Auto-Scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows

Ming Mao
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
ming@cs.virginia.edu

Marty Humphrey
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
humphrey@cs.virginia.edu

ABSTRACT

A goal in cloud computing is to allocate (and thus pay for) only those cloud resources that are truly needed. To date, cloud practitioners have pursued schedule-based (e.g., time-of-day) and rule-based mechanisms to attempt to automate this matching between computing requirements and computing resources. However, most of these “auto-scaling” mechanisms only support simple resource utilization indicators and do not specifically consider both user performance requirements and budget concerns. In this paper, we present an approach whereby the basic computing elements are virtual machines (VMs) of various sizes/costs, jobs are specified as workflows, users specify performance requirements by assigning (soft) deadlines to jobs, and the goal is to ensure all jobs are finished within their deadlines at minimum financial cost. We accomplish our goal by dynamically allocating/deallocating VMs and scheduling tasks on the most cost-efficient instances. We evaluate our approach in four representative cloud workload patterns and show cost savings from 9.8% to 40.4% compared to other approaches.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications

General Terms

Algorithms, Management, Performance, Economics.

Keywords

Cloud computing, auto-scaling, cost-minimization,

1. INTRODUCTION

Mapping performance requirements to the underlying resources in the cloud is challenging. Resource under-provisioning will inevitably hurt performance and resource over-provisioning can result in idle instances, thereby incurring unnecessary costs.

There are a number of complex goals that a cloud “auto-scaling” framework should achieve. Fundamentally, it should balance performance and cost. Of course, it should deliver at least the same level of performance as existing enterprise IT infrastructure. It must recognize and reflect the different options for computing resources – e.g., many cloud providers offer multiple types of virtual machines (VMs), each with different capabilities at a different price. For example, Amazon EC2 [1] currently offers 11 VM instance types, such as standard machines designed for most types of applications, high-CPU machines for compute intensive applications and high-memory machines for databases. An auto-scaling framework must recognize that these different instance types are often not priced linearly according to their processing power. Finally, it should not ignore the practical considerations of real cloud deployments, such as that additional computing capacity such as VMs can generally be acquired at any time but could take several minutes to be ready to use [2]. Also, it cannot ignore that current cloud practice is to charge computing resources by the hour, so it must not assume that resources should be immediately shut down to save money. All these factors make cloud auto-scaling a challenging problem to solve.

SC11, November 12-18, 2011, Seattle, Washington, USA

Currently, cloud providers and third party cloud services offer schedule-based and rule-based mechanisms to help users automatically scale up/down. Many of these are based on resource utilization, such as “Add 2 small instances when the average CPU is above 70% for more than 5 minutes.” The mechanisms are simple and convenient. However, it is not always straightforward for the users to select the “right” scaling indicators and thresholds, especially when the application models are complex, and the resource utilization indicators are limited and very low-level. In other words, these trigger mechanisms do not really solve the performance-resource mapping problem, and sometimes the resource utilization indicators are not expressive enough to address user performance requirements directly.

In our previous research [3], we explored the cloud auto-scaling problem with deadline and budget constraints for a batch-queue application model. We constrained the workload to be independent jobs with a uniform performance requirement. We used the integer programming technique to solve the problem. The results showed that our auto-scaling mechanism was effective in meeting deadlines. However, it was only applicable to a very limited problem set – neither precedence constraints among tasks nor individual performance requirements were supported. Additionally, it did not comprehensively address cost-efficiency.

In this paper, we explore the auto-scaling problem for a more general application model (jobs are expressed as workflows) and allow individual (non-uniform) job deadlines. The goal is to ensure that all jobs finish before their respective deadlines by using resources that cost the least amount of money. We note that deadlines are not “hard deadlines” as in hard real-time systems, whereby missing a deadline is defined as catastrophic failure. Rather, deadlines serve as the performance requirements specified by the users, and deadline misses are not strictly forbidden. We choose deadline as the performance requirement because we believe people generally care about the turnaround time of a service, such as Web request response time, network latency and a program’s running time. We use deadline assignment techniques to calculate an optimized resource plan for each job and determine the number of instances using the Load Vector idea. We address job scheduling and resource scaling at the same time by considering both the job-level and global-level cost-efficiency.

