \eta_2 > \eta_3 > \eta_4$

Fig. 1. An example with four tasks.

further shows the RMS schedule of the tasks within the LCM (Least Common Multiple) of task periods.

PPA, according to Algorithm 1, first checks the two ALAP tasks T_3 and T_4 as candidates for the lowest priority level. From Algorithm 2, the response times for T_3 and T_4 can be found as $R_3 = 7$ and $R_4 = 8$, respectively. Task T_3 is not eligible since $R_3 > p_3$. Therefore, only task T_4 is eligible and it is assigned the lowest priority. After that, as an ALAP task, T_3 is checked first again for the next lowest priority. In this iteration, we can get $R_3 = 5 = p_5$ and task T_3 becomes eligible. Hence, T_3 is assigned the second lowest priority.

Continuing with Algorithm 1, we can find the feasible PPA priorities of tasks as $\eta_1 > \eta_2 > \eta_3 > \eta_4$. Even though T_3 has a smaller period than T_2, as an ALAP task it is assigned a lower priority than the ASAP task T_2, which is different from that of the traditional RMS approach. The PPA schedule of the tasks is shown in Fig. 1b.

Compared to the RMS schedule in Fig. 1a, we can see that the execution of the ALAP task instance $T_{3,1}$ is delayed to its latest time while ASAP task instance $T_{2,1}$ is executed at an earlier time. That is, by incorporating tasks' execution preferences into priority assignment, PPA can fulfill the execution preferences of tasks T_2 and T_3 in a better way when compared to the preference-oblivious RMS scheduler.

5. Preference-oriented FP scheduler

To effectively address the execution preferences of ASAP and ALAP tasks, we have identified two basic principles for designing preference-oriented scheduling algorithms [10]: a) at any time t, if there are ready ASAP tasks, the scheduler should not let the processor idle; and b) at any time t, if all ready tasks are ALAP tasks, the scheduler should let the processor stay idle if it is possible to do so without causing any deadline miss for current and future task instances.

Here, the second design principle conflicts directly with the *work-conserving* strategy adopted in the classical fixed-priority scheduling, where the processor becomes idle only if there is no ready task for execution. For instance, from Fig. 1, we can see that both RMS and PPA schedules have two units of idle time at the end after all tasks finish their executions. Here, for the ALAP task instances $T_{3,2}$ and $T_{4,1}$, their executions could be further delayed by exploring such idle time without missing their deadlines.

In this work, focusing on the second design principle, we propose a *non-work-conserving Preference-Oriented Fixed-Priority (POFP)* scheduling algorithm. Before presenting the algorithm, in what follows, we first investigate how to find the safe amount of delay for ALAP tasks without violating their timing constraints under the fixed-priority scheduling.

5.1. Promotion times for delayed executions

Based on the *phase delay* technique [2], the concept of *promotion time* was introduced for periodic tasks in the *Dual-Priority (DP)* scheduling algorithm [7]. In that work, the promotion time of a periodic task defines the *longest* time the task can safely wait, after its arrival, before entering the ready queue without missing its deadline. This provides a crucial guideline for the DP scheduler to postpone the executions of hard real-time periodic tasks with the objective of improving the response time of soft real-time aperiodic tasks when they are executed on the same processor [7].

Hence, it is critical to properly derive the safe promotion time for periodic tasks. From Lemma 1 (see Section 3), we know that, for a given feasible priority assignment in fixed-priority scheduling, the response time for any task in Ψ will be no more than its period (i.e., $R_i \le p_i, \forall T_i \in \Psi$) [1,13]. The promotion time of task T_i is formally defined as [7]:

$$\gamma_i = p_i - R_i \tag{3}$$

That is, upon arrival, any instance of task T_i can be safely delayed for γ_i time units before entering the ready queue without missing its deadline [2,7].

In this work, we also exploit promotion times to systematically delay the executions of ALAP tasks without causing any deadline miss. Specifically, once a feasible priority assignment for all tasks is given, the promotion times for (*only*) ALAP tasks are calculated according to Eq. (3).

5.2. The POFP scheduling algorithm

The POFP scheduler utilizes two runtime queues to handle the different execution preferences of tasks: the *ready* and *delay* queues, which are denoted as $\mathcal{Q}_R$ and $\mathcal{Q}_D$, respectively. As in other schedulers, the ready queue $\mathcal{Q}_R$ holds the tasks that can be immediately executed, in the order of their priorities. ASAP tasks enter the ready queue $\mathcal{Q}_R$ in order to be executed quickly. The second queue $\mathcal{Q}_D$ is used to temporarily hold ALAP tasks upon their arrival. Each ALAP task stays in $\mathcal{Q}_D$ until its promotion time, at that time the task is *promoted* to the ready queue $\mathcal{Q}_R$ and becomes eligible for execution.

Basically, POFP leverages the delay queue $\mathcal{Q}_D$ to postpone the executions of ALAP tasks until their promotion times, which also provides opportunities for low priority ASAP tasks to get executed at earlier times. Unlike the work-conserving Dual-Priority scheduler [7], POFP leaves the processor idle as long as the ready queue $\mathcal{Q}_R$ is empty, regardless of the contents of the delay queue $\mathcal{Q}_D$. Since the tasks are independent, an ALAP task is technically *ready* for execution whenever it arrives. Hence, POFP is a *non-work-conserving* scheduler.

The basic steps of the POFP scheduler are given in Algorithm 3, which is invoked at a few occasions involving task T_k: a) the arrival time of task T_k; b) the completion of task T_k; and, c) when an ALAP task T_k is promoted from the delay queue $\mathcal{Q}_D$ to the ready queue $\mathcal{Q}_R$. When an ALAP task T_k arrives, it is put into the delay queue by the function *Enqueue*(T_k, $\mathcal{Q}_D$), where a timer with its promotion time is set (lines 2 and 3). Observe that, if the promotion time of T_k is $\gamma_k = 0$, it will be promoted immediately after its arrival.

When a task T_k completes its execution, POFP will execute the next highest-priority task in the ready queue $\mathcal{Q}_R$ (line 6). However, if there is no ready task in $\mathcal{Q}_R$, POFP lets the processor idle (line 8), which effectively delays the execution of ALAP tasks in the delay queue until their promotion times. When an ALAP task is promoted or an ASAP task arrives, it preempts the currently running task T_c if it has a higher priority than that of T_c (lines 12 and 13); otherwise, the task is inserted to the ready queue (line 15).

Algorithm 3 : The POFP scheduling algorithm.

```
1: Input: $\{c_i, p_i, \eta_i\}$ for $\forall T_i \in \Psi$and $\gamma_i$ for $\forall T_i \in \Psi_L$; Invocation after anevent at time $t$ involving task instance $T_k$; The current runningtask instance is denoted by $T_c$;
2: if ($T_k \in \Psi_L$ arrives at time $t$) then
3:     Enqueue($T_k$, $\mathcal{Q_D}$); SetTimer($\gamma_k$);
4: else if ($T_k$ completes at time $t$) then
5:     if (Ready queue $\mathcal{Q_R}$ is not empty) then
6:         $T_k$ = Dequeue($\mathcal{Q_R}$); Execute($T_k$);
7:     else
8:         Let the processor idle; //regardless of tasks in $\mathcal{Q_D}$
9:     end if
10: else
11:     //$T_k \in \Psi_L$ is promoted OR $T_k \in \Psi_S$ arrives at time $t$
12:     if ($\eta_k > \eta_c$) then
13:         Enqueue($T_c$, $\mathcal{Q_R}$); Execute($T_k$);//$T_k$ preempts $T_c$
14:     else
15:         Enqueue($T_k$, $\mathcal{Q_R}$); //Insert $T_k$ to ready queue $\mathcal{Q_R}$
16:     end if
17: end if
```

5.3. Analysis of the POFP scheduler

From the algorithm, we can see that, when an ALAP task is promoted from the delay queue to ready queue, the processing of such a promotion event is similar to that of a normal task arrival event. Compared to the classical fixed-priority scheduler (i.e., RMS), only the promotion events for ALAP tasks are additional scheduling events for POFP. Therefore, the run-time complexity of POFP will be at the same level as that of the preemptive fixed-priority scheduler.

