International Scholarly Research Network
ISRN Software Engineering
Volume 2012, Article ID 681985, 12 pages
doi:10.5402/2012/681985

Research Article

On-Line Real-Time Service-Oriented Task Scheduling Using TUF

Shuo Liu,[1] Gang Quan,[1] and Shangping Ren[2]

[1] *Electrical and Computer Engineering Department, Florida International University, Miami, FL 33174, USA*
[2] *Computer Science Department, Illinois Institute of Technology, Chicago, IL 60616, USA*

Correspondence should be addressed to Shuo Liu, sliu005@fiu.edu

Received 17 January 2012; Accepted 27 March 2012

Academic Editors: G. Gössler, J. A. Holgado-Terriza, and U. K. Wiil

We present our approach to real-time service-oriented scheduling problems with the objective of maximizing the total system utility. Different from the traditional utility accrual scheduling problems that each task is associated with only a single time utility function (TUF), we associate two different TUFs—a profit TUF and a penalty TUF—with each task, to model the real-time services that not only need to reward the early completions but also need to penalize the abortions or deadline misses. The scheduling heuristics we proposed in this paper judiciously accept, schedule, and abort real-time services when necessary to maximize the accrued utility. Our extensive experimental results show that our proposed algorithms can significantly outperform the traditional scheduling algorithms such as the Earliest Deadline First (EDF), the traditional utility accrual (UA) scheduling algorithms, and an earlier scheduling approach based on a similar model.

1. Introduction

With the proliferation of the Internet the opportunity has come to provide real-time services over the cloud infrastructure [1–3]. From media on-demand service by Netflix to online gaming by Nintendo, from Amazon's e-commerce to Google's free turn-by-turn direction service over the phone, all these indicate that we are entering a new era of real-time computing. These real-time services are usually built on Internet-based cloud infrastructure, not only because they need to be highly available, but also because they generally rely on large data sets that are most conveniently hosted in large data centers. According to O'Reilly [4], the entire Internet is becoming not only a platform, but also an operating system itself, and "the future belongs to services that respond in real time to information provided either by their users or by nonhuman sensors" [1].

For real-time services, timeliness is a major criterion of judging real-time service-quality levels. Due to the high variability of the Internet, real-time service-oriented applications are more of soft real-time in nature. Guaranteeing hard deadlines for real-time services would be neither practical nor necessary in most scenarios. In this regard, besides pre-assigned deadlines, some other timing information that is closely related to quality of service (QoS) become important metrics when processing real-time service requests.

To improve the real-time service performance, one approach is to employ the traditional UA approach [5, 6]. In [7], Jensen et al. first proposed to associate each task with a TUF, which indicates that the completion of a task will assign the system a certain value of utility, and the utility value varies with the time when the task is finished. Specifically, a TUF as shown in Figure 1(a) describes the value or utility accrued by a system at the time when a task is completed. Based on this model, there were extensive research results published on the topic of UA scheduling [8–12]. For example, in [8], the author proposed an algorithm, Generic Benefit Scheduling (GBS), based on TUF to schedule activities that subject to various time and mutual exclusive resource constraints. Utility density is implemented as the activity's priority metric. While Jensen's definition of TUF allows the semantics of soft timing constraints to be more precisely specified, all these variations of UA-aware scheduling algorithms imply that the aborted tasks neither increase nor decrease the accrued value or utility of the system.

We believe that, to further improve the performance of real-time services over the Internet, it is important to not

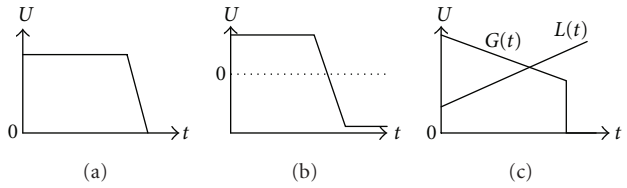


Figure 1: Time utility functions.

only measure the profit when completing a task in time, but also account for the penalty when a task is aborted or discarded. In addition, the time at which a real-time service is aborted is also important. First, the more service requests are discarded and the longer a client waits fruitlessly, the lower the quality of service client receives. As a result, service providers have to pay higher cost, either in the form of monetary compensation or losing future service requests from unsatisfied clients. Second, before a task is aborted or discarded, it needs to consume system resources, including network bandwidth, storage space, and processing power, and thus can directly or indirectly affect the system performance. This is especially true if we assume real-time applications may be dissected and migrated across an entire cloud infrastructure [13, 14]. Therefore, if a real-time task is deemed to miss its deadline with no positive semantic profit, a better choice should be one that can detect it and discard it as soon as possible.

A number of models [15–18] were proposed to account for the penalty when a real-time service request is discarded or misses its deadline. For example, Bartal et al. [15] studied the online scheduling problem when penalties have to be paid for rejected real-time tasks. Chun and Culler [16] and Irwin et al. [18] adopted an extended time utility function as shown in Figure 1(b). According to this model, a decay rate is associate with each real-time task, reflecting the increasing risk of completing the task late in the future. Therefore, when a real-time task is completed late, it earns a negative utility, indicating a penalty rather than the profit. These models, however, do not account for different penalties when aborting a real-time task at different times.

In this paper, we study the real-time service scheduling problem based on a task model similar to the one proposed by Yu et al. [19]. Specifically, a task is associated with two different TUFs, as shown in Figure 1(c), a profit TUF ($G(t)$), and a penalty TUF ($L(t)$). The system takes a profit (determined by its profit TUF) if the task completes by its deadline and suffers a penalty (determined by its penalty TUF) if the task misses its deadline or is dropped before its completion. The penalty to abort a pending real-time service request can be the same or different from that of missing the deadline, which depends on the characteristics of the penalty TUF. Different from Yu's model, we use a novel method to calculate task's utility and use utility density to describe a task's priority. The "critical time" for each task is more strict, and we add an admission step when there is a new task arrives since congested ready queue will decrease the system's performance. It is a waste of system resources if tasks wait fruitlessly.

We conduct analysis on how to optimize the accrual utility when scheduling a set of aperiodic real-time service requests. We first assume that the service requests are scheduled in a nonpreemptive manner. Two scheduling methods are presented. The first scheduling method is developed based on the concept of "opportunity cost" [20] from economics that can help evaluate the fulfillment of a real-time service request. The second method employs a more sophisticated but robust method to formulate the potential system profit by developing a speculated execution order for the ready tasks. We then extend our scheduling methods to deal with real-time services that may preempt each other. In addition to carefully choosing the ready task to run, our scheduling methods judiciously discard pending requests, abort task executions, cautiously preempt current running tasks, and therefore can achieve better performance. Our experimental results also show that the proposed algorithms can significantly outperform the traditional scheduling approaches such as the Earliest Deadline First (EDF), the traditional UA scheduling algorithm, that is, the Generic Utility Scheduling (GUS) [8], the Risk/Reward algorithm [18], and a previous scheduling approach based on a similar model, that is, the Profit Penalty-aware scheduling (PP-aware scheduling) [19].

The rest of the paper is organized as follows. Section 2 describes the models we used in the paper, formulates the problem formally, and then presents a motivation example. Sections 3 and 4 introduce our scheduling approaches in details. Experiment results are discussed in Section 5, and we make conclusions in Section 6.

