On-line Scheduling of Real-time Services for Cloud Computing

Shuo Liu Gang Quan
Electrical and Computer Engineering Department
Florida International University
Miami, FL, 33174
{sliu005, gang.quan}@fiu.edu

Shangping Ren
Computer Science Department
Illinois Institute of Technology
Chicago, IL 60616
ren@iit.edu

Abstract

In this paper, we introduce a novel utility accrual scheduling algorithm for real-time cloud computing services. The real-time tasks are scheduled non-preemptively with the objective to maximize the total utility. The most unique characteristic of our approach is that, different from the traditional utility accrual approach that works under one single time utility function (TUF), we have two different TUFs—a profit TUF and a penalty TUF—associated with each task at the same time, to model the real-time applications for cloud computing that need not only to reward the early completions but also to penalize the abortions or deadline misses of real-time tasks. Our experimental results show that our proposed algorithm can significantly outperform the traditional scheduling algorithms such as the Earliest Deadline First (EDF), the traditional utility accrual scheduling algorithm and an early scheduling approach based on the similar model.

1 Introduction

Cloud computing has the potential to dramatically change the landscape of the current IT industry [1, 5, 10]. While there exist different interpretations and views on cloud computing [1, 5, 10], it is less disputable that being able to effectively exploit the computing resources in the clouds to provide computing service at different quality levels is essential to the success of cloud computing. For real-time applications and services, the timeliness is a major criterion in judging the quality of service. Due to the nature of the real-time applications over the Internet, the timeliness here refers to more than the deadline guarantee as that for the hard real-time systems. In this regard, an important performance metric for cloud computing can thus be the sum of certain value or utility that is accrued by processing all real-time service requests.

To improve the performance of cloud computing, one approach is to employ the traditional utility accrual (UA) approach [3, 9]. Jensen et al. first proposed to associate each task with a Time Utility Function (TUF), which indicates the task's importance [4]. Specifically, the TUF describes the value or utility accrued by a system at the time when a task is completed. Based on this model, there have been extensive research results published on the topic of UA scheduling [7, 8, 11, 12, 13, 14, 15]. While Jensen's definition of TUF allows the semantics of soft time constraints to be more precisely specified, all these variations of UA-aware scheduling algorithms imply that utility is accrued only when a task is successfully completed, and the aborted tasks neither increase nor decrease the accrued value or utility of the system.

We believe that, to improve the performance of cloud computing, it is important to not only measure the profit when completing a job in time, but also account for the penalty when a job is aborted or discarded. Note that, before a task is aborted or discarded, it consumes system sources including network bandwidth, storage space, and processing power, and thus can directly or indirectly affect the system performance. This is especially true for cloud computing in considering the large possibility of migration of a task within the clouds for reasons such as the economy considerations [2, 6]. If a job is deemed to miss its deadline with no positive semantic gain, a better choice should be one that can detect it and discard it as soon as possible.

Recently, Yu et al. [17] proposed a task model that considers both the profit and penalty that a system may incur when executing a task. According to this model, a task is associated with two different TUFs, a profit TUF and a penalty TUF. 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 it misses its deadline or is dropped before its deadline. It is tempting to use negative values for the penalties, and thus combine both TUFs into one single TUF. However, a task can be completed or aborted and hence can produce either a profit value or a penalty value. Mathematically, if there existed such a single function, it would imply that a single value in its domain was mapped to two values in its range, violating that it is a function. Therefore, one utility function cannot accurately represent both the profit and penalty information when executing a task. There are also some other penalty related models proposed in the literature. For example, Bartal et al. studied the on-line scheduling problem when penalties have to be paid for rejected jobs [16]. This model, however, does not account for the penalty to drop the task before its deadline.

In this paper, we present a novel utility accrual, non-preemptive scheduling algorithm of real-time services based on a task model similar to the profit and penalty model introduced by Yu et al. [17]. In addition to the careful choice of the ready task to run, our scheduling method judiciously discards pending requests and aborts task executions, and therefore can achieve better performance. Our experimental results also show that the proposed algorithm can significantly outperform the traditional scheduling approaches such as the Earliest Deadline First (EDF), the traditional utility accrual scheduling algorithm i.e. the Generic Benefit Scheduling (GBS) [7], and a previous scheduling approach based on the similar model, i.e. the Profit Penalty aware scheduling (PP-aware scheduling) [17].

The rest of the paper is organized as follows. Section 2 describes the models we used in the paper and formulate the problem formally. Section 3 presents our scheduling approach in details. Experiment results are discussed in Section 4 and we present the conclusions in Section 5.