The test results show that our approach can save 9.8%-40.4% cost compared to other mechanisms.

The rest of the paper is organized as follows. Section 2 provides an overview of related work. Section 3 formalizes the problem and presents our solution. Section 4 contains a thorough evaluation of our methodology using different workload patterns on three representative application models, i.e., pipeline, parallel and hybrid. We also present evaluation results on handling imprecise input parameters, such as the estimation of task running time and the lag of instance acquisition. Finally, we conclude the paper and discuss future work in Section 5.

2. RELATED WORK

Related research can be generally grouped into four categories. The first category is dynamic resource provisioning in virtualized environments (e.g., [4][5][6][7][8][9][10][11]). These projects attempt to achieve application performance goals through dynamic resource provisioning in virtualized environments. They mainly focus on the use of control theory to tune application performance in a fine-grained manner. Several reactive and proactive workload prediction models are proposed and evaluated. More recently, people have extended the idea into the cloud environment, such as [12][13][14]. These projects try to handle increasing workload by acquiring VM instances in the cloud when the local computing capacity is insufficient. For example, Marshall [13] uses cloud instances to increase the compute power of a Torque cluster when submitted jobs cannot be finished in time. [14] proposes several instance acquisition policies and evaluates their performance under different workloads. These application models are limited to the batch-queue model and multi-tier Web applications. The main difference between these projects and the research described in this paper is that in general these projects are only focused on improving performance and do not address the financial costs of doing so.

The second category is workflow scheduling with deadline and budget constraints in a heterogeneous resource environment (e.g., [15][16][17][18][19]). These projects take a workflow DAG as input and assume there are different services available for each individual sub-task. They make sub-task scheduling decisions by trading off performance and cost in each step. Menasc and Casalicchio [15] use the ratio of performance-loss and cost-gain to determine the most cost-efficient service for a sub-task. Yu and Buyya [16][17][18] introduce deadline assignment ideas and use genetic algorithms to find optimal scheduling plans by defining the fitness function as the ratio of performance over cost. In [19], Sakellariou, et. al. form the minimum-duration execution plan first and then apply backtracking to refine the plan to meet budget constraints. While these projects directly address deadline and cost, they assume a resource limited environment and a single workflow instance execution instead of a more general stream of workload. Moreover, it is not clear to what extent their billing models can be applied in the cloud environment directly.

The third category is cost-efficiency in the cloud. On the cloud provider side, there are several papers [12][20][21][22][23] discussing resource allocation and instance consolidation strategies for cloud data centers. In general, the goal is to maximize cloud provider profits while still maintaining service level agreements (SLAs). On the cloud consumer side, the general issue is a cost comparison between the local enterprise and the cloud environment. Other research investigates using the cloud to increase local cluster capacity and application availability. For example, [24] addresses the cost of running eScience applications in the cloud, and [25] uses spot instances to accelerate MapReduce job execution with cheaper cost. Assuncao et al. [14] implement job scheduling strategies for a mixed local cluster and cloud environment. Other projects (e.g., [26][27]) build strategies to distribute an application among multiple cloud providers to enhance availability with minimum cost. None of these works deal with the automatic cloud resource provisioning problem in a dynamic workload context, and they do not consider application performance requirements and budget constraints.