2. Preliminary

In this section, we first introduce the task and architecture models considered in this paper. We then use an example to motivate our research.

2.1. Task Model and System Architecture. In this paper, we consider a single sequence of randomly arrived real-time tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$, with τ_i defined using the following parameters:

(i) $[B_i, W_i]$: the best case execution time B_i and the worst case execution time W_i of τ_i;

(ii) D_i: the relative deadline of τ_i;

(iii) $f_i(T)$: the probability density function for the execution time of τ_i;

(iv) $G_i(t)$: the profit TUF, which represents the profit accrued when a task is completed at time t. We assume $G_i(t)$ is a nonincreasing unimodal function before its deadline, that is, $G_i(t_p) \geq G_i(t_q)$ if $t_p \leq t_q$, and $G_i(D_i) = 0$;

(v) $L_i(t)$: the penalty TUF, which represents the penalty suffered when a task is discarded or aborted at time t. We assume that $L_i(t)$ is a nondecreasing unimodal function before its deadline, that is, $L_i(t_p) \leq L_i(t_q)$ if $t_p \leq t_q$, and a task is immediately discarded once it missed its deadline.

Note that, even though the deadline of a task can be implicitly defined using appropriate profit and penalty TUFs, we opt to list the deadline explicitly as a parameter for ease of presentation. As shown above, a task is associated with both a profit function and a penalty function with function values varying with time. Therefore, while executing a task the system has a potential to gain profit, it also has a potential to encounter a penalty at a later time. The system performance is therefore evaluated by its total utility after penalty is deducted from profit.

We assume an architecture for the service provider depicted in Figure 2. Specifically, the service provider contains two computing components, that is, the manager host and the execution host, that can work concurrently. The manager host is in charge of accepting, scheduling, and aborting real-time service requests, and the execution host fulfills the selected service requests from the manager host. There may be one or more execution hosts for each service provider. We limit our research to one single execution host in this paper.

With the task and architecture model introduced as above, our problem can be formally formulated as follows.

Problem 1. Given a task set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ as described above, develop online scheduling methods such that the total accrued utility is maximized.

2.2. A Motivation Example. The problem defined in Problem 1 is NP hard since a simpler version of this problem, that is, the total weighted completion time scheduling problem [21], is shown to be NP hard. To show that the commonly used scheduling policy such as the EDF or the traditional utility accrual approach such as the GUS [8] become ineffective to address this problem, consider the example shown in Figure 3.

Assume that two real-time service requests arrive at the same time ($t = 0$) with their characteristics shown in Figure 3. We assume that the actual processor time of each request is evenly distributed between the interval of its best case and worst case execution time. To make the example more concrete, we assume that the actual processing times for these two requests are 50 and 60, respectively.

When EDF is applied, τ_1 has a higher priority than τ_2 and is executed first. It completes at $t = 50$ with profit of $G_1(50) = 180 - 2 \times 50 = 80$. Then τ_2 starts its execution. At $t = 100$, it misses its deadline and will incur more penalty if its execution continues. Therefore, the execution of τ_2 is discarded at $t = 100$ with penalty of $L_2(100) = 2 \times 100 = 200$. The total utility to process these two requests is therefore $80 - 200 = -120$.

The GUS algorithm chooses the task with the largest expected profit density to execute first. Under our task model, the expected profit of τ_1 and τ_2, that is, $\overline{G}(\tau_1)$ and $\overline{G}(\tau_2)$, can be calculated as

$$\begin{aligned} \overline{G}(\tau_1) &= \int_{20}^{80} (180 - 2t) \times \frac{1}{80 - 20} dt = 80, \\ \overline{G}(\tau_2) &= \int_{20}^{120} (400 - 3t) \times \frac{1}{120 - 20} dt = 190. \end{aligned} \tag{1}$$

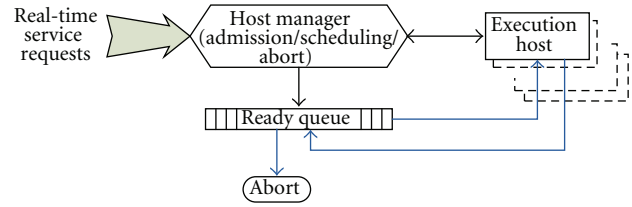


Figure 2: The architecture for the service provider.

At $t = 0$, we have no knowledge of the actual execution time of τ_1 and τ_2, and a reasonable estimate would be the one using their expected values, that is, 50 and 70, respectively. As a result, τ_2 is chosen to execute first since its expected profit density (expected profit divided by expected execution time) 190/70 is higher than that of τ_1, that is, 80/50. It completes at $t = 60$ with profit of $G_2(60) = 400 - 3 \times 60 = 220$. Then τ_1 starts its execution. At $t = 80$, it misses its deadline and is aborted to prevent even higher loss. The total utility to process these two requests is therefore $220 - 80 = 140$.

An astute reader may immediately point out that, after τ_2 completes at $t = 60$, it is less likely that τ_1 can complete by its deadline, given that its best case execution time is 20. Therefore, τ_1 should be immediately aborted at $t = 60$ with a total utility profit of $220 - 60 = 160$. Note that, after τ_2 is selected to execute first, its expected execution time would be 70. Given the expected execution time of τ_1 being 50, it is more likely that τ_1 will miss its deadline. Therefore, a better scheduling decision would discard it at $t = 0$ with total profit of 220 in this case, as the third schedule shown in Figure 3.

In our example, we can see that the EDF has the worst performance since it makes scheduling decisions solely based on tasks' deadlines. The traditional utility accrual scheduling method takes the individual value function into consideration and therefore can achieve better performance. The problem, however, is that the traditional utility accrual scheduling approaches (such as GUS) fail to take the abortion or discard penalty and the timing for the abortion or discard penalty into consideration. Clearly, how to select the appropriate task to run so as to maximize the profit and how to discard real-time tasks as soon as possible in overloaded situations in order to control the penalty are vital for our research problem.

3. Nonpreemptive Approach

In this section, we present our online nonpreemptive scheduling solutions to address the problem defined in the previous section. Since the execution of a task may gain positive profit or suffer penalty and thus degrade the overall computing performance, judicious decisions must be made with regard to executing a task, discarding or aborting a task, and when to discard or abort a task. In what follows, we present two metrics to measure the expected utility when executing a real-time task, and, based on which, we develop two scheduling algorithms.