2 Preliminary

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

- $[B_i, W_i]$: The best case execution time and the worst case execution time of τ_i;
- D_i: The relative deadline of τ_i;
- $f_i(T)$: The probability density function for the execution time of τ_i;
- $G_i(t)$: The profit TUF, which represents the profit accrued when a task is completed at time t. We assume $G_i(t)$ is a non-increasing unimodal function before its deadline, i.e. $G(t_i) \geq G(t_j)$ if $t_i \leq t_j$, and $G_i(t) = 0$ if $t \geq D_i$.
- $L_i(t)$: The penalty TUF, which represents the penalty suffered when a task is discarded at time t. We assume $L_i(t)$ is a non-decreasing unimodal function before its deadline, i.e. $L(t_i) \leq L(t_j)$ if $t_i \leq t_j$, 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 value varying with time. Therefore, while executing a task has a potential to gain profit, it also has a possibility to encounter a penalty at a later time. The system performance is therefore evaluated by its total utility gain after penalty is deducted. With the task model introduced as above, our problem can be formally formulated as follows.

Problem 1 *Given a task set $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ as described above, develop an on-line, non-preemptive scheduling method such that the accrued gain is maximized.*

3 On-line non-preemptive utility accrual scheduling

In this section, we present our on-line non-preemptive scheduling solution 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, dropping or aborting a task, and when to drop or abort a task. The rationale of our approach is very intuitive, i.e. a task can be accepted and executed only when it is statistically promising to bring positive gain, and discarded or aborted otherwise. Before we introduce the details of our scheduling approach, we first introduce two useful concepts, the *expected accrued utility* and the *critical point*.

3.1 The expected accrued utility and the critical point

Since the task execution time is not known deterministically, we do not know if executing the task will lead to positive gain or loss. To solve this problem, we can employ a metric, i.e. the *expected accrued utility*, to help us make the decision.

Given a task τ_i with arrival time of r_i, let its predicted starting time be T. Then the potential profit ($\overline{G_i}(T)$) to execute τ_i can be represented as

$$\overline{G_i}(T) = \int_{B_i}^{D_i-(T-r_i)} G_i(t + (T - r_i))f_i(t)dt. \tag{1}$$

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

$$\overline{L_i}(T) = L_i(D) \int_{D_i-(T-r_i)}^{W_i} f_i(t)dt. \tag{2}$$

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

$$\overline{U_i}(T) = \overline{G_i}(T) - \overline{L_i}(T). \tag{3}$$

A task can be accepted or chosen for execution when $\overline{U_i}(T) > 0$, which means that the probability of to obtain positive gain is no smaller than that to incur a loss. We can further limit the task acceptance by imposing a threshold (δ) to the expected accrued utility, i.e. a task is accepted or can be chosen for execution if

$$\overline{U_i}(T) \geq \delta. \tag{4}$$

We call δ as the *expected utility threshold*.

Furthermore, since the task execution time is not known a prior, we need to decide whether to continue or abort the execution of a task. The longer we execute the task, the closer we are to the completion point of the task. At the same time, however, the longer the task executes, the higher penalty the system has to endure if the task cannot meet its deadline. To determine the appropriate time to abort a task, we employ another metric, i.e. the *critical point*.

Let task τ_i starts its execution at T. Then the potential profit at $T' > T$ (i.e.$\tilde{G_i}(T')$) can be represented as

$$\tilde{G_i}(T') = \int_{T'-T}^{D_i-(T-r_i)} G_i(t + (T - r_i)) f_i(t) dt. \tag{5}$$

Similarly, the potential loss at $T' > T$ (i.e.$\tilde{L_i}(T')$) can be represented as

$$\tilde{L_i}(T') = L_i(D) \int_{D_i-(T-r_i)}^{W_i-T'} f_i(t) dt. \tag{6}$$

Therefore, the expected accrued utility at $T' > T$ (i.e.$\tilde{U_i}(T')$) can be represented as

$$\tilde{U_i}(T') = \tilde{G_i}(T') - \tilde{L_i}(T'). \tag{7}$$

We can make $\tilde{U_i}(t_0) = 0$ and solve for t_0. Then when executing task τ_i to time t_0, the expected profit equals its expected loss. We call t_0 as the *critical point* for executing task τ_i. Due to the non-increasing nature of G_i, $\tilde{U_i}(t)$ is monotonically decreasing as t increases. Therefore, it is not difficult to see that the continuous execution of τ_i beyond the critical point will more likely bring a loss rather than a positive gain.

3.2 The scheduling algorithm

Our 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.