The final category is cloud auto-scaling. Most cloud providers offer APIs to allow users to control their cloud VM instance numbers programmatically, thus facilitating user-defined auto-scaling. AWS [1] and some third-party cloud management services (RightScale [28], enStratus [29], Scalr [30], etc.) offer schedule-based and rule-based auto-scaling mechanisms. Schedule-based auto-scaling mechanisms allow users to add and remove capacity at a given time, such as "run 10 instances between 8AM to 6PM each day and 2 instances all the other time". Rule-based mechanisms allow users to define simple triggers by specifying instance scaling thresholds and actions, such as "add (remove) 2 instances when the average CPU utilization is above 70% (below 20%) for 5 minutes." These mechanisms are simple and convenient when users understand their application workload and when the relationship between the scaling indicator and the performance goal is easy to determine. However, not all applications have time-based workload patterns, and it is not straightforward for users to correctly determine all the related scaling indicators or the thresholds based on the performance goals. RightScale [28] and AzureScale project [31] provide additional support for some popular middleware performance metrics like Mysql connections, Apache http server requests, DNS queries and queue sizes. These supported scaling indicators make dynamic scaling easier for Web applications, but still they do not support deadlines or directly consider user cost.

3. PROBLEM FORMALIZATION

In this section, we enumerate our assumptions for both cloud applications and cloud resources, define the problem, and then detail our solution based on dynamic task scheduling and resource scaling.

3.1 Assumptions

We assume there are four roles: the cloud application owner, the cloud application users, the cloud application itself and the cloud resources. The application owner deploys her application in a cloud provider and offers services to the application users. The application users submit jobs with performance requirements (in the form of deadlines) to the application. The application owner purchases resources (e.g. VMs, storage, etc.) from the cloud provider to run the application and service all job requests. Her goal is to automate this resource provisioning process in the cloud and fulfill all job requests within their deadlines using the minimal amount of money through the auto-scaling mechanism. We make the following assumptions.

A. Cloud Application, Job & Task

- A cloud application consists of several functional service units and a job is composed of sub tasks (or simply referred to as tasks) with precedence constraints.
- All jobs are submitted into the entry unit. They flow from one service unit to another. Jobs are not required to follow the same processing route. In other words, different jobs may consist of different tasks. For example, for an online shopping Web site,

although both inventory query and order submission requests need to go through the customer authentication service, inventory queries only need to deal with the databases while order submission requests need to further go through the payment confirmation service and the receipt printing service.

- Jobs can be categorized into different classes based on their individual processing flows. Different job classes may have different levels of importance and urgency. User can specify different deadlines for different job classes.
- Every service unit can be considered as a queue that holds incoming tasks. Service units can be classified as compute-intensive, I/O-intensive, or neither. Thus, the tasks in each service unit may have different performance on different VM types.
- Application users can submit their jobs at any time and we do not assume the application owner has knowledge about the incoming workload in advance.

B. Cloud resources & pricing

- The cloud provider offers several types of VMs with different processing power (CPU, memory, disk etc.) for the application owner to choose. As per current real-world practice, VM instances are priced by hour. Partial-hour consumption is always rounded up to one hour. VM instances need not be priced linearly based on processing power.
- There could be different classes of tasks, such as compute-intensive and I/O-intensive tasks. This property is inherited from the service unit where it is processed. A task may have different processing times on different instance types. For example, a compute-intensive task can run faster on a high-CPU machine than on a standard machine. In other words, a task may prefer an expensive VM type (high hourly-cost) compared to a cheap VM type (low hourly-cost) when considering the greatly reduced processing time. Further, we assume the application owner can estimate the task processing times on different types of VMs.
- Cloud instances can be acquired at any time. However, it may take some time for an instance to be ready to use. We call such waiting time as instance acquisition lag. We assume acquired instances can be ready within the estimated acquisition lag time and all jobs can be finished within their deadlines if the fastest instances are always available. In other words, we do not consider jobs with theoretically impossible deadlines (too short to finish even on the fastest machines).

Figure 1 shows an example that reflects these assumptions. An insurance company offers online quotes to its customers, categorized as non-members, silver members and gold members. Non-member jobs consist of data validation and quote calculation using a very basic model, while silver and gold member jobs go through more data collection steps, such as the credit history and health record collection modules, and are evaluated based on more complex risk assessment models. The data collection steps are I/O-intensive tasks, and therefore run more cost-efficiently on high-memory instances, while the model building and premium calculation steps are compute-intensive tasks that run more cost-efficiently on high-CPU machines. There could be other job classes (not shown) such as daily and weekly report generation requested by the operation department, and data mining or sales trend analysis performed by the research department.