Moreover, when there is no ALAP task in a task set, POFP reduces to the classical preemptive fixed-priority scheduler. Based on the results related to phase delay [2] and promotion time [7], the delayed executions of ALAP tasks in POFP will not cause deadline misses for such tasks. Therefore, for a set of periodic tasks that have either ASAP or ALAP execution preferences, as long as the task set is schedulable and has a feasible priority assignment (e.g., either RMS or PPA), the tasks can be successfully scheduled under POFP.

As a generic preemptive fixed-priority scheduler, POFP can be applied to any feasible priority assignment of schedulable task sets. In what follows, as the exemplary optimal priority assignments, we focus on RMS and PPA, which are preference-oblivious and preference-aware, respectively.

5.4. An example: PORMS vs. POPPA

We illustrate how the POFP scheduler works by considering the task set introduced in Section 4.2. Here, both RMS and PPA priorities are used for the purpose of comparison, where the corresponding scheduler instances are denoted as PORMS and POPPA, respectively.

First, for PORMS, recall that tasks' priorities are set as $\eta_1 > \eta_3 > \eta_2 > \eta_4$. Based on Eqs. (2) and (3), the promotion times for the ALAP tasks T_3 and T_4 can be found as: $\gamma_3 = 3$ and $\gamma_4 = 2$, respectively. From Fig. 2a, we can see that, when the ALAP task instances $T_{3,1}$ and $T_{4,1}$ arrive, they are put into the delay queue to prevent their immediate execution. In comparison, the ASAP task instances $T_{1,1}$ and $T_{2,1}$ enter the ready queue right after their arrivals.

Once the highest priority $T_{1,1}$ completes its execution at time 1, the ready queue has only one task instance $T_{2,1}$, which is picked for execution next. Although $T_{3,1}$ has higher priority, it is forced to stay in the delay queue until its promotion time (i.e., time 3) since it is an ALAP task instance. At time 2, $T_{4,1}$ is promoted to the ready queue, but it has lower priority than that of the current running task instance $T_{2,1}$. When $T_{3,1}$ is promoted at time 3, it has higher priority and will preempt the execution of $T_{2,1}$.

At time 5, both $T_{1,2}$ and $T_{3,2}$ arrive. Again, the ASAP task instance $T_{1,2}$ enters the ready queue for immediate execution while the ALAP task instance $T_{3,2}$ is put in the delay queue and has to wait there until time 8 (i.e., $\gamma_3 = 3$ units after its arrival). Here, when the task instance $T_{4,1}$ finishes its execution at time 7, the ready queue is empty and the processor becomes idle. Although the ALAP task instance $T_{3,2}$ in the delay queue is ready for execution, it is effectively forced to wait until time 8.

The final PORMS schedule is shown at the bottom of Fig. 2a. Compared to the RMS schedule as shown in Fig. 1a, the executions of both ALAP task T_3's instances are delayed in the PORMS schedule. Moreover, part of the ASAP task instance $T_{2,1}$ is executed at an earlier time. Therefore, by exploiting the promotion times for ALAP tasks to delay their executions, the PORMS scheduler can fulfill the execution preferences of tasks in a better way when compared to the preference-oblivious RMS scheduler.

For POPPA, where the priorities are $\eta_1 > \eta_2 > \eta_3 > \eta_4$, the promotion times for the ALAP tasks T_3 and T_4 can be found as $\gamma_3 = 0$ and $\gamma_4 = 2$, respectively. Here, we can see that, although the ALAP task T_3 takes a lower priority in POPPA, the increased response time for this task makes its promotion time to be 0. This means that, the instances of the ALAP task T_3 have to enter the ready queue right after their arrivals, and cannot take advantage of the delay queue in the POPPA scheduler to postpone their executions.

The states of the runtime queues, transitions of tasks and the final POPPA schedule are shown in Fig. 2b. It is interesting to see that, for this particular task set, its POPPA schedule is the same as its PPA schedule as shown in Fig. 1b. Since its promotion time is 0, the ALAP task T_3 is scheduled the same way in both POPPA and PPA. We observe that the promotion time of the ALAP task T_4 is $\gamma_4 = 2$; but this time interval is not long enough to keep it in the delay queue and affect its execution. Since it has the lowest priority, $T_{4,1}$ waits in the ready queue until time 7 before execution in both schedules.

From this example, we can see that, although the lower priorities of ALAP tasks help delay their executions under PPA, the decreased promotion times (due to increased response times) for such tasks reduce their opportunities to take advantage of the delay queue in POPPA. Moreover, by comparing the PORMS and POPPA schedules, it is hard to say which one performs better in terms of fulfilling the tasks' execution preferences. For instance, although the first instance of task T_3 executes one time unit late in POPPA, its second instance was much delayed to time 8 in PORMS. We have quantitatively evaluated the performance of these schedulers on fulfilling the execution preferences of tasks through extensive simulations and the results will be discussed in Section 7.

6. Runtime techniques for preference-oriented execution

It is well-known that real-time tasks typically take a small fraction of their worst-case execution times (WCETs) [9] and significant amount of slack time can be expected at runtime. Such slack time could be exploited to further delay (expedite) the executions of ALAP (ASAP) tasks, respectively. However, from Algorithm 3, we can see that the execution of ALAP tasks can only be delayed in the delay queue until their promotion times. Once such tasks are promoted to the ready queue, the POFP scheduler treats them in the same way as ASAP tasks and no further delay will be imposed on their executions. Before presenting the runtime techniques to further improve the preference-oriented executions of tasks based

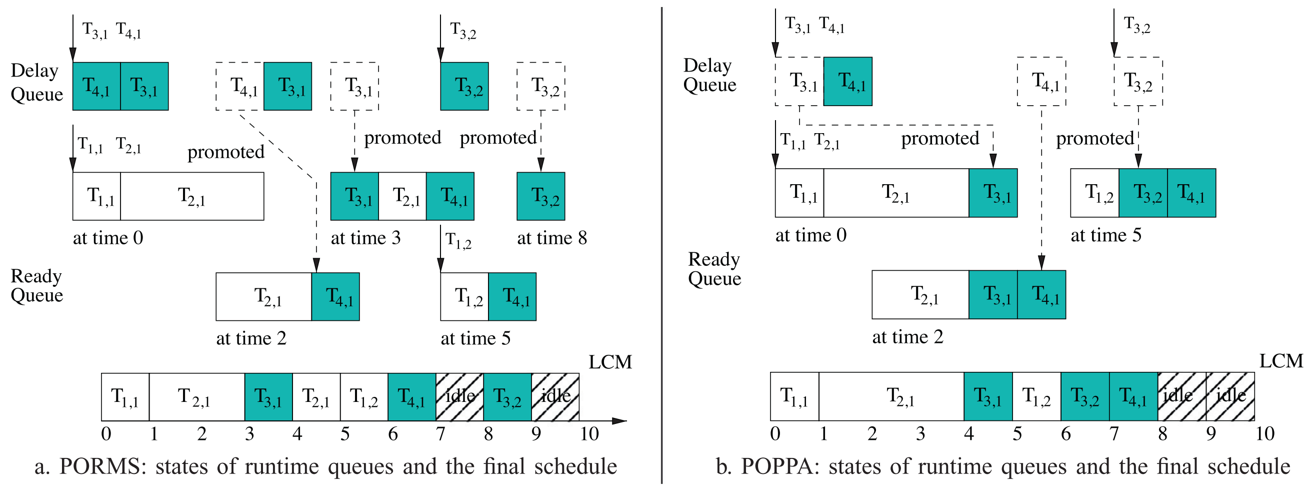


Fig. 2. The PORMS and POPPA schedules for the example task set in Section 4.2.

on slack management, in what follows, we first illustrate the idea through an example.

