Preference-Oriented Fixed-Priority Scheduling for Real-Time Systems

Rehana Begam, Dakai Zhu
University of Texas at San Antonio
San Antonio, TX, 78249
rehan.sheta@gmail.com, dakai.zhu@utsa.edu

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

***Abstract*—Most real-time scheduling algorithms prioritize tasks solely based on their timing parameters and cannot effectively handle them when they have different *execution preferences*. In this paper, for a set of periodic tasks, where some tasks are preferably executed *as soon as possible (ASAP)* and others *as late as possible (ALAP)*, we investigate preference-oriented fixed-priority scheduling algorithms. Specifically, following the idea in dual-priority scheduling, we derive *promotion times* for ALAP tasks (only). Then, we devise a dual-queue based fixed-priority scheduling algorithm that retains ALAP tasks in the *waiting queue* until their promotion times to delay their executions while putting ASAP tasks into the *ready queue* immediately once they arrive for early execution. We also investigate online techniques to further expedite (delay) the executions of ASAP (ALAP) tasks, respectively. Our evaluation results show that the dual-queue technique with ALAP tasks' promotion times can effectively address the execution preferences of both ASAP and ALAP tasks, which can be further improved at runtime with wrapper-task based slack management. Our technique is shown to yield clear advantages over a simple technique that periodically inserts idle intervals to the schedule before ALAP tasks are executed.**

***Index Terms*—Real-Time Systems, Fixed-Priority Scheduling, Preference-Oriented Execution**

I. Introduction

In the past, numerous real-time scheduling algorithms have been proposed by the research community (such as *rate-monotonic-scheduling (RMS)* and *earliest-deadline-first (EDF)*, which are optimal schedulers based on static and dynamic priorities, respectively [11]). With exclusive focus on meeting the timing constraints, most existing real-time scheduling algorithms prioritize and schedule tasks solely based on their timing parameters (e.g., deadlines and periods). Moreover, these algorithms normally adopt the *work-conserving* strategy to keep the processor busy as long as there are ready tasks and to execute the workload at the earliest possible time instants.

However, there are occasions when it can be beneficial to execute tasks at their latest times provided that there are no deadline misses. For instance, to get better response time for aperiodic soft real-time tasks, the execution of periodic hard real-time tasks can be delayed maximally [3, 4]. In addition, in fault-tolerant systems, backup tasks should also be executed as late as possible to reduce the overlapped executions with the primary tasks on other processors and thus the system overhead for fault tolerance [8, 9, 12]. Although the *earliest deadline latest (EDL)* [3] and *dual-priority (DP)* schedulers [4] have been proposed to schedule periodic tasks at their latest times, these schedulers treat <u>all</u> periodic tasks *uniformly* without differentiating their execution preferences and thus cannot effectively handle tasks with different execution preferences.

This work was supported in part by NSF award CNS-1016855 and NSF CAREER Award CNS-0953005.

As the first study to systematically address the different execution preferences of periodic real-time tasks, we have proposed the *preference-oriented earliest deadline (POED)* scheduling algorithms [6, 7]. Note that, POED is a dynamic priority based scheduler. To the best of our knowledge, there is no fixed-priority based preference-oriented scheduling algorithm yet, which will be studied in this work.

We consider a set of periodic real-time tasks that have different execution preferences, where some tasks are preferably executed *as soon as possible (ASAP)* while others *as late as possible (ALAP)*. For such tasks, we propose the *preference-oriented fixed-priority (POFP)* scheduling algorithm that explicitly takes the execution preferences of tasks into consideration when making scheduling decisions. Specifically, POFP is a *dual-queue* based scheduler, where the *waiting queue* is used to hold ALAP tasks temporarily and prevent their executions until their *promotion times*, which can be derived from the dual-priority scheduling framework [4]. In contrast, ASAP tasks are put into the *ready queue* immediately once they arrive. In addition, by exploiting the slack time generated at runtime, we also investigate online techniques to further expedite (delay) the executions of ASAP (ALAP) tasks, respectively. For this part, we extend the wrapper-task based slack management as well as the dummy task based technique [13] to the fixed-priority settings.

As an example fixed-priority scheduler, we evaluate the performance of PO-RMS and compare it against RMS in terms of meeting the execution preferences of tasks through extensive simulations. Our evaluation results show that the runtime overhead of PO-RMS scheduler is comparable to that of RMS. However, by explicitly taking the tasks' execution preferences into consideration, PO-RMS can fulfill the preference requirements of tasks much better than RMS with the help of the dual-queue technique and promotion times of ALAP tasks. The performance of PO-RMS can be further improved with the wrapper-task based online slack management. The technique is also shown to yield clear gains over a simple technique (the *dummy task based technique*) that periodically inserts idle intervals to the schedule before ALAP tasks are executed.