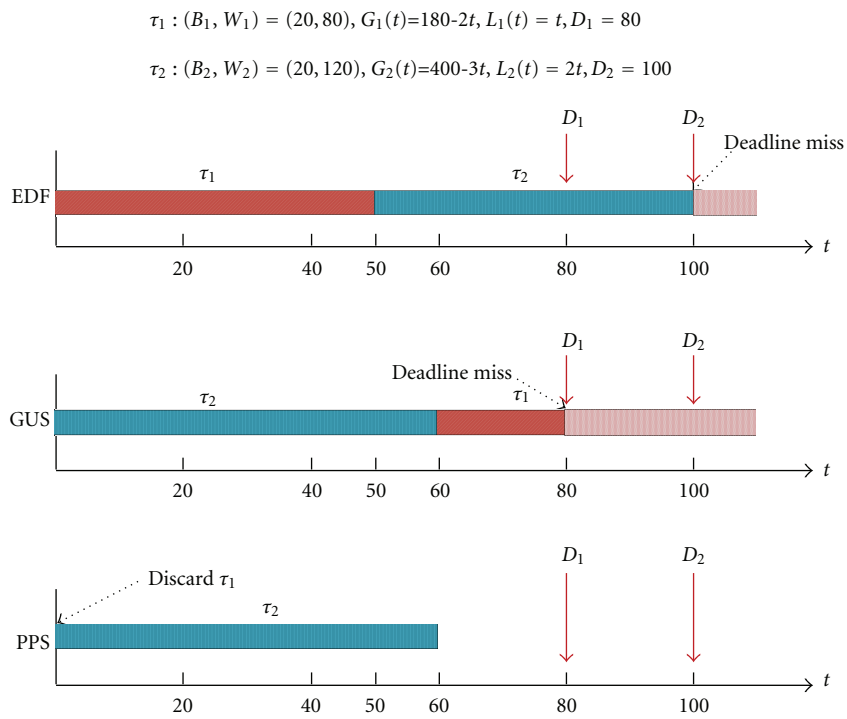


Figure 3: Three different schedules for two real-time tasks τ_1 and τ_2 arriving at the same time $t = 0$.

3.1. The Opportunity Cost-Based Utility Metric. Our first utility metric is built upon the concept of *opportunity cost* [20] in economics. In economics, the *opportunity cost* refers to the value associated with the next best available choice that one has to give up after making a choice. When scheduling a set of real-time tasks at $t = T$, let expected utility of running τ_i alone be $\overline{U_i}(T)$ and its opportunity cost be $\overline{OC_i}(T)$. Then we can conveniently formulate the expected utility $\widetilde{U}(\tau_i, T)$ to run τ_i at $t = T$ as

$$\widetilde{U}(\tau_i, T) = \overline{U_i}(T) - \overline{OC_i}(T). \tag{2}$$

The problem becomes how to calculate $\overline{U_i}(T)$ and $\overline{OC_i}(T)$.

Since the task execution time is not known a prior, we do not know if executing the task will lead to positive profit or loss. Given its probabilistic distribution, we can determine the expected profit and loss statistically. Given a task τ_i with arrival time of r_i, let its predicted starting time be T. Then the expected profit ($\overline{G_i}(T)$) to execute τ_i can be represented as

$$\begin{aligned}\overline{G_i}(T) &= \int_0^{\infty} G_i(t + (T - r_i)) f_i(t \mid t + T < D) dt \\ &= \int_{B_i}^{D_i} G_i(t + (T - r_i)) f_i(t) dt.\end{aligned} \tag{3}$$

Similarly, the expected loss ($\overline{L_i}(T)$) to execute τ_i can be represented as

$$\begin{aligned}\overline{L_i}(T) &= L_i(D) P(t + T > D) \\ &= L_i(D) \int_{D_i - (T - r_i)}^{W_i} f_i(t) dt.\end{aligned} \tag{4}$$

Therefore, the expected utility $\overline{U_i}(T)$ can be represented as

$$\overline{U_i}(T) = \overline{G_i}(T) - \overline{L_i}(T). \tag{5}$$

When $\overline{U_i}(T) > 0$, it means that the probability to obtain positive profit is no smaller than that to incur a loss if we choose to execute τ_i at $t = T$. Since $\overline{G_i}(T)$ is a monotonic decreasing function of T, and $\overline{L_i}(T)$ is a monotonic increasing function of T, $\overline{U_i}(T)$ must be a monotonic decreasing function of T.

Note that even though two tasks may have the same expected utility, they may have different expected execution times. We define a parameter ρ_i to capture the *expected utility density* for task τ_i as follows:

$$\overline{\rho_i}(T) = \frac{\overline{U_i}(T)}{\overline{C_i}}, \tag{6}$$

where $\overline{C_i}$ is the expected execution time of task τ_i. There exists a t_0 such that

$$\overline{\rho_i}(t_0) = 0. \tag{7}$$

The time $t = t_0$ is called the *critical point*. Apparently, when $t > t_0$, it is more likely that it will incur a loss rather than a profit if we choose to execute τ_i. We can further relax (7) by imposing a threshold (δ), that is,

$$\overline{\rho_i}(t_0) \geq \delta. \tag{8}$$

We call δ as the *utility density threshold*.

We next introduce how to formulate the opportunity cost when choosing to run task τ_i at $t = T$. The original concept of "opportunity cost" is the value for the next best available

Input: Let $\{\tau_1, \tau_2, \ldots, \tau_k\}$ be the accepted tasks in the ready queue, and let $\overline{C_i}$ be the expected execution time of τ_i. Let current time be t and let τ_0 be the task currently being executed, expected execution time of τ_0 is $\overline{C_0}$. Let the expected utility density threshold be δ.

if A new task, that is, τ_p arrives **then**
  Accept τ_p if $\overline{\rho}_p(\overline{C_0}) > \delta$;
  Reject τ_p if $\overline{\rho}_p(\overline{C_0}) \leq \delta$;
  Remove τ_j in the ready queue end if $\overline{\rho}_j(\overline{C_0}) \leq \delta$;
end if

If τ_0 is completed **then**
  Choose τ_i with the largest system utility density, that is, $\widetilde{\rho}_i(t) = \max_k \widetilde{\rho}_k(t)$.
  Remove τ_j in the ready queue if $\overline{\rho}_j(\overline{C_i}) < \delta$;
end if

If t = the critical time of τ_0 **then**
  Abort τ_0 immediately;
  Choose τ_i with the largest system utility density, that is, $\widetilde{\rho}_i(t) = \max_k \widetilde{\rho}_k(t)$.
  Remove τ_j in the ready queue if $\overline{\rho}_j(\overline{C_i}) < \delta$;
end if

Algorithm 1: The scheduling algorithm based on opportunity cost.

choice. It is hard to identify the "next best choice" since the exact reason we need the opportunity cost is to set up the preference order when choosing tasks to run. In our metric, the opportunity cost is calculated as the decay of expected utilities by other tasks. Specifically, let the expected utility of τ_j at $t = T$ be $\overline{U_j}$. Then if we choose τ_i to execute at $t = T$ and after its completion, the expected utility of τ_j is reduced to $\overline{U_j}(T + \overline{C_i})$, where $\overline{C_i}$ is the expected execution time of τ_i. Provided that we can remove the task timely when its expected utility is less than zero, we thus define the opportunity cost to run τ_i at $t = T$, that is, $\overline{OC_i}(T)$ as

$$\overline{OC_i}(T) = \frac{1}{n-1} \sum_{j=1, j\neq i}^{n} \max\left(\left(\overline{U_j} - \overline{U_j}\left(T + \overline{C_i}\right)\right), 0\right). \quad (9)$$

With both $\overline{U_i}(T)$ and $\overline{OC_i}(T)$ formulated, we are now ready to introduce our scheduling algorithm. Our nonpreemptive scheduling algorithm works at scheduling points that include the arrival of a new task, the completion of the current task, and the critical point of the current task. The detailed algorithm is described in Algorithm 1.