6.1. An example for runtime techniques

To illustrate different aspects with runtime slack, we consider another example task set with four tasks: $T_1(1.5,\ 4)$, $T_2(1,\ 8)$, $T_3(1.5,\ 4)$ and $T_4(1,\ 12)$. Here, tasks T_1 and T_2 have ASAP preference while T_3 and T_4 have ALAP preference (i.e., $\Psi_S = \{T_1, T_2\}$ and $\Psi_L = \{T_3, T_4\}$). The PORMS scheduler is considered with RMS priorities of tasks as $\eta_1 > \eta_3 > \eta_2 > \eta_4$. From Eq. (3), the promotion times of ALAP tasks T_3 and T_4 can be found as $\gamma_3 = 1$ and $\gamma_4 = 4$, respectively. To have runtime slack times, it is assumed that the actual execution times of tasks are: $a_1 = 0.5,\ a_2 = 1,\ a_3 = 1$ and $a_4 = 0.5$.

When the actual execution times are less than the WCETs, tasks generate dynamic slack time at runtime. In this example, each instance of task T_1 produces 1 unit of slack time while each instance of tasks T_3 and T_4 yields 0.5 unit of slack time. If no special consideration is taken for these slack times, following the steps in Algorithm 3, the states of runtime queues and the PORMS schedule for the first few instances of the tasks can be found as those shown in Fig. 3a.

When $T_{3,\ 1}$ is promoted at time 1, it preempts the execution of $T_{2,\ 1}$ since it has higher priority (i.e., $\eta_3 > \eta_2$). Similarly, when $T_{4,\ 1}$ is promoted at time 4, it is executed right after the early completion of $T_{1,\ 2}$ since it is the only ready task in the ready queue.

Next, we show that how the executions of tasks can be affected when the slack times are *explicitly* managed at runtime. When the task instance $T_{1,\ 1}$ completes its execution early at time 0.5, its generates one unit of slack time S_1, which is kept in a separate slack queue as shown in Fig. 3b. Here, the slack inherits the priority of its generating task and will compete the processor with other ready tasks.

That is, at time 0.5, S_1 will be picked for execution since it has higher priority than the ready task instance $T_{2,\ 1}$. However, since S_1 is not a *real* task, it actually *wraps* the execution of the available ASAP task instance $T_{2,\ 1}$ in the ready queue with the priority of S_1 during its allocated time. From a different perspective, this can be viewed as S_1 *lending* its allocated time to $T_{2,\ 1}$. However, since such wrapped execution is performed with S_1's priority (i.e., T_1's priority), when the ALAP task instance $T_{3,\ 1}$ is promoted at time 1, it cannot preempt the wrapped execution of $T_{2,\ 1}$ as the priority of $T_{3,\ 1}$ is lower than that of S_1.

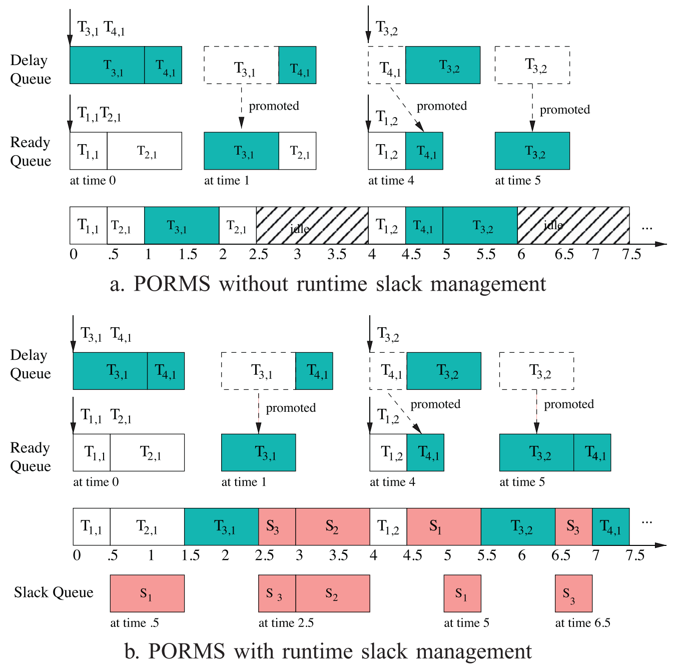


Fig. 3. Impact of runtime slack management in PORMS for the example in Section 6.1.

When $T_{2,\ 1}$ finishes with the wrapped execution at time 1.5, it has to return the borrowed time as a *new* slack S_2, which has the size of one unit and inherits $T_{2,\ 1}$'s priority. After that, $T_{3,\ 1}$ is executed as it has higher priority than S_2. When $T_{3,\ 1}$ completes its execution early at time 2.5, another new piece of slack S_3 is generated that has the size of 0.5 unit with $T_{3,\ 1}$'s priority. The state of the slack queue at time 2.5 is also shown in the figure where slacks are ordered in their priorities.

Since the ready queue is empty, S_3 and S_2 occupy the processor in the order of their priorities and let it idle during their allocated times. At the meantime, the ALAP task instance $T_{4,\ 1}$ is held in the delay queue until time 4. At that time, $T_{4,\ 1}$ is promoted to the ready queue and a higher priority ASAP task instance $T_{1,\ 2}$ arrives. In addition, the ALAP task instance $T_{3,\ 2}$ also arrives and enters the delay queue,

When $T_{1,\ 2}$ completes early at time 4.5, it generates another piece of slack S_1, which inherits $T_{1,\ 2}$'s priority. When S_1 takes the processor at time 4.5, it lets the processor idle even though the ready queue has an ALAP task instance $T_{4,1}$. The reason is that, as an ALAP task instance, $T_{4,1}$ should be executed at a later time when possible. Moreover, idling the processor in view of S_1 will not cause $T_{4,1}$ to miss its deadline as this would be the time it would have to wait if $T_{1,2}$ had taken its WCET.

Similarly, when the ALAP task instance $T_{3,2}$ is promoted at time 5, S_1 continues idling the processor as its has higher priority. Once S_1 uses up its time, $T_{3,2}$ starts its execution at time 5.5. Again, $T_{3,2}$'s early completion generates another piece of slack S_3, which further delays the execution of $T_{4,1}$ until time 7 (actually, it is the time $T_{4,1}$ would start its execution should all other task instances take their WCETs).

6.2. Slack management with wrapper-tasks

From the above example, we can see that, through judicious slack management, ASAP tasks can borrow high priority slack time and be executed at their earliest possible times. Moreover, the executions of ALAP tasks can be dramatically delayed by such high priority slack even after they are promoted to the ready queue. To generalize the above idea and enable runtime slack to compete for the processor, we extend the *wrapper-task* based slack management, which has been studied for dynamic priority based task systems [18], to the fixed-priority setting.