Algorithm 1 ON-LINE NON-PREEMPTIVE ACCRUED-UTILITY SCHEDULING

```
1: Input: Let $\{\tau_1, \tau_2, ..., \tau_k\}$ be the accepted tasks in the ready queue ordered in non-increasing order of their expected accrued utility, and let $r_i, i = 1, ..., k$ represent their specific arrival times. Let current time be $t$ and let $\tau_0$ be the task currently being executed. Let the expected utility threshold be $\delta$.
2:
3: if $t$ = the critical time of $\tau_0$ then
4:     Abort the execution of $\tau_0$;
5:     Choose $\tau_1$ from the ready queue and start its execution;
6:     Re-calculate the expected accrued utility for each of the tasks in the ready queue;
7:     Remove the tasks with expected accrued utility smaller than $\delta$;
8: end if
9:
10: if $\tau_0$ is completed then
11:     Choose $\tau_1$ from the ready queue and start its execution;
12:     Re-calculate the expected accrued utility for each of the tasks in the ready queue;
13:     Remove the tasks with expected accrued utility smaller than $\delta$;
14: end if
15:
16: if a new job, i.e. $\tau_j$ arrives then
17:     insert it into the ready queue;
18:     Calculate the expected accrued utility of $\tau_j$ and also calculate the expected accrued utilities of tasks in the ready queue;
19:     Arrange the ready queue according to their expected accrued utility. Following this sequence, calculate the expected accrued utility for each task in the ready queue;
20:     for Each task in the ready queue do
21:         Remove the tasks with expected accrued utility smaller than $\delta$;
22:     end for
23: end if
```

We assume that tasks are inserted to the ready queue according to their expected accrued utility as they come. As shown in Algorithm 1, when the time reaches the critical point of the current task, the current active task is immediately discarded and the next task with the highest expected accrued utility is selected to be executed. Upon the finish of the current task, the task with the highest expected accrued utility is selected for execution. After the selection of the new task, the expected accrued utility for the rest of the tasks are re-calculated. The tasks with the expected accrued utility smaller than the threshold value are discarded.

Algorithm 2 THE CALCULATION OF THE EXPECTED GAIN

```
1: Input:Let $\{\tau_1, \tau_2, ..., \tau_k\}$ be the accepted tasks in the ready queue ordered in non-increasing order of their expected accrued utility, and let $r_i, i = 1, ..., k$ represent their specific arrival times. Let the current time be $t$ and $\tau_0$ be the task currently being executed
2: Output:The expected gain of $\tau_j, 1 \le j \le k$.
3: $T = t - r_j$;
4: for $i$ = 0 to $j - 1$ do
5:     $T$ = $T$ + expected finishing time of $\tau_i$;
6: end for
7: Calculate the expected accrued utility with $T$ based on equation 3.
```

When a new job comes, it is first inserted at the head of the ready queue, assuming its expected starting time would be the expected finishing time of the current active task. Based on this starting time, we then can compare its expected utility with the rest of the jobs in the queue. If its expected utility is less than that of the one following it, we re-insert this job to the queue according to its new expected utility. We then calculate the new expected utility according to Algorithm 2, by estimating its new expected starting time as the sum of the expected time of the leading tasks in the ready queue. This procedure continues until the entire ready queue becomes a listed ordered by their expected utilities. We then recalculate the expected utilities for the tasks behind, and remove the ones with expected utility lower than the threshold.

4 Experiment

In this section, we use experiments to investigate the performance of our proposed algorithm. Specifically, we first compare our algorithm with some traditional algorithms, including the non-preemptive EDF and a traditional utility accrual scheduling i.e. the Generic Benefit Scheduling (GBS) [7], as well as the Profit/penalty-aware non-preemptive scheduling proposed in [17]. The key of GBS is the metric called Potential Utility Density (or PUD), which determines the priority of a task based on the accrued utility per unit time. Profit/penalty-aware scheduling uses a heuristically defined parameter, called the risk factor to identify the importance of tasks. We also study how different utility thresholds may affect the performance of our algorithm.

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:

- B, W, and D were randomly generate such that they are uniformly distributed within interval of $[1, 10]$, $[30, 40]$, and $[50, 60]$, respectively;
- The execution time of a task is assume to be evenly distributed between interval of [B,W], i.e. $f(t) = \frac{1}{W-B}$
- G, L were assumed to be linear functions, i.e. $G(t) = -a_g(-t + D)$ in the range of $[0, D]$ and $L(t) = a_l t$. The gradient for $G(t)$ and $L(t)$, i.e. a_g and a_l were randomly picked from the interval of $[4, 10]$ and $[1, 5]$, respectively;
- Task release times follow the Poisson distribution with $\lambda = 5$;
- The utility threshold δ is set to 0.