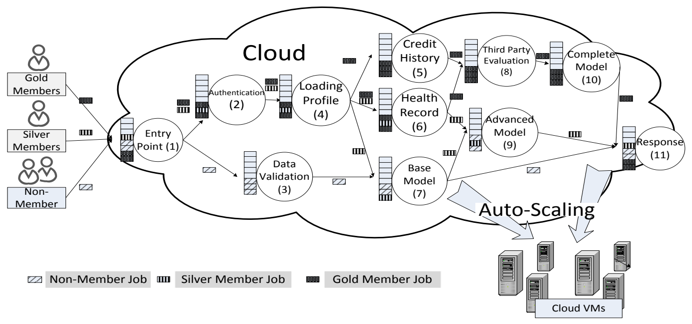

Figure 1. Cloud application model.

3.2 Problem Definition

We formally define a cloud application, cloud resources and the auto-scaling problem as follows.

3.2.1 Cloud Application

Definition 1 (Cloud Application). A cloud application consists of a set of service units. A service unit is an abstraction of a processing module/component in an application, such as the order submission and data persistence steps for an online shopping site, or data reduction and compression steps in a scientific workflow. All jobs are submitted into the entry service unit S_0.

$$app = \{S_i\}$$

Definition 2 (Job Class). A job class is a directed acyclic graph (DAG) with a deadline. In other words, a job is a set of tasks with precedence constraints and each task is processed at some service unit. The deadline is relative -- e.g. one hour, from the time a job instance is submitted to the time it is should be finished. Formally, a job class can be defined as

$$J = \{\mathrm{DAG}(S_i),\ deadline \mid S_i \in app\}$$

For example, the job class "non-member insurance quote" in Figure 1 can be defined as $\{\{(S_1,S_3)(S_3,S_7)(S_7,S_{11})\}, 1\text{min}\}$, which means the request should go through the entry point, data validation, basic model and response within 1 minute. In the application, there could be several concurrent job instances of the same job class. We use j_J to represent a job instance of job class J and $j_J^{S_i}$ to represent the task of j_J at service unit S_i.

3.2.2 Cloud Resources

Definition 3 (Cloud VM**).** The cloud provider may offer different types of VM instances, suitable for different types of workloads. These VMs have different processing power (the number of cores, memory, disk etc.) and prices (ranging from \$0.085/hour to \$2.10/hour for Amazon EC2). We define a VM type in terms of the estimated processing time for each service unit, the cost per unit time (currently it is 1 hour for cloud providers) and the estimated instance acquisition lag. We assume there is no performance inference among VMs.

$$\mathrm{VM}_v = \{[t_{S_i}]_v\,,\ c_v\,,\ lag_v\}$$

Performance is a vector, in which each element is the estimated processing time for tasks from service unit S_i. Such performance estimation on different types of VMs can be achieved by using existing performance estimation techniques [16] (e.g. analytical modeling [33], empirical [34] and historical data [35][36]). Furthermore, we know the cost of running task $j_J^{S_i}$on VM$_v$ is $t_{S_i} \times c_v$.

3.2.3 The auto-scaling problem

Definition 3 (Workload). We assume the application owner does not know the incoming requests in advance. Therefore, the workload is defined as all the jobs that have been submitted into the application. They are either waiting to be processed or partially processed. At some time *t*, we use the tasks waiting at every service unit to represent the workload *W*.

$$W_t = \sum_{S_i} \sum_J j_J^{S_i}$$

Definition 4 (Scheduling Plan). The auto-scaling mechanism needs to balance two decisions. The first decision, which we call "Scheduling Plan", is to determine the instance type for each running task at some time point *t*.

$$Schedule_t = \{ j_J^{S_i} \rightarrow \mathrm{VM}_v \}$$

Definition 5 (Scaling Plan). The second decision, which we call the "Scaling Plan", the auto-scaling mechanism needs to make is to determine the number of instances for each instance type VM_v at some time point *t*.