When a new job arrives, its expected utility density is calculated based on (2), (5), and (9). If its expected utility density is larger than the pre-set threshold, it is accepted and is rejected otherwise. When the current running task completes, the task in the ready queue with the highest expected system utility density is chosen to be executed. When the time reaches the critical point of the current running task, it implies that it will mostly likely incur utility density less than the threshold and is thus worthless of continuous execution. In that case, the task is immediately discarded, and a new task will be chosen to execute. At every scheduling point, the expected utility density of the tasks in the ready queue is checked. Since the expected utility density decreases monotonically with time, the task with expected utility density less than the threshold is aborted. The complexity of Algorithm 1 comes from the calculation of the expected system utility values for the task set, with the complexity of $O(n^2)$, where n is the number of tasks in the ready queue.

3.2. The Speculation-Based Utility Metric. From (3), (4), and (5), we can clearly see that the expected utility of running a task depends heavily on variable T, that is, the time when the task can start. If we can know the execution order and thus the expected starting time for tasks in the ready queue, we will be able to quantify the expected utility density of each task more accurately. In this section, we develop our second utility metric based on a speculated execution order of the tasks in the ready queue.

The general idea to generate the speculated execution order is as follows. We first calculate the expected utility density for each task in the ready queue based on the expected finishing time of the current running task. Then the task with the largest one is assumed to be the first task that will be executed after the current task is finished. Based on this assumption, we then calculate the expected utility densities for the rest of the tasks in the ready queue and select the next task. This process continues until all tasks in the ready queue are put in the order. While completed, we essentially generate a speculated execution order for the tasks in the ready queue and, at the same time, calculate the corresponding expected utility density for each task. The detailed algorithm is described in Algorithm 2.

The scheduling algorithm based on our speculated utility metric is very similar to Algorithm 1 and is thus omitted. The only difference is that the speculation-expected utility, rather than the opportunity cost-based utility, for each task in the

1: **Input**: Let $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_k\}$ be the accepted tasks in the ready queue, and let r_i, $\overline{C_i}$ represent the arrival time and expected execution time of τ_i. Let the current time be t
2: **Output**: The new list $\Gamma' = \{\tau_1', \tau_2', \ldots, \tau_k'\}$ with the speculated execution order and their corresponding expected utility density $\widehat{\rho}_j'$ for τ_j', $1 \le j \le k$.
3: **If** A task τ_0 is being executed **then**
4: $T = r_0 + \overline{C_0}$;
5: **else**
6: $T = t$;
7: **end if**
8: **While** Γ is not empty **do**
9: **For** Each task i in Γ **do**
10: Calculate $\overline{\rho}_i(T)$ based on (3), (4), (5), and (6);
11: **end for**
12: Select τ_j with the highest $\overline{\rho}_j(T)$;
13: Add τ_j to the end of Γ';
14 : $\widehat{\rho}_j = \overline{\rho_j}(T)$;
15: $T = T + \overline{C_j}$;
16: Remove τ_j from Γ;
17: **end while**

Algorithm 2: Generating the speculated execution order and the expected utility for task in the ready queue.

ready queue is calculated at each scheduling point, including the arrival of a new task, the completion of the current task, and the critical point of the current task.

The complexity of the scheduling algorithm mainly comes from Algorithm 2. It is not difficult to see that the complexity of Algorithm 2 is $O(n^2)$ with n the number of tasks in the ready queue.

4. Preemptive Approaches

In the previous section, we introduce two methods to quantify the potential system utility when scheduling a set of real-time requests *nonpreemptively*. Since a preemptive real-time scheduling technique tends to be more responsive for a higher priority request and can achieve higher schedulability and throughput than its nonpreemptive counterpart, we are interested in studying how to schedule a real-time task set preemptively to maximize the total accrued system utility.

When employing the preemptive scheduling method to schedule real-time tasks with the goal of maximizing the accrued utility, a critical issue is to determine when the preemption should occur. An intuitive approach is to define the priority of a task based on its expected utility density (6). Nevertheless, such an unconstrained preemptive scheduling may or may not improve the system performance, in terms of accrued system utility, when compared to a nonpreemptive one.

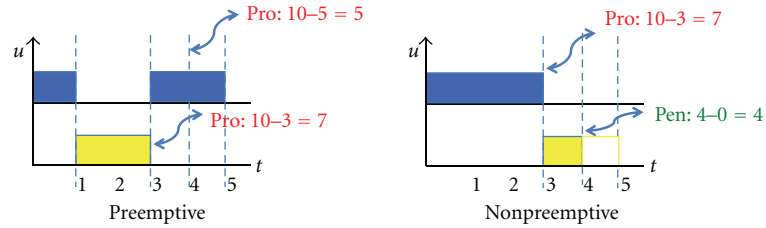
Consider the two examples in Figure 4. Figure 4(a) shows two tasks scheduled both preemptively (based on the expected utility density) and nonpreemptively. The parameters for both tasks are listed in the figure. In preemptive method, task τ_1 arrives and starts its execution at arrival time $t = 0$. At time $t = 1$ task τ_2 arrives. Note that, at $t = 1$, we have $\rho_1(t) = 1.3$ and $\rho_2(t) = 1.6$. Therefore, τ_2 comes with a higher expected utility density and preempts τ_1. Task τ_1 continues its execution after task τ_2 completes. The total utility in this method is 12. In the corresponding nonpreemptive method, task τ_2 misses its deadline, and the total utility in this method is 3. This example shows that, by processing the higher "*priority*" job first, the preemption helps increase the total utility of the system.

Now let us consider the example in Figure 4(b). For the two tasks in Figure 4(b), at time $t = 1$, we have $\rho_1(t) = 1.6$ and $\rho_2(t) = 2.3$. Therefore, τ_2 has a higher priority than τ_1. When these two tasks are scheduled in the preemptive manner, task τ_1 misses its deadline, and the total utility is 3. Both two tasks can meet their deadlines when they are scheduled in the nonpreemptive manner with a total utility of 12.

This example illustrates that unconstrained preemption does not always help improve the accrued utility. Note that since the profit and penalty TUFs of each task vary with time, its "priority" also varies with time. In this case, all tasks in the ready queue need to be checked for priority at every time instance. Hence, a perfect preemptive scheduling would be impractical due to its prohibit computational cost, even if it is theoretically possible. In addition, a large number of unconstrained preemptions disrupts task executions, makes them less likely to complete before their deadlines, and leaves alone the large overhead coming with the preemptions. Our empirical studies also showed that unconstrained preemptive scheduling can potentially degrade the performance than the corresponding nonpreemptive scheduling. To this end, we want to limit the scenarios of when the preemption can occur to improve the performance of the preemptive scheduling.