The rest of this paper is organized as follows. Section II presents system models and preliminaries. The POFP scheduler is proposed in Section III and the online enhancements are addressed in Section IV. Section V discusses the evaluation results and Section VI concludes the paper.

II. System Models and Preliminaries

A. 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. 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. p_i is also the relative deadline of task T_i. The offset of task T_i is denoted by ϕ_i. That is, the first *task instance* (or *job*) of task T_i arrives at time ϕ_i. The j^{th} task instance of task T_i, which is denoted by $T_{i,j}$, arrives at time $r_{i,j} = \phi_i + (j-1) \cdot p_i$ and has a deadline at time $d_{i,j} = \phi_i + j \cdot p_i$. The task T_i's utilization is defined as $u_i = \frac{c_i}{p_i}$, and the system utilization is further defined as $U = \sum_{T_i \in \Psi} u_i$.

It is assumed that each task T_i has a fixed priority level η_i and the priorities of different tasks are assumed to be different. That is, for any two tasks T_i and T_j $(i \neq j)$, either $\eta_i > \eta_j$ or $\eta_i < \eta_j$, denoting whether T_i has higher or lower priority than T_j, respectively. Therefore, tasks can be totally ordered according to their priorities.

In addition, each task T_i also has a parameter θ_i to indicate its *execution preference*, which can be either ASAP or ALAP [6]. Based on the preferences of tasks, we can partition them 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. When all tasks have ASAP (or ALAP) preference, they can be optimally scheduled by the RMS [11] (or Dual-Priority [4]) scheduler. Hence, in this work, we consider task sets that consists of both ASAP and ALAP tasks (i.e., both Ψ_S and Ψ_L are non-empty).

We note that, as opposed to stringent deadline constraints, the preferences of tasks are not *hard* constraints and just provide guidelines (or *soft* requirements) on how early or late the tasks' instances should be *preferably* executed. As long as the timing constraints are met, a feasible schedule that provides better fulfillment of tasks' preferences can lead to less execution overhead and/or energy consumption [7].

In [6], based on the aggregated executions of ASAP/ALAP tasks within certain intervals, we have defined optimal (feasible) schedules in terms of fulfilling the preference requirements of tasks. However, due to the conflicting demands from ASAP and ALAP tasks regarding the placement of idle times in schedules, finding a feasible schedule that can optimally fulfill the preference requirements of both ASAP and ALAP tasks may not be always possible [6].

In fixed-priority scheduling, the priorities of tasks directly affect the execution order of their task instances and thus the fulfillment of their preference requirements. Ideally, ASAP tasks should have higher priorities to get them executed early while ALAP tasks should have late executions. However, exploring the optimal priority assignment of tasks that preserves schedulability while maximally fulfilling their execution preferences is beyond the scope of this paper and will be left for our future work.

Problem Description: In this work, for a set of tasks that are schedulable under conventional fixed-priority scheduling with a given priority assignment, we focus on how to manage the executions of the tasks to better fulfill their preference requirements while maintaining the schedulability of the tasks.

B. Dual-Priority Scheduling and Promotion Times

In this section, we first review how the executions of periodic tasks can be postponed using the Dual-Priority (DP) scheduling framework [4]. The main objective of the DP scheduler is to improve the response time of soft real-time aperiodic tasks when they are executed on the same processor as a set of periodic tasks. For that purpose, the DP scheduler utilizes three runtime queues: the upper queue, middle queue and lower queue, and executes tasks from the queues in that order.

Specifically, the middle queue is used to hold the aperiodic soft real-time tasks, which are executed when the upper queue is empty. The periodic tasks are put into the lower queue at the time of their arrival to postpone their executions and thus provide opportunities to execute aperiodic tasks early in order to improve their response time. However, in order to prevent the soft real-time aperiodic tasks from causing deadline misses for periodic tasks, these tasks are promoted to the upper queue after a certain amount of time, which is called the *promotion time*.

Hence, it is critical to properly derive the promotion time for periodic tasks, which determines how long their executions can be postponed. Considering only the periodic tasks with a given fixed-priority assignment, the response time of a task T_i can be found iteratively as follows [1, 10]:

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

$hp(T_i) = \{T_x | T_x \in \Psi \wedge \eta_x > \eta_i\}$ denotes the set of tasks in Ψ that have higher priorities than that of task T_i. Suppose that the periodic tasks are schedulable with their given priorities under the fixed-priority scheduler. We know that the above equation will eventually converge (i.e., $R_i^{k+1} = R_i^k$) and then task T_i's response time can be set accordingly as $R_i = R_i^k$.