$$Scaling_t = \{\mathrm{VM}_v , N_v\}$$

Definition 6 (Goal). The goal is to find a scheduling plan and scaling plan that minimizes the total running cost *C*, in which the actual finishing time for each job *j* of class *J* is before the deadline. In other words, $\forall\, j_J$, $T(j_J) < Deadline(J)$.

$$\mathrm{Min}(C) = \mathrm{Min}(\textstyle\sum_v c_v N_v)$$

3.3 Solution

Unlike a more simplistic and static workflow scheduling situation that only considers a single workflow instance, there could be multiple job instances submitted to the application, and the workload is changing all the time. Therefore, the auto-scaling mechanism needs to keep monitoring the changing workload as well as the progress of submitted jobs, and then make fast scheduling/scaling responses. This is a repeated process instead of a one-time process. Therefore, our solution is based on a monitor-control loop. Every time inside the loop, a scheduling decision and a scaling decision are made based on updated information. Because cloud VMs are currently billed by instance hours (not by the exact user consumption time), the scheduling and scaling decisions should avoid partial instance-hour waste. Unlike a resource limited environment, as long as there are services available, the scheduling mechanism can reserve a time slot and place a task on it.

As we detail in the remainder of this section, we make the scheduling and scaling decisions step by step. We first calculate the number of VMs needed for each VM type based on the workload. Next, we determine if two or more existing VMs can/should be "consolidated". We then schedule tasks on each VM using the Earliest Deadline First (EDF) algorithm. Moreover, we also pre-analyze job classes for potential task bundling and deadline re-assignment to improve the runtime performance.

3.3.1 Preprocessing

To reduce the runtime overhead of the auto-scaling mechanism and accelerate dynamic scheduling/scaling plan generation, we pre-analyze job classes and calculate deadlines for each sub task using the following techniques.

Step 1 – Task Bundling. Task bundling treats adjacent tasks that prefer the same instance type as one task and forces them to run on the same instance. Therefore, it can save data transfers by using the temporary results stored locally. Figure 2 shows one example of task bundling. Both Task 6 and Task 8 run most cost-efficiently on high-CPU machines, so our mechanism will treat these two tasks as a single task Task 6'. In this work, we only bundle the tasks that prefer the same type of instances and have one-to-one task dependencies. The trade-off between large data movements and different VM type choices, as well as complex task dependencies, will be considered in our future research work.

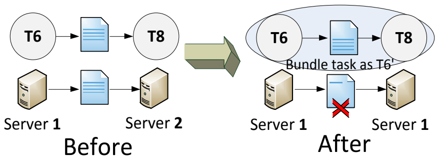

Figure 2. Task bundling.

Step 2 – Deadline Assignment. Deadlines are associated with jobs not tasks. When a job is submitted, we use the following mechanism to assign deadlines for each sub task. If every task can be finished by its assigned deadline, then the job will be finished within its deadline. Deadline assignment in a DAG is introduced and detailed in [16]. Their approach is to assign deadlines proportionally by tasks' fastest execution time and then search for the cheapest service for each task. In this paper, we assign individual deadlines to tasks proportionally based the processing time on their most cost-efficient machines. If such deadline assignment cannot make the job be finished within the deadline, we further use a heuristic introduced by [17] to locate an optimized plan. The idea is to calculate the job makespan for the initial deadline assignment, and if the job can be finished within the deadline, the process stops. If not, we try to schedule each task on a faster but more expensive machine and calculate the new job makespan to see which one reduces the makespan the most with the same amount of money. In other words, we upgrade the task with the highest cost-efficiency rank to a faster machine. We repeat such process until the job makespan is within the deadline. Through the deadline assignment process, we break the task precedence constraints and treat each task independently. Figure 3. shows one example of deadline assignment.

$$rank = \frac{makespan_{before} - makespan_{after}}{cost_{after} - cost_{before}}$$

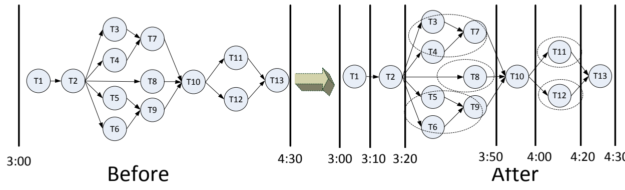

Figure 3. Deadline assignment.