Basically, each piece of slack time will be represented by a *wrapper-task* with two parameters (c_i, η_i). Here, c_i denotes the size of the slack and η_i represents the slack's priority, which is inherited from the task whose early completion gives rise to this slack. At runtime, wrapper-tasks are kept in a separate *slack queue* $\mathcal{Q}_S$ and compete for the processor with tasks in the ready queue. At the dispatch time of the POFP scheduler, there are four possibilities regarding the states of the ready queue $\mathcal{Q}_R$ and the slack queue $\mathcal{Q}_S$.

If both queues are empty, the POFP scheduler will let the processor idle while waiting for the new arrival of ASAP tasks and/or the promotion of ALAP tasks from the delay queue. Otherwise, suppose that T_k and S_h are the highest priority task and wrapper-task in $\mathcal{Q}_R$ and $\mathcal{Q}_S$, respectively.

For the cases where $\eta_k > \eta_h$ (the ready task T_k has higher priority) **OR** the slack queue is empty (i.e., $S_h = NULL$), the POFP scheduler can dispatch task T_k normally from the ready queue $\mathcal{Q}_R$. However, for the cases of $\eta_h > \eta_k$ (i.e., the slack S_h has higher priority), if the ready queue is empty (i.e., $T_k = NULL$) **OR** all tasks in the ready queue $\mathcal{Q}_R$ are ALAP tasks, the slack (represented by the wrapper task S_h) will get the processor and keep it idle for the interval of its allocated time. This effectively delays the executions of the ALAP tasks (if any) that have been promoted to the ready queue $\mathcal{Q}_R$.

The most interesting case occurs when the slack has higher priority (i.e., $\eta_h > \eta_k$) and the ready queue $\mathcal{Q}_R$ contains at least one ASAP task. Suppose that the highest priority ASAP task in $\mathcal{Q}_R$ is T_s (and it is possible that $\eta_s < \eta_k$). In this case, the slack (i.e., the wrapper task S_h) obtains the processor and will *lend* its time to T_s by *wrapping* its execution. That is, during the wrapped execution of T_s, T_s inherits the higher priority of S_h, which can prevent preemptions from future promoted ALAP tasks as shown in the above example.

Note that, once such wrapped execution ends due to the completion of T_s or S_h using up its slack time, a new piece slack with the size of the wrapped execution and T_s's priority will be generated and inserted back to the slack queue $\mathcal{Q}_S$. The operations of slack (i.e., wrapper tasks) are similar to those for the dynamic priority based scheme and interested readers can find more detailed discussions in [18]. However, we would like to point out that, the wrapper-task based slack management is a generic technique, which can be applied to any classical fixed-priority scheduler (e.g., RMS) as well.

6.3. Dummy task to exploit spare capacity

For a given set of tasks that are schedulable under fixed-priority scheduling, it is more likely that the system is not fully utilized (i.e., $U < 1$). However, the wrapper-task technique discussed in the last section is designed to handle dynamic slack generated from the early completion of tasks, which cannot directly utilize such spare capacity. In [18], we utilized a *dummy task* T_0 to represent system spare capacity, which can periodically introduce slack time into the system at runtime.

Following the same idea, we can also augment a given task set with a dummy task $T_0(c_0, p_0, \eta_0)$. From [18], we know that the timing parameters of T_0 have a direct impact on system performance by controlling how slack from the spare capacity is introduced to the system. For instance, T_0's period p_0 determines how often the slack is introduced at runtime. In fixed-priority scheduling, T_0's priority η_0 also plays a very important role. From the above discussions, we know that slack time needs to have a higher priority to wrap an ASAP task for its early execution as well as to delay the execution of ALAP tasks. Therefore, it is desirable to have a higher priority for the dummy task T_0.

However, on the other hand, the choice of T_0's timing parameters and priority should not compromise the schedulability of the augmented task set. Considering the much more complex interplay between tasks' schedulability and their priorities and timing parameters, selecting the appropriate (c_0, p_0, η_0) for T_0 becomes more challenging than the case for the dynamic-priority based scheduling [18].

A simple utilization based dummy task: In this work, we consider a simple but conservative utilization-based approach to determine T_0's timing parameters and priority. From [15], we know that, a given task set Ψ is schedulable under RMS if the system utilization of the task set satisfies: $U \leq U^{bound}(n) = n(2^{1/n} - 1)$. Here, n is the number of tasks in a task set and $U^{bound}(n)$ is the utilization bound to ensure the task set's schedulability under the RMS scheduler.

In this work, we set $u_0 = U^{bound}(n) - U$, $p_0 = \min\{p_i | T_i \in \Psi\}$ and $c_0 = u_0 \cdot p_0$. According to the results in [14], since the dummy task has the same period as the task with the smallest period, the addition of the dummy task will not compromise the schedulability of the augmented task set under RMS. Moreover, by having the smallest period, T_0 will have the highest priority η_0 in RMS. In addition, we assume that the dummy task T_0 will have the ASAP execution preference, which enables it act as slack time in the ready queue at the earliest possible time.

7. Evaluations and discussions

The performance of the proposed PPA algorithm, the POFP scheduler and the runtime techniques, in terms of on how well the execution preferences of tasks are fulfilled, have been evaluated through extensive simulations. To this end, we developed a discrete event simulator using C++ and implemented the work-conserving fixed-priority (FP) scheduler as well as the non-work-conserving POFP scheduler.

Moreover, we consider both the classical RMS and the proposed PPA priority assignments for the tasks. Combining the priority assignments with the two different fixed-priority schedulers, we evaluated four different schemes:

- **RMS**: which represents the classical RMS scheduler [15] and is used as the baseline in our evaluations;
- **PPA**, standing for the work-conserving FP scheduler with the proposed PPA priority assignment;

- **PORMS**, which denotes the proposed POFP scheduler with RMS priority assignment; and
- **POPPA**, which stands for the proposed POFP scheduler with our new PPA priority assignment.

7.1. Evaluation metrics and simulation settings

Due to the preemptive nature of the fixed-priority schedulers under consideration, it is not straightforward to quantify how well the execution preferences of tasks are fulfilled. Here, we use the *preference value (PV)* of tasks, a simple metric proposed in our previous work [10], which is defined over the completion and start times of ASAP and ALAP tasks. Specifically, the preference value for a task instance $T_{i,j}$ is defined as [10]:

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

where $st_{i,j}$ and $ft_{i,j}$ denote the *actual* start and complete times, respectively, of the task instance $T_{i,j}$ during a specific execution under a given scheduler. Moreover, ft_i^{min} and ft_i^{max} represent the *ideal* earliest and latest completion times, respectively, if T_i is an ASAP task. Similarly, st_i^{min} and st_i^{max} stand for the *ideal* earliest and latest start times, respectively, if T_i is an ALAP task.

Note that, due to preemptions and interference among tasks, it would be very difficult (if not impossible) to find the earliest/latest start/completion times for each individual task instance. In this work, we use the *ideal* values for those time points by assuming that there is only one task T_i in the system. Specifically, suppose that $T_{i,j}$ arrives at time $r_{i,j}$. $T_{i,j}$ could start its execution as early as $st_i^{min} = r_{i,j}$, while the latest time it has to start its execution to avoid a deadline miss would be $st_i^{max} = (r_{i,j} + p_i) - c_i$, where c_i and p_i are the WCET and period of task T_i, respectively. Similarly, assume that the actual execution time of $T_{i,j}$ is $a_{i,j}$, we can find its *ideal* earliest and latest finish times are $ft_i^{min} = r_{i,j} + a_{i,j}$ and $ft_i^{max} = r_{i,j} + p_i - c_i + a_{i,j}$, respectively.