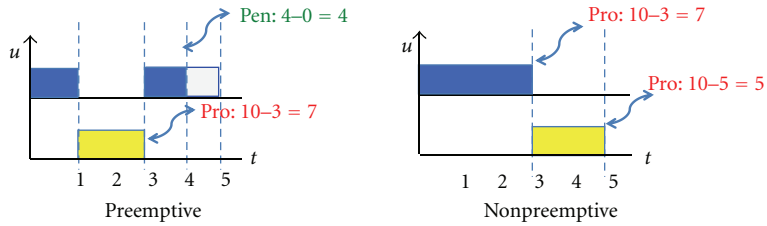
To constrain the preemptions, we first limit the time instances that at when preemptions can occur. Instead of letting a higher priority task always preempts a lower priority task, we allow that such a preemption can only happen when a new task comes or at a regular checking point, which we call

	Arrival	Exec	Best	Worst	G	L	D
Task 1	0	3	2	7	10–t	t–0	6
Task 2	1	2	1	5	10–t	t–1	4



(a) Preemptive scheduling is better than the nonpreemptive scheduling

	Arrival	Exec	Best	Worst	G	L	D
Task 1	0	3	2	5	10–t	t–0	4
Task 2	1	2	1	5	10–t	t–1	6



(b) Nonpreemptive scheduling is better than the preemptive scheduling

Figure 4: Preemptive versus nonpreemptive scheduling two real-time requests to maximize the accrued system utility.

preemption checking point. Let the last preemption occurs at time $t = T_0$. A task can be preempted at $t = T$ only if new tasks arrive at $t = T$ or

$$(T - T_0) \bmod L_{\text{int}} = 0, \tag{10}$$

where L_{int} is the length of the *preemption checking point* interval.

At a *preemption checking point*, the higher priority task does not necessarily always preempt the one with lower priority if the potential gain to execute the high priority task is not significantly higher than the gain achieved by continuously executing the current running task. We define a parameter called *preemption threshold* for this purpose. Let the current running task τ_0's conditional expected accrued utility density be $\hat{\rho}_0(\tau_0, t)$ at time t, and preempting task τ_p's expected accrued utility density be $\overline{\rho_p}$. Task τ_p preempts τ_0 only when the following equation is satisfied:

$$\overline{\rho_p}\left(\tau_p\right) - \hat{\rho}_0(\tau_0, t) > \zeta, \tag{11}$$

where ζ is the *preemption threshold*.

To further constrain preemptions, we do not allow the current task be preempted if it can complete by its deadline even it requires its worst case execution time. Preempting such tasks can delay the completion of these tasks, and potentially turn the profit into penalty if these tasks miss their deadlines. This constraint is illustrated by (12).

$$S_{\tau_0} + WE_{\tau_0} \le D_{\tau_0}, \tag{12}$$

where S_{τ_0} is the starting time of current running task τ_0. WE_{τ_0} means the worst case execution time of τ_0. D_{τ_0} represents τ_0's deadline.

We summarize our preemption rules and present the preemptive scheduling algorithm in Algorithm 3.

From Algorithm 3, when a preemption checking point is reached or when a new task arrives, scheduler first compares the preempting task's expected utility density $\overline{\rho_p}(\overline{C_p})$ with the current running task's conditional expected utility density $\hat{\rho}_0(\tau_0, t)$. If the preempting task's expected utility density exceeds the current running task's conditional expected utility density by a *preemption threshold*, then the scheduler further checks if the current running task can complete its execution in its worst case or not. If the current running task can be completed even in its worst case, no preemption is allowed in order to protect the current running task, since this current running task will absolutely contribute positive utility to the system. Otherwise, the preemption may postpone the current running task and result in a penalty because of missing its deadline.

1: **input:** Let τ_0 be the task currently being executed, and τ_p be the task wants to preempt τ_0, current time be t, $\hat{\rho}_0(\tau_0, t)$ be the conditional expected utility density of τ_0 at time t, $\overline{\rho_p}(\overline{C_p})$ be the expected utility density of τ_p, $\overline{C_p}$ and $\overline{C_0}$ are the expected execution time of τ_p and τ_0, respectively;
2:
3:When a new task arrives or it is the preemption checking point
4: **If** $\overline{\rho_p}(\overline{C_p}) - \hat{\rho}_0(\tau_0, t) > \zeta$ **then**
5: Check what is τ_0's worst case finish time;
6: **If** $S_{\tau_0} + WE_{\tau_0} \leq D_{\tau_0}$ **then**
7: Preemption not allowed;
8: **else**
9: Preemption allowed;
10: **end if**
11: **end if**

Algorithm 3: Preemption checking.

5. Experiments

In this section, we use experiments to investigate the performance of our proposed algorithms. The following six representative scheduling approaches were implemented and compared:

(i) EDF: the execution order of tasks are determined based on the EDF scheduling policy;

(ii) GUS [8]: the execution order of tasks is determined by the expected utility density, or the accrued utility per unit time;

(iii) PP: this is a previous approach developed based on a metric called *Risk Factor* [19]. It adopts similar system models as those used in this paper;

(iv) RR: the risk/reward approach described in [18]. This is a utility accrual approach that allows the utility value to be negative (e.g., similar to Figure 1(b));

(v) PPOC: this is the scheduling approach (i.e., Algorithm 1) built upon the utility metric that is developed based on the opportunity cost;

(vi) PPS: this is the scheduling approach built upon the speculated utility-based metric as discussed in Section 3.2.

5.1. Experiment Setup. The test cases in our experiments were randomly generated. Specifically, each task $\tau = ([B, W], f(T), G(t), L(t), D)$ was randomly generated as below.

(i) B, W, and D were randomly generated such that they are uniformly distributed within interval of $[1, 10]$, $[30, 50]$, and $[40, 50]$, respectively.

(ii) The execution time of a task is assumed to be evenly distributed between interval of $[B, W]$, that is, $f(t) = 1/(W - B)$.

(iii) G and L were assumed to be linear functions, that is, $G(t) = a_g(-t + D)$ in the range of $[0, D]$ and $L(t) = a_l t$. The gradient for $G(t)$ and $L(t)$, that is, a_g and a_l were randomly picked from the interval of $[4, 10]$ and $[1, 5]$, respectively.

(iv) Task release times follow the Poisson distribution with $\mu = 1$.

(v) Preemption check interval length L_{int} is set to be 1.