If the tasks are schedulable, $R_i \leq p_i$ will hold for all tasks. Then, the promotion time of task T_i can be found as [4]:

$$\gamma_i = p_i - R_i \tag{2}$$

Therefore, task T_i can be safely delayed for γ_i time units in the lower queue before entering the upper queue without missing its deadline. Note that, to provide better response time for future aperiodic tasks, the DP scheduler also adopts the work-conserving strategy. That is, when there are no ready tasks in the upper and middle queues, the periodic tasks in the lower queue will be executed according to their priorities regardless of their promotion times [4].

Algorithm 1 : 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 an event at time $t$ involving task $T_k$;
   Current running task is denoted by $T_c$;
2: if ($T_k \in \Psi_L$ arrives at time $t$ AND $\gamma_k > 0$) then
3:     Enqueue($T_k$, $\mathcal{Q_W}$); 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 $\mathcal{Q_W}$
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
```

III. Preference-Oriented Fixed-Priority (POFP) Scheduler

To effectively address the preference requirements of ASAP and ALAP tasks, we have developed two basic principles for designing preference-oriented scheduling algorithms [6]: a) at any time t, if there are ready ASAP tasks in Ψ_S, the scheduler should not let the processor idle; and b) at any time t, if all ready tasks are ALAP tasks in Ψ_L, 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. From [6], we also know that these two principles can pose conflicting demands when making scheduling decisions, especially for task sets with utilization strictly less than 100%.

A. The POFP Scheduling algorithm

In this work, we focus on the second principle to handle ALAP tasks and investigate scheduling techniques to delay their executions. However, in contrast to the dynamic-priority based preference-oriented schedulers [6] where the delay for ALAP tasks needs to be determined at runtime, we first investigate how to derive the static delay for ALAP tasks for fixed-priority scheduling. For that purpose, we adopt the concept of *promotion time* in Dual-Priority scheduling [4], and propose a *preference-oriented fixed-priority (POFP)* scheduling algorithm with the dual-queue technique [9].

Specifically, in the POFP scheduler, the promotion times are calculated offline according to Equation (2), but *only* for ALAP tasks. At runtime, in addition to the *ready* queue $\mathcal{Q}_R$ that holds the ready tasks, POFP utilizes a second (i.e., *waiting*) queue $\mathcal{Q}_W$ to hold the ALAP tasks that have arrived. Only the tasks in $\mathcal{Q}_R$ are ready for execution and they are executed in the order of their priorities. While ASAP tasks enter the ready queue $\mathcal{Q}_R$ and are ready for execution when they arrive, an ALAP task T_i becomes ready for execution only after its promotion time (i.e., γ_i time units after its arrival time) and is moved to $\mathcal{Q}_R$ at that time.

Basically, POFP exploits the waiting queue $\mathcal{Q}_W$ to delay the executions of ALAP tasks (regardless of their priorities) until their promotion times, which provides opportunities for ASAP tasks to be executed at earlier times. Also, in contrast to the Dual-Priority scheduling [4], as long as the ready queue $\mathcal{Q}_R$ is empty, POFP will let the processor idle, even if there are ALAP tasks in the waiting queue $\mathcal{Q}_W$. That is, POFP is not a work-conserving scheduler.

The basic steps of the POFP scheduler are given in Algorithm 1, 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 waiting queue $\mathcal{Q}_W$ to the ready queue $\mathcal{Q}_R$. An ALAP task T_k with promotion time $\gamma_k > 0$ will be put into the waiting queue by the function *Enqueue(T_k, $\mathcal{Q}_W$)*. Moreover, a timer with its promotion time is set (lines 2 and 3).

When 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 waiting queue until their promotion times. When an ALAP task is promoted or an ASAP task arrives at the invocation time, 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).