From these definitions, we can see that the value of $PV_{i,j}$ has the range of [0, 1], where a larger value indicates that $T_{i,j}$'s execution preference has been fulfilled better. For a specific running of a task set under a given scheduler, a task's preference value is defined as the average PV of all its instances. Moreover, the overall PV of a task set is the average preference value of all its tasks.

Task generation: We consider synthetic tasks that are generated as follows: for a given number of tasks n and system utilization U, the utilization of each task is generated using the *UUniFast* scheme proposed in [5]. Then, the period of each task is uniformly distributed within the range of [10, 100] and its WCET (Worst-Case Execution Time) is set accordingly. A certain number k of these n tasks are randomly chosen to have ASAP execution preference while the remaining are considered as ALAP tasks.

We vary the system utilization U (from 0.1 to 0.8), the number of tasks in a task set n (from 10 to 100), and the number of ASAP tasks k ($0.2 \cdot n$, $0.5 \cdot n$ and $0.8 \cdot n$) and evaluate their impacts on the performance of the proposed schedulers and techniques. In the figures below, each data point corresponds to the average result of 100 schedulable task sets (where task sets that are not schedulable under RMS, especially for high system utilizations, are discarded in our simulations). Tasks are assumed to take their WCETs at run-time unless otherwise specified.

7.2. Effects of system utilizations

The effects of system utilization on the achieved PVs for different types of tasks under the four scheduling schemes (i.e., RMS, PPA, PORMS, and POPPA) are first shown in Fig. 4. Here, we consider task sets with $n = 10$ tasks. The number of ASAP tasks in each task set is indicated with the value of k. Three cases with $k = 2$, $k = 5$ and $k = 8$, respectively, are considered to represent different workload mixtures of ASAP and ALAP tasks. The results for these three cases under different scheduling schemes are denoted accordingly.

Focusing on only ASAP tasks in the task sets, Fig. 4a shows their achieved average PVs. Recall that the PV for each task has a upper bound of 1, which may not be *simultaneously* achievable for all tasks by any scheduler when there are more than one tasks in a task set. Here, we can see that, for the cases with low system utilizations (e.g., $U \leq 30\%$), the resultant PVs for ASAP tasks under different schemes are very close where all values are larger than 0.97. The reason is that, at low system utilizations, almost all ASAP tasks can be executed right after their arrivals under different scheduling schemes, regardless of the number of ASAP tasks in a task set.

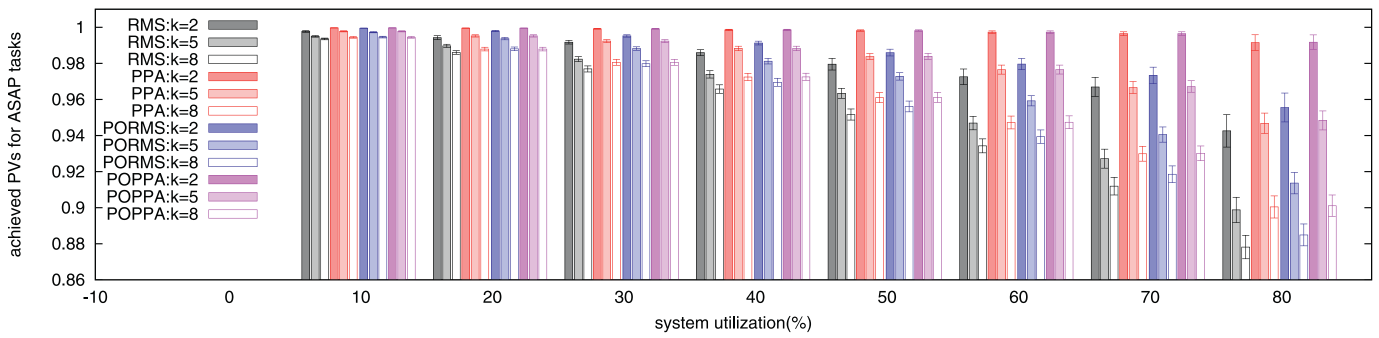
However, as system utilization increases where the size of each task becomes larger, the differences between the four scheduling schemes for different workload mixtures become more pronounced. In particular, when there are more ASAP tasks in a task set, it becomes more difficult to execute them right after their arrivals under all scheduling schemes, which leads to reduced PVs for such tasks. Moreover, for all the settings, by having higher priorities for ASAP tasks (where ALAP tasks take lower priorities), PPA can execute ASAP tasks at earlier times with larger achieved PVs when compared to that of RMS, especially at higher system utilizations.

With the help of the dual-queue technique, PORMS performs slightly better than RMS as it provides more opportunities to execute ASAP tasks at earlier times by holding (possibly high priority) ALAP tasks in the delay queue. However, the differences in the achieved PVs for ASAP tasks between PPA and POPPA are almost negligible for all the settings. The reason is that, the higher priorities of ASAP tasks under PPA already enable them to execute at earlier times. Therefore, it becomes extremely difficult for ASAP tasks to explore the delayed executions of ALAP tasks under POPPA and to further improve their early executions.

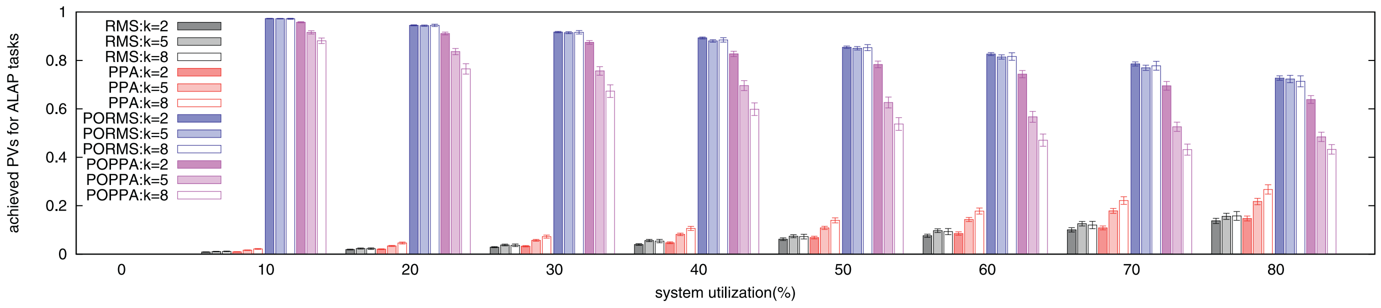
Next, we focus on the achieved PVs for ALAP tasks, where the results are shown in Fig. 4b. Compared to those for ASAP tasks, the achieved PVs for ALAP tasks have much larger variations. Note that, the execution preference of ALAP tasks conflicts directly with the design principle of the work-conserving fixed-priority scheduler. In particular, the classical RMS scheduler does not have any special consideration for ALAP tasks, which results in very low PVs for ALAP tasks (i.e., poor fulfillment of ALAP tasks' execution preference), especially for the cases with low system utilizations. As system utilization increases, the resultant PVs get slightly higher under RMS where the executions of ALAP tasks start relatively late due to increased system loads. However, even for the case of $U = 80\%$, the values are still less than 0.2.

By assigning lower priorities to ALAP tasks, PPA can perform slightly better than RMS in all the settings under consideration. However, due to the work-conserving nature of its underlying fixed-priority scheduler, ALAP tasks are still executed quite early, which leads to small PVs for such tasks.