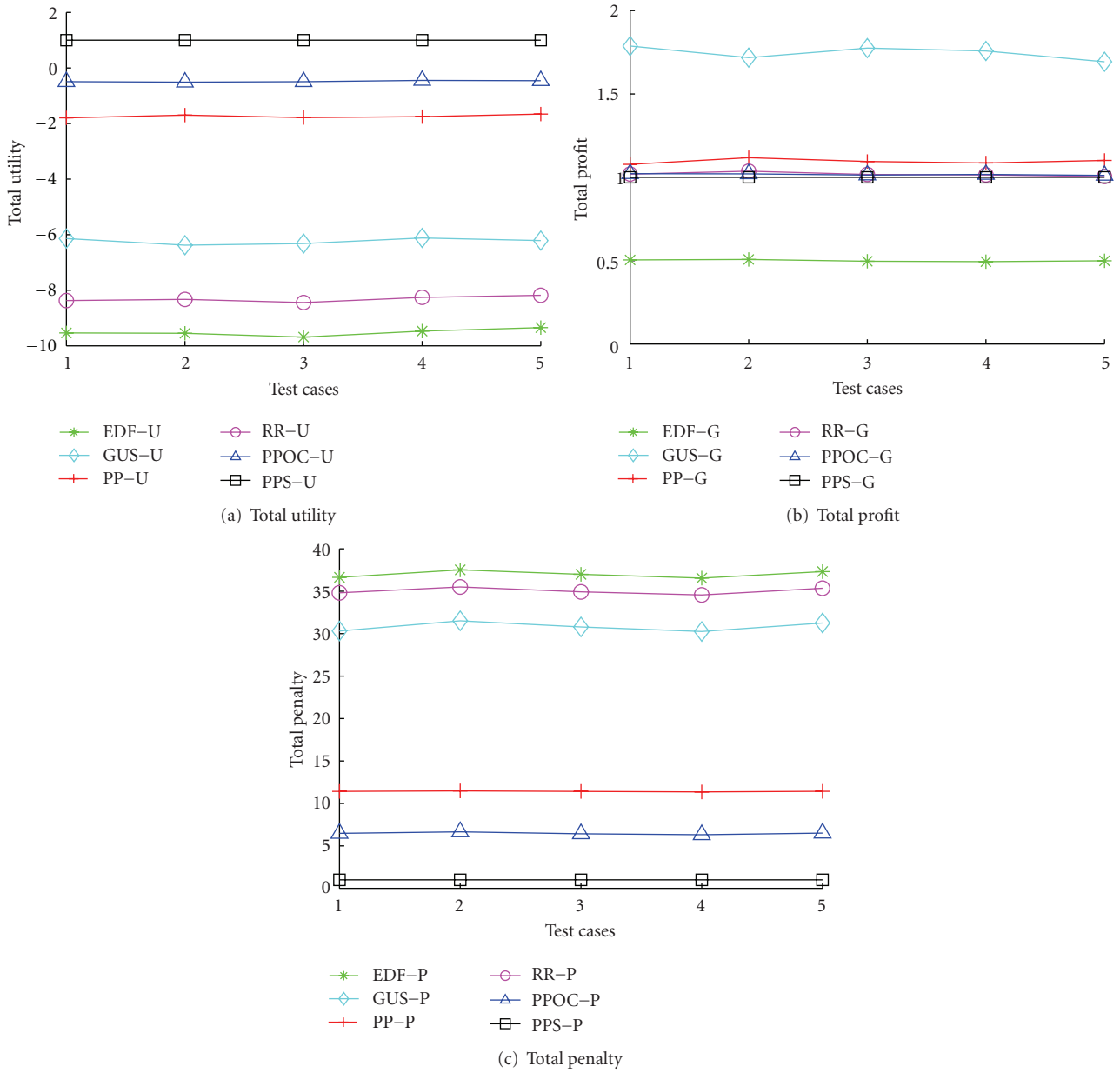
(vi) Preemption threshold ζ is set to be 0.

(vii) The utility density threshold δ is set to 0.

We conducted several different groups of experiments to study and compare the performance of different approaches under different conditions. The results are reported as follows.

5.2. Overall Performance Comparison. We first constructed 5 groups of experiments to study the overall performance of our proposed nonpreemptive scheduling algorithms. Each group has 1000 task sets, each of which consists of 20 tasks. The six different nonpreemptive scheduling algorithms were applied to the same task sets. The overall utility, the total profit, and the total penalty by each scheduling approach were collected and plotted in Figures 5(a), 5(b), and 5(c), respectively. For ease of presentation, the experimental results are normalized to that by PPS.

Figure 5(a) clearly shows that both PPOC and PPS can significantly outperform the other scheduling approaches. It is not surprising that, from Figure 5(c), we can see that the penalty-conscious approaches, that is, PP, PPOC, and PPS, are more effective to control the penalty than the other three, that is, EDF, GUS, and RR. PPOC and PPS are particular effective in penalty control. It is interesting to note from Figures 5(b) and 5(c) that, while the profits obtained by PPOC and PPS are comparable or even inferior to the other approaches, the penalties are dramatically decreased. This is because tasks that would potentially lead to high penalty are declined or discarded at early stages of their execution. As a result, the overall utilities are significantly higher than other approaches. As shown in Figure 5(a), with more elaborate scheduling algorithms to formulate the expected utility more accurately, and thus to make more appropriate decisions in



(a) Total utility

(b) Total profit

(c) Total penalty

Figure 5: The comparison of total utility, profit, and penalty by different nonpreemptive scheduling approaches.

task acceptation, abortion, and discard, PPOC improve upon PP by more than 70%, and PPS improve upon PP by more than 120% on average.

When comparing PPOC nonpreemptive and PPS nonpreemptive, we can see from Figure 5(a) that PPS is slightly better than PPOC. We can tell that the speculation-based utility metric predominant the opportunity cost metric in the control of penalty. The speculation order plays the major role in predicting the high risk of penalty.

5.3. Arrival Burst Impacts. We next studied the performance of our nonpreemptive scheduling methods under different burst conditions. In this experiment, we set the number of tasks to 20 and varied the expected number of occurrences within a unit interval, μ, from 1 to 5. By changing μ, we essentially changed the interval length between task arrivals. Figure 6 shows the results of the 1000 task sets' total utility with different values of μ achieved by the nonpreemptive scheduling algorithms.

When μ increases from 1 to 5, the number of task that comes within the same length of interval decreases, so the ready queue becomes less crowded and the overall workload reduces. The reduction in workload also helps lower down the deadline miss rate. More tasks can contribute positive profits instead of negative penalties to the system. Therefore, the total accrued utility is improved. From Figure 6, we can

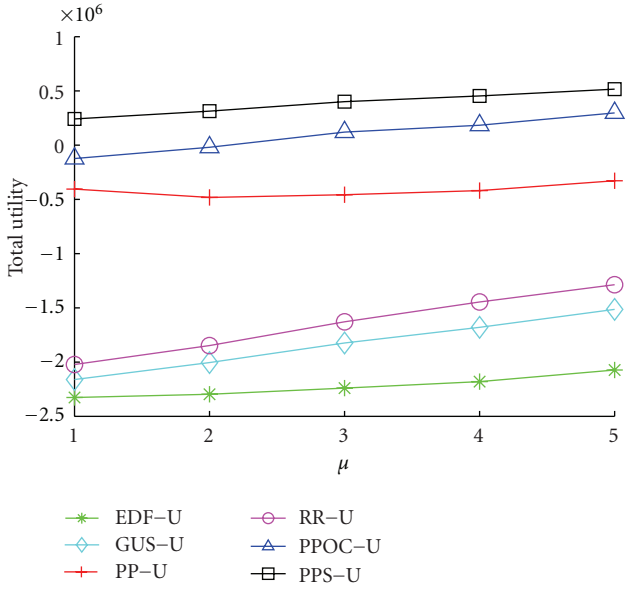


Figure 6: The total utility with different μ from nonpreemptive scheduling algorithms.

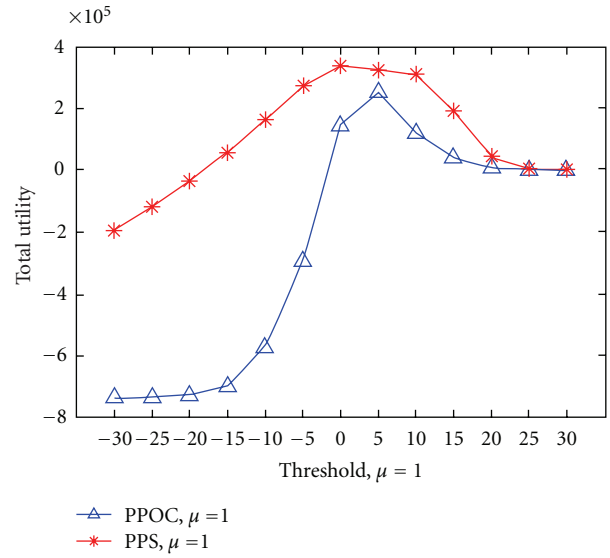


Figure 7: The total utility varies with the threshold.

see that all the methods have a better performance as μ increases, and PPS and PPOC significantly outperform the other approaches.