Note that, compared to the conventional fixed-priority schedulers, only the promotion events for ALAP tasks are additional scheduling events for POFP and the processing of such events is the same as a normal task arrival event. Therefore, the run-time complexity of POFP will be at the same level as that of the fixed-priority scheduler.

B. An Example: PO-RMS vs. RMS

We further illustrate how the preference-oriented fixed-priority scheduler works through a concrete example. In particular, we consider the well-known RMS scheduler [11], where the priorities of tasks are inversely related to their periods. That is, tasks will smaller periods have higher priority. RMS is the optimal fixed-priority scheduler for implicit-deadline tasks with synchronous arrival patterns [11]. When two tasks have the same period, we assume that the task with smaller index has higher priority. Once the RMS priorities of tasks are determined, the corresponding POFP scheduler is called PO-RMS.

The example task set has four tasks, namely, $T_1(1,5)$, $T_2(3,10)$, $T_3(1,5)$ and $T_4(1,10)$ with $\phi_i = 0\ \forall i$. Tasks T_1 and T_2 have ASAP preference while T_3 and T_4 have ALAP preference. That is, $\Psi_S = \{T_1, T_2\}$ and $\Psi_L = \{T_3, T_4\}$. According to the RMS priorities, we have $\eta_1 > \eta_3 > \eta_2 > \eta_4$, and the feasible RMS schedule can be easily found as shown in Figure 1a.

Note that, RMS is a work-conserving scheduler and does not consider tasks' preferences when making scheduling decisions.

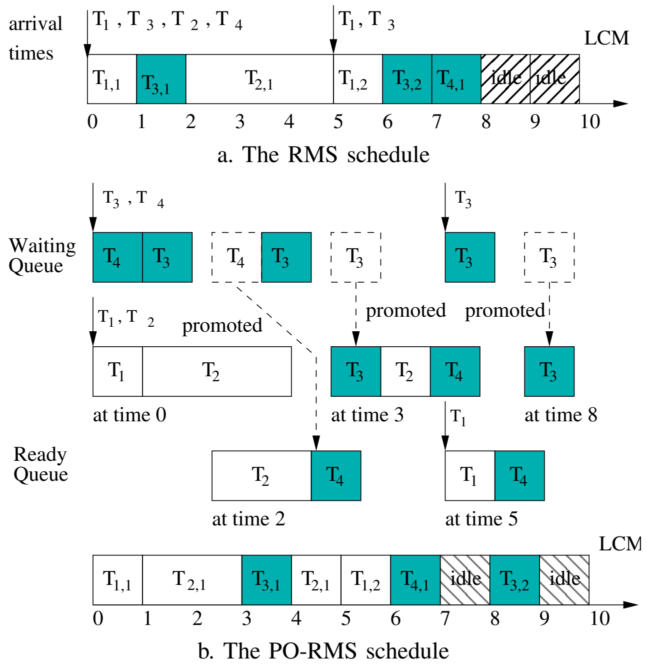


Fig. 1. The RMS and PO-RMS schedules for an example task set of four tasks: $T_1(1,5)$, $T_2(3,10)$, $T_3(1,5)$ and $T_4(1,10)$; $\Psi_S = \{T_1, T_2\}$, $\Psi_L = \{T_3, T_4\}$; $\gamma_3 = 3$ and $\gamma_4 = 2$.

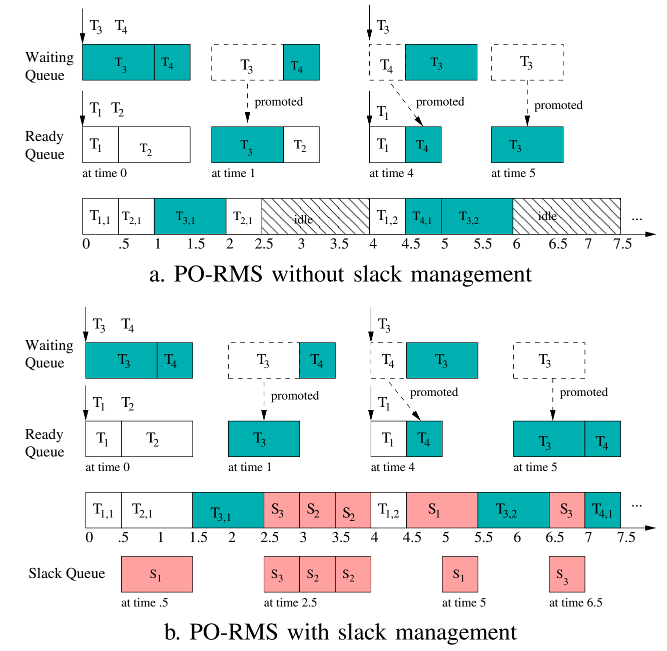


Fig. 2. PO-RMS with online slack management; Here, the task set has four tasks: $T_1(1.5,4)$, $T_2(1,8)$, $T_3(1.5,4)$ and $T_4(1,12)$; $\Psi_S = \{T_1, T_2\}$, $\Psi_L = \{T_3, T_4\}$, $\gamma_3 = 1$ and $\gamma_4 = 4$. At runtime, the actual execution times of tasks are assumed to be $a_1 = 0.5$, $a_2 = 1$, $a_3 = 1$ and $a_4 = 0.5$;

Therefore, the processor is idle only when there are no ready jobs. Moreover, although the ASAP tasks T_1 and T_2 are executed before the execution of ALAP tasks T_3 and T_4 in the RMS schedule, we can see next that the execution sequence can be further improved to better fulfill the preference requirements under PO-RMS.

Based on Equations (1) and (2), the promotion times for the ALAP tasks T_3 and T_4 can be found as: $\gamma_3 = 3$ and $\gamma_4 = 2$. Therefore, as shown in Figure 1b, the instances of task T_3 and T_4 are held in the waiting queue for 3 and 2 time units, respectively, before they are moved to the ready queue and compete with ASAP tasks for the processor. We can see that, as opposed to the RMS schedule where the task instance $T_{3,1}$ is executed at time 1, PO-RMS holds $T_{3,1}$ in the waiting queue until time 3. Hence, the ASAP task instance $T_{2,1}$ gets the chance to run partially before the execution of the ALAP task instance $T_{3,1}$. Similarly, the execution of $T_{3,2}$ is delayed until time 8 with one unit of idle time before its execution. Clearly, we can see that, compared to the RMS schedule, the preference requirements of all tasks are better fulfilled under the PO-RMS scheduler.

IV. Online Techniques for Preference-Oriented Executions

From the above discussions, we can see that POFP can effectively delay the execution of ALAP tasks until their promotion times. However, once such tasks are promoted to the ready queue, POFP treats them in the same way as ASAP tasks and no further delay will be imposed on their executions. On the other hand, it is well-known that real-time tasks typically take a small fraction of their worst-case execution times (WCETs) [5] and significant amount of slack time can be expected at runtime. Such slack time can be exploited to execute ASAP tasks at earlier times and to further delay the executions of ALAP tasks. Before presenting the online techniques for preference-oriented executions of tasks, in what follows, we first illustrate the idea through an example.