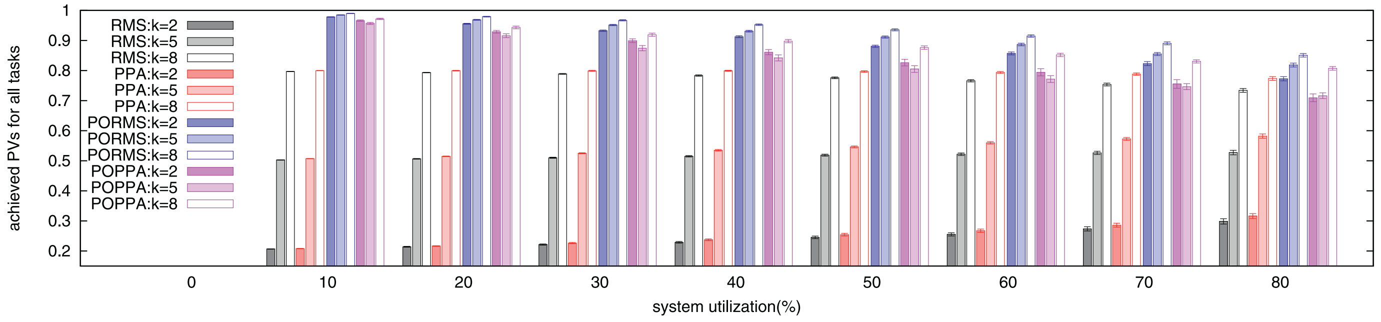
With the help of the dual-queue technique, our proposed non-work-conserving POFP scheduler delays the executions of ALAP tasks (at least) until their promotion times. Such delays lead to dramatically increased PVs for ALAP tasks under both PORMS and POPPA, when compared to their corresponding counterparts, RMS and PPA, respectively. Therefore, the dual-queue technique adopted in the POFP scheduler plays a dominant rule in delaying the executions of ALAP tasks, when compared to that of the PPA priorities.



a. Achieved average PVs for ASAP tasks



b. Achieved average PVs for ALAP tasks



c. Achieved average PVs for all tasks

Fig. 4. Achieved PV vs. system utilization U under different schedulers for task sets with $n = 10$ tasks.

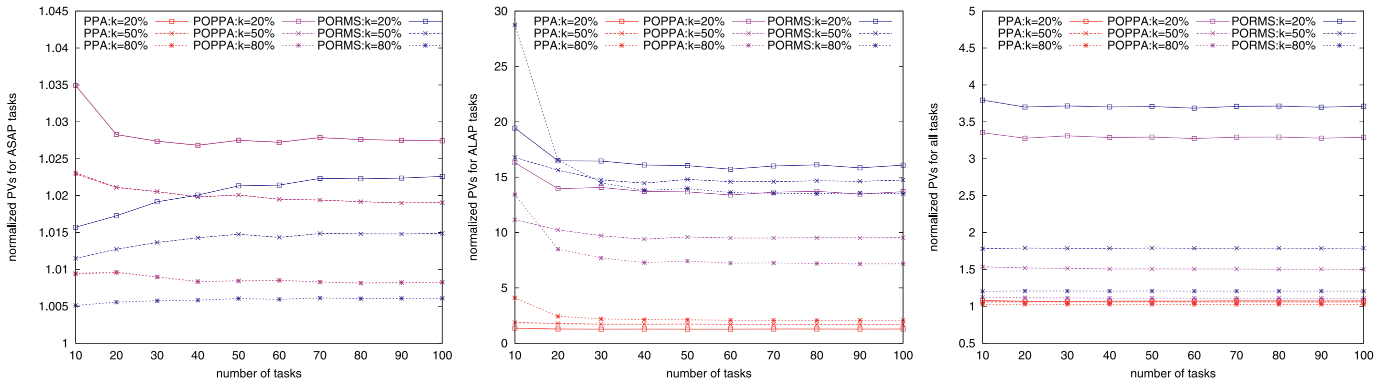
For different workload mixtures (i.e., different values of k), PORMS performs roughly the same for a given system utilization. This comes from the fact that, with the randomly chosen execution preferences for tasks, the priorities of ALAP tasks scatter quite evenly within the priority spectrum regardless of the number of ALAP tasks in a task set. Therefore, the average promotion times of ALAP tasks will be similar, which results in roughly the same PVs for ALAP tasks.

However, it is interesting to see that higher PVs can be achieved for ALAP tasks under PORMS than POPPA, especially for cases with higher system utilizations. The reason is that, the lower priorities of ALAP tasks in POPPA lead to reduced promotion times for such tasks (due to their increased response times). Hence, ALAP tasks can be held in the delay queue for less time before they are forced to move to the ready queue, which potentially leads to earlier start times and thus reduced PVs for them. Moreover, when there are fewer number of ALAP tasks (i.e., larger values of k), it is more likely that PPA will assign the lowest few priorities to them and such effects become more prominent.

Fig. 4c further shows the overall achieved PVs for all tasks (including both ASAP and ALAP tasks) in a task set. Note that, the underlying work-conserving fixed-priority scheduler for RMS and PPA performs well only for ASAP tasks. Thus, the overall achieved PVs under RMS and PPA depend heavily on the number of ASAP tasks, where larger overall PVs are achieved with more ASAP tasks. The benefit of having PPA priorities on improving the fulfillment of tasks' execution preferences is very marginal where the overall PVs are only slightly larger compared to those of RMS.

Both PORMS and POPPA have quite stable performance on the overall PVs of tasks, which vary only slightly for different workload mixtures for a given system utilization. As system utilization increases, both PORMS and POPPA performs slightly worse since it becomes more difficult to satisfy the execution preferences of all tasks. Again, PORMS performs slightly better than POPPA due to decreased promotion times for ALAP tasks in POPPA, which turns to be a dominant factor for fulfilling the execution preferences of all tasks in the underlying POFP scheduler.

Fig. 4 also shows the 95% confidence intervals of the achieved PVs, which have a rather small range for all the settings. This indicates that the reported average results for the PVs of tasks are quite reliable statistically.



a. normalized PVs for ASAP tasks b. normalized PVs for ALAP tasks c. normalized PVs for all tasks

Fig. 5. Effects of task number n on the normalized PVs (compared to RMS; $U = 50\%$).

7.3. Effects of number of tasks

When the system utilization is $U = 50\%$, we evaluate the effects of n (i.e., the number of tasks in a task set) on the resultant PVs of tasks. The number of tasks n varies from 10 to 100. As before, three cases are considered to represent different workload mixtures, where the number of ASAP tasks k in a task set are $k = 0.2 \cdot n$, $k = 0.5 \cdot n$ and $k = 0.8 \cdot n$, respectively. For different scheduling schemes, these three cases are denoted as 20%, 50% and 80% accordingly. The PVs obtained under RMS are considered as the baseline, and the PVs of the schemes are indicated as normalized with respect to those baseline values (Fig. 5).

First we consider ASAP tasks only and show their normalized PVs with varying number of tasks in Fig. 5a. Here, we can see that, the normalized PVs for ASAP tasks are quite stable (close to 1 with very small variations) for different number of tasks under different scheduling schemes. For the case of $n = 10$, the results are in line with the previously reported PVs for ASAP tasks with $U = 50\%$. In general, when there are more ASAP tasks, it becomes more difficult to complete all ASAP tasks early under all schemes, which leads to slightly smaller normalized PVs for ASAP tasks.