5.4. Utility Density Threshold Effect. We further studied the impacts of the utility density threshold δ on the scheduling performance. As indicated in Section 3, the threshold δ plays an important role in task admission, abortion, and execution. The larger the threshold, the smaller the number of tasks can be accepted and executed, and the smaller the penalty the system will suffer. To study this impact, we conducted another set of experiments. We generated task sets as before but changed the threshold from -30 to 30, with an interval of 5. The total utilities were collected and shown in Figure 7. It shows the effects on the 1000 task sets' total utilities at various threshold values.

It is interesting to see that the highest utility does not always occur at the point when the threshold equals zero. With the help from the figure, we can tell that the highest utility seldom occurs at the point with the lowest or the highest threshold value. The lower the threshold, the more tasks can be accepted to the system and get executed. This helps to improve the value of potential total profit. However, having more tasks accepted into a ready queue may potentially increase the potential penalty cost as many jobs can not meet their deadlines. On the contrary, using a higher threshold helps control the potential penalty but may limit the potential total profit that can be obtained. As a result, the total utility is a tradeoff between the two as shown in Figure 7. From Figure 7 we can see the significant impact that the different threshold values may have on the overall performance. How to choose an appropriate threshold value for a specific task set to strike the balance between the profit and penalty and hence achieve the optimal accrued utility is an interesting problem and needs further study.

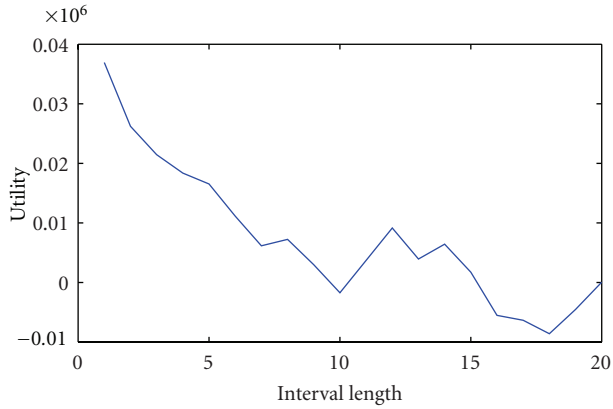


Figure 8: Preemption checking interval effect with $\lambda = 1$, task number = 20 each group, and $\zeta = 0$.

5.5. Effects of Preemption-Checking Interval Length L_{int} and Preemption Threshold ζ. In order to design a proper preemptive approach, we studied the preemption effects that come from variables L_{int} and ζ. We want to avoid aggressive preemptions. In Algorithm 3, when there is a high-priority task that wants to preempt the current running task, our scheduler first tries to protect the current running task and guarantee the current running task to finish execution without being preempted. The first constraint we added on preemptions is the preemption checking interval. In Figure 8, the result shows the effect of L_{int}. Even though there are bumps in the figure, it demonstrates a major trend that the smaller the preemption checking interval, the higher the system utility.

A preemption threshold ζ is another preemption constraint. Its effect is reflected by Figure 9. An optimal preemption threshold for a special task set can be hard to find. As shown in this figure, similar to the utility density threshold

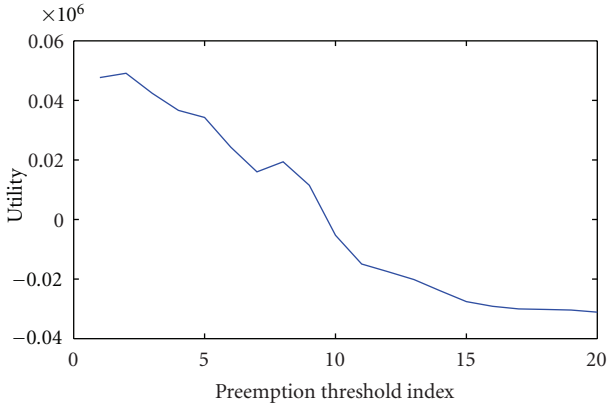


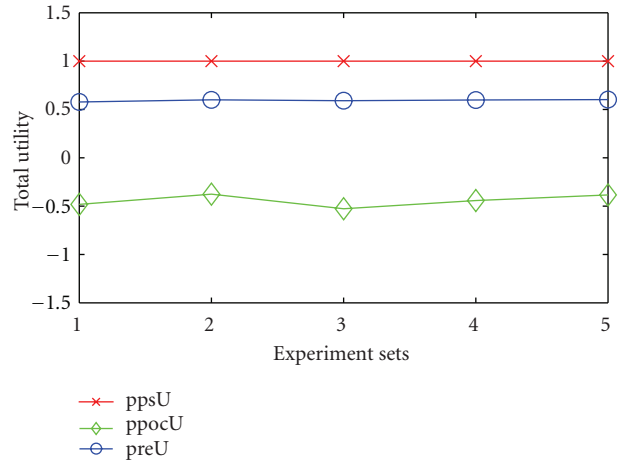
FIGURE 9: Preemption threshold effect with $\lambda = 1$, task number = 20, and $L_{\text{int}} = 1$.

(δ) effect, the optimal value is seldom achieved at the two extremes. For this particular data set we tested, the best preemption threshold value ζ is around 2.

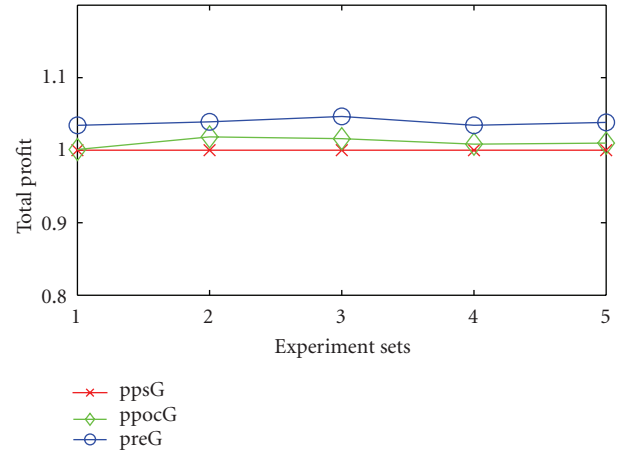
High preemption number does not mean better performance. Besides overheads generated by preemptions, potential penalties caused by preemptions may also be large. A set of carefully designed preemption rules can significantly improve the preemption performance. Results are shown in the next subsection.

5.6. Preemption versus Nonpreemption. Finally we compare our nonpreemptive and preemptive scheduling approaches. Figure 10 shows comparisons in details between nonpreemptive and preemptive scheduling approaches with the same task sets. It illustrates that PPS has the highest system utility, followed by the preemptive approach, then PPOC obtains the lowest system utility among these three approaches. Even though from Figure 10(b) we can tell that preemptive approach achieves the highest profit among them, it does not have a good control on penalty as PPS does. This results in a lower system utility in the preemptive approach than that in PPS. Nevertheless, Figure 10(b) illustrates the value of constrained preemptions for increasing system-accrued utility, since our preemptive scheduler always selects high-priority tasks to run at proper time.