Consider another task set with four tasks: $T_1(1.5,4)$, $T_2(1,8)$, $T_3(1.5,4)$ and $T_4(1,12)$, where $\Psi_S = \{T_1, T_2\}$ and $\Psi_L = \{T_3, T_4\}$. Again, it is assumed that tasks have RMS priorities with $\eta_1 > \eta_3 > \eta_2 > \eta_4$. The promotion times for tasks T_3 and T_4 are found as $\gamma_3 = 1$ and $\gamma_4 = 4$, respectively. At runtime, it is assumed that most tasks take less time than their WCETs and their actual execution times are assumed to be $a_1 = 0.5$, $a_2 = 1$, $a_3 = 1$ and $a_4 = 0.5$.

Without taking the slack time into consideration, the schedule for the first few instances of the tasks under PO-RMS is shown in Figure 2a. 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, $T_{4,1}$ gets promoted at time 4 and is executed right after the early completion of $T_{1,2}$.

On the other hand, if slack time is explicitly managed at runtime, we can have one unit of slack S_1 due to the early completion of $T_{1,1}$ at time 0.5. Moreover, we assume that the slack S_1 inherits the priority of its contributing task T_1. Since the slack S_1 has higher priority than that of $T_{2,1}$ in the ready queue, the processor will be "allocated" to the slack, which in turn can *wrap* the execution of $T_{2,1}$ by lending its time to the task instance as shown in Figure 2b. When $T_{3,1}$ is promoted to the ready queue at time 1, its priority is lower than that of the slack S_1 to which the processor is allocated and no preemption occurs. Hence, through such wrapped execution, the ASAP task $T_{2,1}$ can complete before the ALAP task $T_{3,1}$ gets executed.

However, when the processor is allocated to the slack and there is no ASAP task in the ready queue, this will enable the processor to idle and delay the execution of ALAP tasks (even if they have been promoted to the ready queue). For instance, as shown in Figure 2b, when the task instance $T_{1,2}$ completes early, the processor is allocated to another generated slack S_1 at time 4.5 but then it is left idle in order to delay the execution of $T_{4,1}$ that has been promoted to the ready queue at time 4.

A. Slack Management with Wrapper-Tasks

To generalize the above idea and enable slack times to compete for the processor, we extend the *wrapper-task* based slack management, which has been studied for dynamic priority based task systems [13], to the fixed-priority setting. Basically, each piece of slack time will be represented by a wrapper-task with two parameters (c, η). Here, c denotes the size of the slack and η represents the slack's priority, which is inherited from the task giving 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 of them are empty, POFP will let the processor idle while waiting for the new arrival of tasks and/or the promotion of ALAP tasks in the waiting 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. If the ready queue is empty (i.e., $T_k = NULL$) or $\eta_h > \eta_k$ but there is no ASAP task in the ready queue $\mathcal{Q}_R$, the slack (represented by the wrapper task S_h) will get the processor and also keep it idle for the interval of its allocated time, which effectively delays the executions of ready ALAP tasks in $\mathcal{Q}_R$ (if any). For the case of the empty slack queue (i.e., $S_h = NULL$) or $\eta_k > \eta_h$, POED will dispatch task T_k normally from the ready queue $\mathcal{Q}_R$.

An 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 very similar to those for dynamic priority based scheme [13], and are omitted due to space limitations.

B. Dummy Task Technique to Exploit Static Spare Capacity

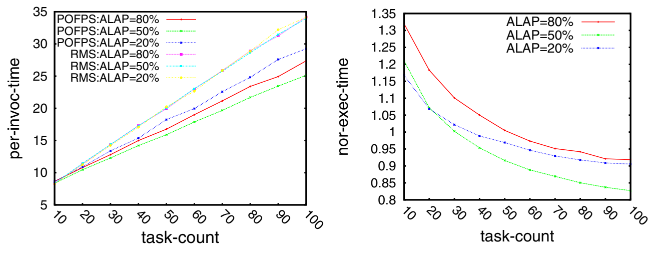
For a given set of tasks that are schedulable under fixed-priority scheduling, it is more likely that the system is not fully utilized (i.e., $U < 1$). However, the wrapper-task technique discussed in the last section is designed to handle dynamic slack from early completion of tasks and cannot directly utilize such spare system capacity. In [13], we have utilized a *dummy task* T_0 to represent the spare system capacity and to periodically introduce slack time into the system at runtime. Moreover, considering the utilization bound for EDF scheduling, the utilization of T_0 is set as $u_0 = 1 - U$.