Fig. 5b shows the normalized PVs for ALAP tasks. Recall that RMS performs very poorly for ALAP tasks and PVs for such tasks are very small (less than 0.2). Hence, we can see that the normalized PVs of ALAP tasks have quite large variations for different schemes, especially for the case of $n = 10$. However, as the number of tasks increases, the performance variation of the scheduling scheme becomes more stable with smaller sizes of tasks. Here, the achieved PVs for ALAP tasks under PPA are almost twice as that under RMS with the normalized PVs being close to 2 (when $n \geq 20$). For PORMS, the normalized PVs for ALAP tasks can be as high as 17, which indicates significant improvement for ALAP tasks to fulfill their execution preference. The same as before, POPPA performs slightly worse than PORMS with smaller normalized PVs, which can be as high as 14 when there are more ALAP tasks in the task sets (i.e., $k = 20\%$).

The normalized overall PVs for all the tasks, as a function of the number of tasks are further shown in Fig. 5c. Again, as task number varies, the different scheduling schemes perform quite stable with very little variations in the normalized overall PVs. Moreover, with its preference-aware priority assignment, PPA can perform slightly better than RMS, where its normalized PVs are marginally larger than 1. When there are more ALAP tasks in the task sets, the normalized PVs under PORMS and POPPA are close to 3.5 since the underlying POFP scheduler can effectively delay the executions of ALAP tasks. However, when there are more ASAP tasks, the normalized PVs for PORMS and POPPA reduce quickly since all schemes achieve similar PVs for ASAP tasks.

7.4. Effects of runtime techniques

From the above discussions, we can see that PPA can only moderately improve the fulfillment of tasks' execution preferences over RMS. Therefore, in the evaluations of runtime techniques, we consider only the non-work-conserving schedulers PORMS and POPPA. In addition, for the number of tasks, we consider the case of $n = 10$ with each task set having $k = 5$ ASAP tasks (i.e., the balanced workload mixtures).

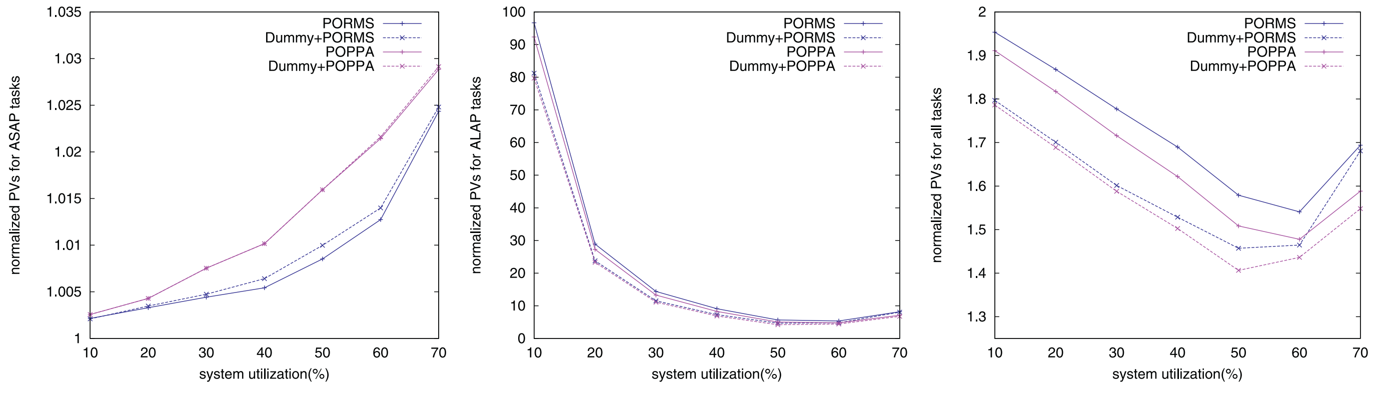
Effects of dummy task: With varying system utilization, the effects of dummy task on the normalized PVs of tasks are shown in Fig. 6. Since we focus on exploiting static spare capacity using dummy task, we assume that all tasks take their WCETs at runtime and all slack times are introduced by the dummy task. Moreover, for simplicity, the utilization of the dummy task is set as $u_0 = \ln 2 - U$, where $\ln 2$ is the asymptotic utilization bound for task sets to be schedulable under fixed-priority scheduling [15].

First, the normalized PVs for ASAP tasks can be seen in Fig. 6a. Here, the normalized PVs are close to 1 with quite small variations. The reason is that, both PORMS and POPPA can perform very well for ASAP tasks especially for low system utilizations, which is consistent with our previous results. As system utilization increases and tasks become larger, PORMS and POPPA can perform slightly better than RMS, which leads to marginally increased PVs for ASAP tasks. However, for both PORMS and POPPA, the additional improvement of utilizing the dummy task is almost negligible.

From Fig. 6b shows the normalized PVs for ALAP tasks, which again have quite large variations. By delaying ALAP tasks with the dual-queue technique, both PORMS and POPPA can perform significantly better for such tasks than RMS (close to two magnitudes for the case of $U = 10\%$). As system utilization increases, the size of tasks become larger and the promotion times of ALAP tasks can decrease quickly, which leads to much smaller normalized PVs for such tasks.

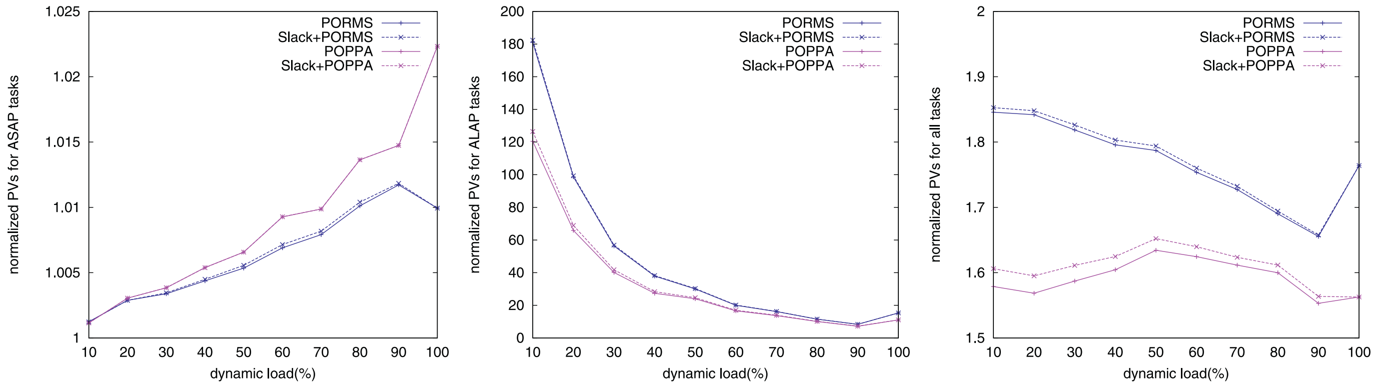
Moreover, it is interesting to see that, although the dummy task has been introduced with the objective of further delaying the execution of ALAP tasks, both PORMS and POPPA perform worse when the dummy task is utilized. The reason is that, the dummy task has the highest priority in RMS with its the smallest period. Moreover, with its ASAP execution preference, it is very likely that PPA also assigns it the highest priority. Hence, the reduced promotion times of ALAP tasks due to the highest priority dummy task in the augmented task set can overshadow the benefits of the introduced slack time.

The normalized overall PVs for all tasks are shown in Fig. 6c, which further confirms that the dummy task can lead to negative effects on the achieved PVs of tasks. When we consider all the



a. normalized PVs for ASAP tasks b. normalized PVs for ALAP tasks c. normalized PVs for all tasks

Fig. 6. Effects of the dummy task on normalized PVs (compared to RMS; $n = 10$ and $k = 5$).



a. normalized PVs for ASAP tasks b. normalized PVs for ALAP tasks c. normalized PVs for all tasks

Fig. 7. Effects of dynamic slack on normalized PVs (compared to RMS; $n = 10, k = 5$ and $U = 50\%$).

tasks, the PVs of tasks can be improved by more than 50% under both PORMS and POPPA over RMS.