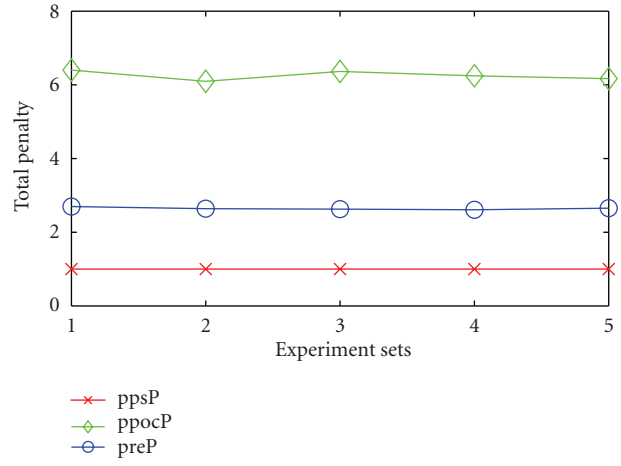
Figure 11 highlights the importance of preemption constraints. Some improper preemption instances postpone the running task's natural execution, in which a task may meets its deadline constraint without being preempted. In addition, it is hard to predict the future condition of the postponed task. Preemptions may help maximize the accrued system utility since the scheduler always runs high-priority tasks first, whereas whether to allow preemptions happen needs prudential measures. The observation from Figure 11 is that by applying constraints on preemptions, we successfully improve the performance of the preemptive scheduling approach.



(a) Comparison of total utility



(b) Comparison of total profit



(c) Comparison of total penalty

FIGURE 10: Comparison between PPS nonpreemptive and preemptive scheduling under the burstiness effect.

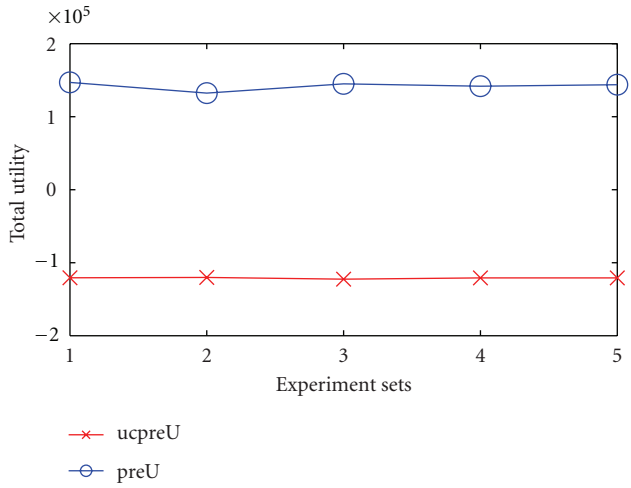


Figure 11: Comparison between constrained preemption approach and unconstrained preemption approach.

6. Conclusions

The popularity of Internet has grown enormously, which has presented a great opportunity for providing real-time services over Internet. Considering the tremendously large scale of the Internet infrastructure, it is necessary that not only the profit but also the cost of real-time task executions should be taken into consideration during the resource management process. Our experimental results clearly show that the traditional utility accrued approaches become ineffective.

In this paper, we first present two novel nonpreemptive utility accrued scheduling approaches upon a metric developed according to the *opportunity cost* concept and a speculation-based metric for expected utility, respectively. Then, a constrained preemptive approach is proposed. Our scheduling algorithms carefully choose highly profitable tasks to execute and aggressively remove tasks that potentially lead to large penalty. Our extensive experimental results clearly show that our proposed algorithms can significantly outperform the traditional EDF approach, the traditional utility accrued approaches, and an earlier heuristic approach based on a similar profit and penalty aware task model.

Acknowledgment

This work is supported in part by NSF under projects CNS-0969013, CNS-0917021, CNS-1018108, CNS-1018731, and CNS-0746643.

References

[1] M. Armbrust, A. Fox, R. Griffith et al., "Above the clouds: a berkeley view of cloud computing," *UC Berkeley*, 2009.

[2] E. Knorr and G. Gruman, "State of the internet operating system," 2010, http://radar.oreilly.com/.

[3] A. Weiss, "Computing in the clouds," *NetWorker*, vol. 11, no. 4, pp. 16–25, 2007.

[4] T. O'Reilly, "What cloud computing really means," *O'Reilly Radar*, 2010, http://www.infoworld.com/.

[5] R. K. Clark, *Scheduling dependent real-time activities*, Ph.D. dissertation, Carnegie Mellon University, 1990.

[6] C. D. Locke, *Best-effort decision making for real-time scheduling*, Ph.D. dissertation, Carnegie Mellon University, 1986.

[7] E. D. Jensen, C. D. Locke, and H. Tokuda, "A time-driven scheduling model for real-time systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, 1985.

[8] P. Li, *Utility accrual real-time scheduling: models and algorithms*, Ph.D. dissertation, Virginia Polytechnic Institute and State University, 2004.

[9] P. Li, H. Wu, B. Ravindran, and E. D. Jensen, "A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints," *IEEE Transactions on Computers*, vol. 55, no. 4, pp. 454–469, 2006.

[10] H. Wu, B. Ravindran, and E. D. Jensen, "Energy-efficient, utility accrual real-time scheduling under the unimodal arbitrary arrival model," in *Proceedings of the ACM Design, Automation and Test in Europe (DATE '05)*, pp. 474–479, March 2005.

[11] H. Wu, *Energy-efficient utility accrual real-time scheduling*, Ph.D. dissertation, Virginia Polytechnic Institute and State University, 2005.

[12] H. Wu, U. Balli, B. Ravindran, and E. D. Jensen, "Utility accrual real-time scheduling under variable cost functions," in *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 213–219, August 2005.

[13] F. Casati and M. Shan, "Definition, execution, analysis and optimization of composite e-service," *IEEE Data Engineering*, vol. 24, no. 1, pp. 29–34, 2001.

[14] H. Kuno, "Surveying the e-services technical landscape," in *Proceedings of the 2nd International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, 2000.

[15] Y. Bartal, S. Leonardi, A. Marchetti-Spaccamela, J. S. Gall, and L. Stougie, "Multiprocessor scheduling with rejection," in *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '96)*, pp. 95–103, 1996.

[16] B. N. Chun and D. E. Culler, "User-centric performance analysis of market-based cluster batch schedulers," in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, p. 30, 2002.

[17] F. I. Popovici and J. Wilkes, "Profitable services in an uncertain world," in *Proceedings of the ACM/IEEE Supercomputing Conference (SC '05)*, p. 36, November 2005.

[18] D. E. Irwin, L. E. Grit, and J. S. Chase, "Balancing risk and reward in a market-based task service," in *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pp. 160–169, June 2004.

[19] Y. Yu, S. Ren, N. Chen, and X. Wang, "Profit and penalty aware (pp-aware) scheduling for tasks with variable task execution time," in *Proceedings of the ACM Symposium on Applied Computing (SAC '10)*, 2010.

[20] Z. Bodie, R. Merton, and D. Cleeton, *Financial Economics*, Prentice Hall, New York, NY, USA, 2008.

[21] I. D. Baev, W. M. Meleis, and A. Eichenberger, "Algorithms for total weighted completion time scheduling," in *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '99)*, pp. S852–S853, January 1999.