Following the similar approach as in [13], we can also augment a given task set with a dummy task T_0. Here, in addition to its timing parameters (c_0, p_0), we need to determine T_0's priority η_0. From the discussions in the last section, slack time needs to have a higher priority to wrap an ASAP task for its early execution and to delay the execution of ALAP tasks. Therefore, it is desirable to assign the highest priority to the dummy task T_0; that is, $\eta_0 > \eta^{max} = \max\{\eta_i | T_i \in \Psi\}$. In addition, the period of T_0 determines how often slack is introduced to the system at runtime and can have a direct impact on how the preference requirements of tasks are fulfilled [13].

Note that, the choice of T_0's timing parameters and priority should not compromise the schedulability of the augmented task set. Considering the complex interplay between tasks' schedulability and their priorities and timing parameters, it is much more difficult to find the appropriate (c_0, p_0, η_0) values for T_0 than that for the dynamic-priority based EDF scheduling [13]. One possible approach can be exploiting the response time analysis [1, 10] and iteratively examining the feasibility of the dummy task T_0's priority assignment (from the highest to lowest) and associated timing parameters.

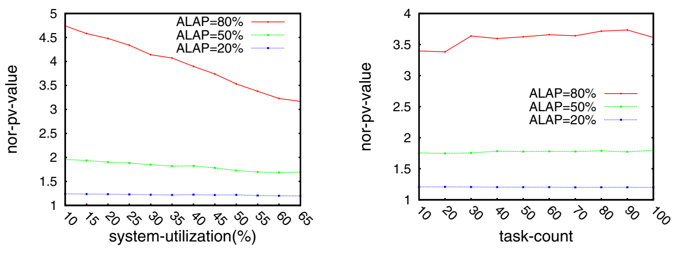
Dummy Task Technique for RM Scheduling: Next, focusing on the RM scheduler, we discuss a conservative but simple approach to determine T_0's priority and timing parameters. Suppose that the given task set is schedulable under RMS and that $U \leq U^b_{rms}(n) = n(2^{1/n} - 1)$ holds, where n is the number of tasks and $U^b_{rms}(n)$ is the utilization bound of the RMS scheduler [11]. In this case, we can safely set $u_0 = U^b_{rms}(n + 1) - U$ without compromising the schedulability of the tasks. In addition, we can have $p_0 = p^{min} = \min\{p_i | T_i \in \Psi\}$ and $c_0 = u_0 \cdot p_0$. By setting the period of T_0 as the smallest period of the tasks, we know that the dummy task T_0 will have the highest priority η_0.

While T_0 can help to transform system spare capacity to slack at runtime, it also affects the promotion time of the lower priority ALAP tasks. Actually, as our evaluation results in the next section show, the negative effects of the dummy task on the reduced promotion time of ALAP tasks, by causing ALAP tasks be promoted and executed at earlier times, can overshadow its benefit for introducing the slack time from spare capacity. This is quite different from dynamic-priority based preference-oriented scheduling [6].



a. Per-invocation overhead b. Normalized overall overhead

Fig. 3. Runtime overhead for PO-RMS and RMS; $U = 0.5$.



a. Impact of U with 10 tasks b. Impact of task count with $U = 0.5$

Fig. 4. Normalized PV values of PO-RMS vs. RMS.

V. Evaluations and Discussions

In this section, we evaluate the proposed POFP scheduler in terms of its runtime overhead as well as its performance on how well the preference requirements of tasks are fulfilled through extensive simulations. In particular, we focus on a typical fixed-priority scheduler RMS and compare it against the PO-RMS scheduler. To this aim, we developed a discrete event simulator using C++ and implemented both RMS and PO-RMS schedulers.

In the simulations, we consider synthetic task sets with 10 to 100 tasks, where the utilization of each task is generated using the *UUniFast* scheme proposed in [2]. The period of each task is uniformly distributed within the range of $[10, 100]$, and its WCET is set accordingly. We vary the system load (i.e., utilization U), the workload of ALAP tasks, as well as the number of tasks and evaluate their effects on the performance of the proposed PO-RMS scheduler. In the figures below, each data point corresponds to the average result of 100 task sets.

A. Scheduling Overhead

We first evaluate the scheduling overhead of the PO-RMS scheduler and compare it against RMS. Note that, as we discussed in Section III, PO-RMS has the same complexity as that of RMS, which depends on the number of tasks in the system. Therefore, we vary the number of tasks per task set from 10 to 100. To ensure that the generated task sets are schedulable, we set the system utilization as $U = 0.5$.