We compared the four algorithms with a thousand task sets, each of which consists of ten tasks. Figure 1, Figure 2, and Figure 3 plot the accrued utility, accrued profit, as well as the accrued penalty for four different approaches: EDF (denoted as **edf**), the GUS algorithm(denoted as **pud**), the original profit/penalty-aware scheduling(denoted as **pp**) and our new algorithm (denoted as **ppnew**).

Figure 1 clearly shows that, by taking the penalty into consideration, both **ppnew** and **pp** can achieve much better performance than the traditional EDF or the utility accrued scheduling approach. Also, with an more elaborate scheduling algorithm that can make more appropriate decision in task acceptation, abortion, and discard, **ppnew** improves upon **pp** by more than 120% in average. It is interesting to note from Figure 2 and 3 that, while the accrued gain by **ppnew** is comparable or even inferior to the other approaches, the sum of penalty when executing the tasks are dramatically decreased. This is because the tasks that would potentially lead to high possibility of penalty are declined or discarded at the early stage of its execution by **ppnew**. This may reduce the number of tasks that can be completed close to their deadlines. On the other hand, however, it also significantly reduces the total penalty and, as a result, greatly increases the total utilities. Figure 1, 2 and 3 clearly demonstrate the effectiveness of our proposed algorithm.

Note that, the utility threshold plays an important role in task acceptation, abortion, and execution. The larger the

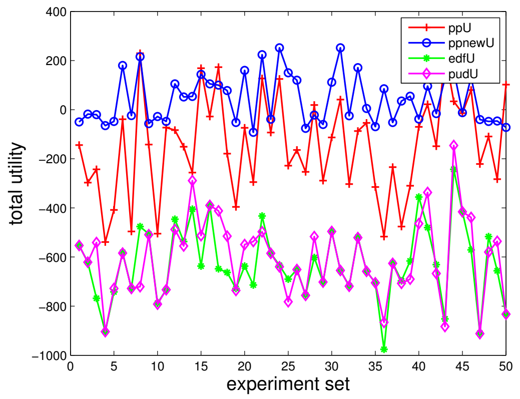

Figure 1. The comparison of total utility by four different approaches.

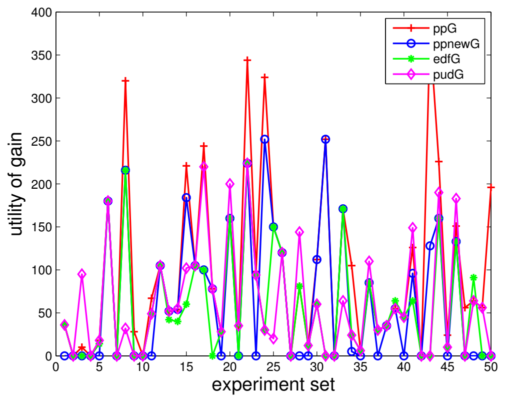

Figure 2. The comparison of total profit by four different approaches.

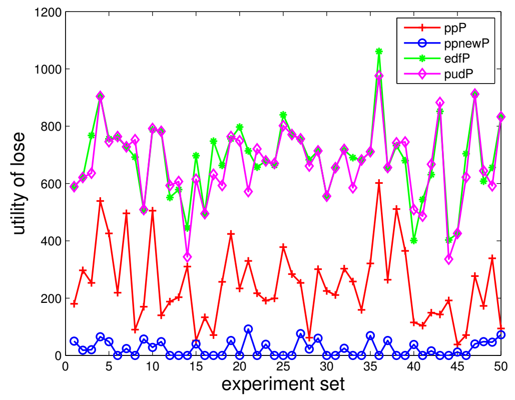

Figure 3. The comparison of total penalty by four different approaches.

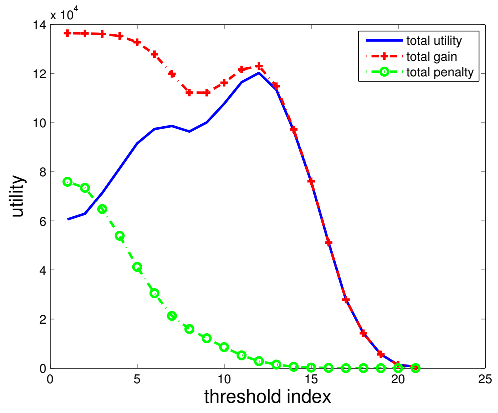

Figure 4. The total profit, penalty, and utility varies with the threshold.

threshold, the smaller the number of tasks can be accepted and executed, and smaller the penalty the system will suffer. To study the impacts of the utility threshold to the accrued utility, we conducted another set of experiments. We generated test cases as before, but changed the threshold from -100 to 300, with an interval of 20. The total profit, penalty, and utility of each task set is shown in Figure 4. For ease of presentation, we use the index number instead of the true utility threshold In Figure 4, i.e. $index = 1$ for threshold value of -100, and $index = 21$ for threshold value of 300.