Effects of dynamic slack: The effects of dynamic slack are also evaluated and the results are shown in Fig. 7. Here, we consider the case of $U = 50\%$ (without dummy task). We vary the dynamic load α of tasks from 10% to 100%. Smaller values of α indicate more dynamic slack can be expected from the early completion of tasks and $\alpha = 100\%$ represents the case where no dynamic slack is available at runtime. For a given value of α, the actual execution times for the instances of task T_i are randomly generated within a range to have the average value as $\alpha \cdot c_i$.

Similar patterns of the normalized PVs for tasks can be observed. First, most ASAP tasks can complete at their earliest times under all scheduling schemes, especially for the low dynamic loads where tasks only take a small fraction of their WCETs. This leads to the normalized PVs for ASAP tasks being close to 1 as can be seen in Fig. 7a. Although PORMS and POPPA perform better than RMS as the dynamic load increases, the improvement is very marginal (less than 1.5% even at $\alpha = 100\%$). Moreover, the advantage of managing such dynamic slack for ASAP tasks is hardly noticeable, especially for POPPA.

The normalized PVs of ALAP tasks can be as high as 180 when the dynamic load is low (e.g., $\alpha = 10\%$) due to extremely poor performance of RMS for completing such tasks at earlier times (see Fig. 7b). When tasks have longer executions at high dynamic loads, the advantages of PORMS and POPPA over RMS quickly diminish. Moreover, although managing dynamic slack can improve the PVs of ALAP tasks under POPPA, such improvements decrease quickly as dynamic load increases (with less slack time). The effects of dynamic slack on PORMS are hardly noticeable.

Fig. 7c shows the normalized overall PVs for all the tasks, and the results are consistent with previous evaluations. The additional improvement from dynamic slack is rather marginal. Note that, the PV of an ASAP (ALAP) task instance is defined on a single finish (start) time point. With the complex interference among tasks' executions, we can see that the overall PVs of tasks do not change monotonically as dynamic load increases.

8. Conclusions

In this work, for periodic real-time tasks where some tasks are preferably executed ASAP while others ALAP, we investigated various techniques for fixed-priority scheduling. First, as a variation of Audsley's optimal priority assignment, we studied a *preference priority assignment (PPA)* algorithm that favors ALAP tasks for lower priorities. Then, a non-work-conserving *preference-oriented fixed-priority (POFP)* scheduling algorithm is proposed that exploits the dual-queue technique to address the late execution requirements of ALAP tasks. Runtime techniques based on slack management with dummy and wrapper tasks are also investigated with the objective of further improving tasks' execution preferences.

The proposed techniques and schemes are evaluated through extensive simulations. The results show that, although both PPA and the dual-queue POFP scheduler are quite effective, the non-work-conserving POFP scheduler plays a dominant role in addressing tasks' execution preferences. In particular, for ALAP tasks, the performance can be improved up to two magnitudes when compared to the classical RMS scheduler, which is preference-oblivious. The wrapper-task based runtime technique can slightly improve tasks' performance, while the dummy task can have negative impacts due to reduced promotion times of ALAP tasks.

In our future work, we will investigate better evaluation metrics that can incorporate all execution segments of task instances for perference-oriented executions. Moreover, we will study preference-oriented scheduling algorithms for multiprocessor real-time systems.

Acknowledgements

The authors would like to thank the reviewers for their valuable suggestions that helped to improve the paper. This work is supported in part by the National Science Foundation through the NSF CAREER Award CNS-0953005, and the awards CNS-1422709 and CNS-1421855. Any opinions, findings, and conclusions as well as recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] N. Audsley, A. Burns, M. Richardson, K. Tindell, A.J. Wellings, Applying new scheduling theory to static priority pre-emptive scheduling, Softw. Eng. J. 8 (5) (Sep 1993) 284–292.
[2] N. Audsley, K. Tindell, A. Burns, The end of line for static cyclic scheduling? in: Real-Time Systems, 1993. Proceedings., Fifth Euromicro Workshop on, Jun. 1993, pp. 36–41.
[3] N.C. Audsley, On priority assignment in fixed priority scheduling, Inf. Process. Lett. 79 (1) (2001) 39–44.
[4] R. Begam, D. Zhu, H. Aydin, Preference-oriented fixed-priority scheduling for real-time systems, in: Proceedings of the 12th IEEE International Conference on Embedded Computing (EmbeddedCom; co-located with DASC), Aug. 2014.
[5] E. Bini, G.C. Buttazzo, Biasing effects in schedulability measures, in: Proceedings of the Euromicro Conference on Real-Time Systems, 2004.
[6] H. Chetto, M. Chetto, Some results of the earliest deadline scheduling algorithm, IEEE Trans. Softw. Eng. 15 (1989) 1261–1269.
[7] R. Davis, A. Wellings, Dual priority scheduling, in: Proceedings of the IEEE Real-Time Systems Symposium, 1995, pp. 100–109.
[8] R.I. Davis, L. Cucu-Grosjean, M. Bertogna, A. Burns, A review of priority assignment in real-time systems, J. Syst. Arch. (2016). Pages –
[9] R. Ernst, W. Ye, Embedded program timing analysis based on path clustering and architecture classification, in: Proceedings of The International Conference on Computer-Aided Design, Nov. 1997, pp. 598–604.
[10] Y. Guo, H. Su, D. Zhu, H. Aydin, Preference-oriented real-time scheduling and its application in fault-tolerant systems, J. Syst. Arch. 61 (2) (2015) 127–139.
[11] 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), 2013.
[12] 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 International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Aug. 2013.
[13] M. Joseph, P.K. Pandya, Finding response times in a real-time system, BCS Comput. J. 29 (5) (1986) 390–395.
[14] T.-W. Kuo, A.K. Mok, Load adjustment in adaptive real-time systems, in: Proceedings of The IEEE Real-Time Systems Symposium, Dec. 1991, pp. 160–170.
[15] C.L. Liu, J. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, J. ACM 20 (January 1973) 46–61.
[16] H. Su, D. Zhu, S. Brandt, An Elastic Mixed-Criticality Task Model and Early-Release EDF Scheduling Algorithms, Tech. rep. cs-tr-2016-002, University of Texas at San Antonio, February 2016.
[17] O.S. Unsal, I. Koren, C.M. Krishna, Towards energy-aware software-based fault tolerance in real-time systems, in: Proceedings of the International Symposium on Low Power Electronics and Design, 2002, pp. 124–129.
[18] D. Zhu, H. Aydin, Reliability-aware energy management for periodic real-time tasks, IEEE Trans. Comput. 58 (10) (2009) 1382–1397.

Rehana Begam is currently a PhD candidate in the Department of Computer Science at The University of Texas at San Antonio (U.S.A) and she joined the program in 2012. Her research interests are in Heterogeneous Real-Time Systems and Cloud Computing. She has worked on resource allocation and scheduling algorithms for such systems.

Qin Xia received the PhD degree in Computer Science from Xi'an Jiaotong University (China) in 2012. She is currently a Lecture in the School of Electronic and Information Engineering at Xi'an Jiaotong University. She is a visiting scholar at The University of Texas at San Antonio from 2015 to 2016. Her research interests cover real-time systems and network security.

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 interests include real-time systems, power aware computing and fault-tolerant systems. He has served on program committees (PCs) for several major IEEE and ACM-sponsored real-time conferences (e.g., RTSS and RTAS). 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.