Figure 3a shows the average per-invocation overhead for both PO-RMS and RMS. We consider cases where ALAP tasks contribute to the 20%, 50% and 80% of the total workload, which are denoted as *ALAP*=20%, *ALAP*=50% and *ALAP*=80%, respectively. From the results, we can see that the runtime overhead of RMS only depends on the number of tasks in a task set, which increases with the number of tasks.

However, as PO-RMS exploits an additional runtime queue to handle ALAP tasks, there are fewer number of tasks in each queue compared to the ready queue in RMS. Thus, on average, it takes less time for PO-RMS to process each invocation compared to RMS, especially for the cases with balanced ASAP and ALAP workloads (i.e., ALAP=50%). Such runtime overhead difference becomes larger as there are more tasks in each task set.

Figure 3b further shows the normalized overall runtime overhead of PO-RMS with that of RMS as the baseline. The overall runtime overhead of PO-RMS is higher than that of RMS for task sets with small number of tasks due to the extra promotion events for ALAP tasks in PO-RMS. However, when there are more tasks in a task set, the overhead for additional promotion events in PO-RMS can be quickly offset by its much reduced per-invocation overhead, which leads to smaller overall overhead for PO-RMS when compared to that of RMS.

B. Fulfillment of Tasks' Preference Requirements

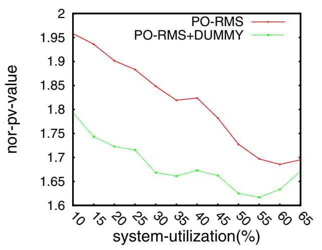
Considering the preemptive nature of the scheduler, it is difficult to quantify how well the preference requirements of tasks are fulfilled. As in [6], we use the *preference value (PV)* metric, which is defined over the completion and start times for ASAP and ALAP tasks, respectively. In particular, the preference value for a task instance $T_{i,j}$ is defined as [6]:

$$PV_{i,j} = \begin{cases} \frac{ft_{max} - ft}{ft_{max} - ft_{min}} & \text{if } T_i \in \Psi_S; \\ \frac{st - st_{min}}{st_{max} - st_{min}} & \text{if } T_i \in \Psi_L. \end{cases} \quad (3)$$

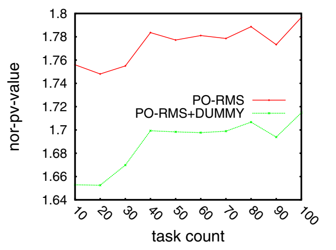
where st and ft denote the task's start and complete times, respectively. Moreover, ft_{min} and ft_{max} represent the earliest and latest completion times of an ASAP task instance, respectively. Suppose the task instance arrive at time r and its task has WCET c and period p; we have$ft_{min} = r + c$ and $ft_{max} = r + p$. Similarly, $st_{min} = r$ and $st_{max} = r + p - c$ represent the earliest and latest start times of an ALAP task instance, respectively.

We can see that the preference value for a task instance has the value within the range of $[0, 1]$. A larger value of $PV_{i,j}$ indicates that $T_{i,j}$'s preference requirement has been fulfilled better. 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 and the overall preference value of a task set is the average preference value of all its tasks. In what follows, the normalized preference values achieved for all tasks under PO-RMS are shown with that of RMS as the baseline.

Effects of the system utilization and task numbers: Figure 4 first shows the effects of system utilization and the number of tasks on the achieved preference values for tasks under PO-RMS. Clearly, we can see that PO-RMS can obtain better preference values for tasks (up to 4 times) when compared to those obtained with RMS, especially when there are more ALAP tasks (i.e., *ALAP*=80%). This comes from the fact that PO-RMS exploits the waiting queue



a. Impact of U with 10 tasks



b. Impact of task count ($U = 0.5$)

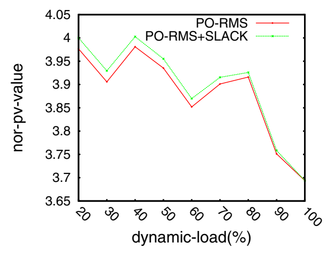
Fig. 5. The effects of dummy task; $ALAP = 50\%$;

and promotion times to delay the start time of ALAP tasks, thereby providing good opportunities for ASAP tasks to complete early. However, when there are only a few number of ALAP tasks, the performance of PO-RMS gets close to that of RMS as ASAP tasks are processed in the same way in both schedulers.

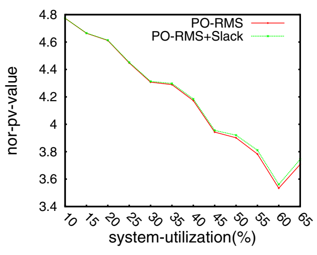