It is interesting to see that the highest utility does not occur at the point when the threshold equals zero. Neither does it occurs at the point with threshold value equals 300. This is because the lower the threshold, the more tasks can be accepted to the system and get executed. On the contrary, the larger the threshold, the fewer tasks can enter the system and be executed. This is the reason we can see that in Figure 4 the total gain in general decreases as the threshold increases. On the other hand, few tasks in the system lead to smaller penalty. As a result, we can see from Figure 4 that the total penalty also decreases as the threshold increases. The total utility is a tradeoff between the two. From Figure 4 we can see the significant impact that the different threshold value may have for the overall performance. In addition, how to choose an appropriate threshold value to strike the balance between the profit and penalty and achieve the optimal accrued utility for the system is an interesting problem and needs further study.

5 Conclusion

Considering the tremendously large scale of the computing resource for cloud computing, it is necessary that not only the profit but also the cost of task executions should be taken into consideration during the resource management of these resource. The traditional utility accrued approaches become inadequate with the implication that utility is accrued only when a task is successfully completed, and the aborted tasks neither increase or decrease the accrued value or utility of the system. In this paper, we present a novel utility accrued approach which account for not only the gain by completing a real-time task in time but also the cost when discarding or aborting the task. Our scheduling algorithm carefully chooses the high profitable tasks to execute, and also aggressively removes the tasks that potentially lead to large penalty. Experimental results show that our proposed algorithm can significantly outperform the traditional EDF approach, the traditional utility accrued approach, and an earlier heuristic approach based on the similar profit/penalty task model. There are a quite a few interesting research problems for our future work, including striking a balance between the profit and penalty to achieve the optimal performance for the system, incorporating the preemption into our scheduling method, and incorporating more complicated time utility models.

Acknowledgement

This work is supported in part by NSF under projects CNS-0545913, CNS-0746643 and CNS-0917021.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. *UC Berkeley*, 2009.

[2] F. Casati and M. Shan. Definition, execution, analysis and optimization of composite e-service. *IEEE Data Engineering*, 2001.

[3] R. K. Clark. *Scheduling dependent real-time activities*. PhD thesis, Carnegie Mellon University, 1990.

[4] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time systems. In *IEEE Real-Time Systems Symposium*, 1985.

[5] E. Knorr and G. Gruman. What cloud computing really means. *http://www.infoworld.com*, 2010.

[6] H. Kuno. Surveying the e-services technical landscape. In *2nd International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*, 2000.

[7] P. Li. *Utility Accrual Real-Time Scheduling: Models and Algorithms*. PhD thesis, Virginia Polytechnic Institute and State University, 2004.

[8] P. Li, H. Wu, B. Ravindran, and E. Jensen. A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints. *Computers, IEEE Transactions on*, 55(4):454–469, April 2006.

[9] C. D. Locke. *Best-effort decision making for real-time scheduling*. PhD thesis, Carnegie Mellon University, 1986.

[10] A. Weiss. Computing in the clouds. *NetWorker*, 11(4):16–25, 2007.

[11] H. Wu. *Energy-Efficient utility Accrual Real-Time Scheduling*. PhD thesis, Virginia Polytechnic Institute and State University, 2005.

[12] H. Wu, U. Balli, B. Ravindran, and E. Jensen. Utility accrual real-time scheduling under variable cost functions. pages 213–219, Aug. 2005.

[13] H. Wu, B. Ravindran, and E. Jensen. On the joint utility accrual model. pages 124–, April 2004.

[14] H. Wu, B. Ravindran, and E. D. Jensen. Utility accrual scheduling under joint utility and resource constraints. pages 307–, May 2004.

[15] H. Wu, B. Ravindran, and E. D. Jensen. Energy-efficient, utility accrual real-time scheduling under the unimodal arbitrary arrival model. In *ACM Design, Automation, and Test in Europe (DATE)*, 2005.

[16] Y.Bartal, S. Leonardi, A. Marchetti-Spaccamela, J. S. gall, and L. Stougie. Multiprocessor scheduling with rejection. In *Proceedings of SODA*, pages 95 – 103, 1996.

[17] Y. Yu, S. Ren, N. Chen, and X. Wang. Profit and penalty aware (pp-aware) scheduling for tasks with variable task execution time. In *SAC2010 - Track on Real-Time System (RTS'2010)*.