Effects of the dummy task technique: For task sets with different system utilizations and task numbers, Figure 5 shows how the dummy task can affect the performance of PO-RMS. Here, we assume that all tasks take their WCETs and all available slack at runtime is introduced by the dummy task that transforms the spare capacity. For simplicity, the utilization of the dummy task is set as $u_0 = 0.69 - U$, where 0.69 is the asymptotical utilization bound for the RMS scheduler [11].

From the results, we can see that, when the dummy task exploits the spare system capacity, PO-RMS performs worse in terms of fulfilling the preference requirements of tasks. This is because ALAP tasks are promoted to the ready queue at earlier times due to their decreased promotion times as the result of the highest-priority dummy task. Since the preference values for ALAP tasks rely on their start times only, such early promotion of ALAP tasks enables them to start earlier and overshadows the benefits of the slack time introduced by the dummy task from the spare system capacity.

In Figure 5a, as system utilization becomes higher, each task becomes larger since there are only 10 tasks per task set. The increased task sizes make it harder for ASAP tasks complete at earlier time and also may also force ALAP tasks start at earlier times, which in turn leads to reduced preference values. This trend can also be seen as there are more tasks with fixed system utilization as shown in Figure 5b.



a. Impact of dynamic load; $U = 0.5$



b. Impact of U; $L^{dyn} = 50\%$

Fig. 6. The effects of slack management; 10 tasks and $ALAP = 80\%$.

Effects of the online technique: By exploiting wrapper-tasks to manage online slack, the performance of PO-RMS is further shown in Figure 6. No dummy task is incorporated as it cannot improve the performance of PO-RMS. Moreover, the amount of online slack from early completion of tasks is controlled by a parameter dynamic load L^{dyn}, which indicates the percentage of their WCETs that will be taken by the tasks. That is, smaller L^{dyn} values mean more dynamic slack.

It is interesting to see that the performance improvement of the online technique is quite limited. One possible reason for this is the fact that the slack cannot help much to postpone the start times of ALAP tasks or completion times of ASAP tasks, which are the main factors for the performance values.

VI. Conclusions

In this work, for real-time tasks where some tasks are executed preferably early while others late, we proposed a *preference-oriented fixed-priority (POFP)* scheduling algorithm. The basic idea is to use the promotion time and an additional runtime queue to delay the execution of late tasks, which also enable other tasks to run earlier. Online techniques with slack management are also investigated. The evaluation results for RMS schedulers show that the dual-queue POFP scheduler can effectively address the execution preferences tasks. While online technique with wrapper-tasks can slightly improve the performance of POFP, the dummy task can lead to worse performance as it can reduce the promotion time of late tasks.

References

[1] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, Sep 1993.

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

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

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

[5] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Proc. of The International Conference on Computer-Aided Design*, pages 598–604, Nov. 1997.

[6] Y. Guo, H. Su, D. Zhu, and H. Aydin. Preference-oriented scheduling framework for periodic real-time tasks (extended version). Technical Report CS-TR-2013-007, Univ. of Texas at San Antonio, 2013.

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

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

[9] M.A. Haque, H. Aydin, and D. Zhu. Energy management of standby-sparing systems for fixed-priority real-time workloads. In *Proc. Of the Second Int'l Green Computing Conference (IGCC)*, Jun. 2013.

[10] M. Joseph and P.K. Pandya. Finding response times in a real-time system. *BCS Comput. J.*, 29(5):390–395, 1986.

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

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

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