In this step, we can explore some other techniques to further tailor our approach to the cloud. Unlike the utility computing environment [15][16][17], as long as there are services available, the scheduling mechanism can reserve a time slot and place the task on the service. In the cloud, although there are unlimited resources and we can acquire an instance at any time, however, it may not be always good to do so whenever a task needs to be processed, especially when the task could waste a significant portion of a purchased instance-hour. Therefore, reducing task concurrency is a way to improve instance utilization rate. Figure 4 illustrates this idea. Assume tasks T2, T3 and T4 are tasks shorter than one hour. By processing T2 – T8 sequentially instead of in parallel, three instance hours can be saved. We use breadth-first search to combine parallel tasks, and search until some task has to change its originally scheduled machine type to finish before deadline. We stop searching because this strategy is only job-wide optimized, and we do not want to affect the global scheduling decision. The overall algorithm is shown below.

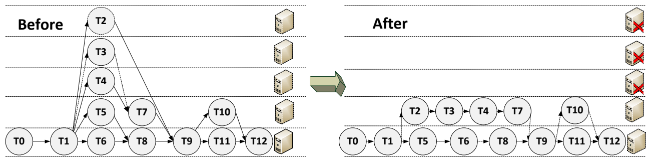

Figure 4. Parallelism reduction.

Algorithm 1 – Deadline Assignment

Input – A DAG of a job, performance estimation of $task_i$ on each VM_m price of each VM_m type c_m

Output – Schedule plan $S = \{ task_i \rightarrow VM_m\}$

```
1:  Generate the cheapest schedule $S=\{task_i \rightarrow$cost-efficient $VM_m\}$ [16]
2:  While (true)
3:     For (parallel tasks that share the same parents and children)
4:        Combine them into sequential tasks in DAG, re-compute $S$
5:        If ($S$ is changed)
6:           Next
7:        End If
8:     End For
9:     Re-compute $S$
10:    Break
11: End While
12: While(true)
13: If ($makespan(S) < deadline$)
14:          return $S$
15:    Else
16:       For (each $task_i$)
17:          $S_i = S - (task_i \rightarrow VM_m) + (task_i \rightarrow nextFasterVM(task_i))$
18:          $SpeedUp_i = (makespan(S) - makespan(S_i))/(\text{cost}(S_i)-\text{cost}(S))$
19:       End For
20:       $index = subscript(max(SpeedUp_i))$
21:       $S = S_{index}$
22:    End If
23: End While
24: return $S$
```

3.3.2 *Dynamic scaling-consolidation-scheduling*

Inside each monitor-control loop, we make dynamic scheduling and scaling decisions using the most recent information. We recalculate task deadlines to determine the instance number, consolidate partial instance-hours and schedule tasks using EDF.

Step 3 – Scaling.

Definition 7 (Load Vector). A load vector (LV) is defined for each task. After deadline assignment, an execution interval $[T_0 , T_1]$ is scheduled for each task, and we know its running time on VM_m is t_m. Therefore, LV_m is defined as $[t_m/(T_1 - T_0)]$ indexed from T_0 to T_1. Intuitively, the vector is the number of the machines needed to finish the task on VM_m. Because a task cannot be arbitrarily divided into pieces and run in parallel, if the ratio is greater than 1, the task cannot be finished in time. For example, as shown in Figure 5, assume the execution interval for T_1 is from 3:00PM to 4:00PM, and it is estimated to run for 15 min on VM_1. We say that the task needs to use 1/4 instance-hours on VM_1 between 3:00PM and 4:00PM. Another Task T_2 needs to be finished between 3:15PM and 3:45PM and also consumes 15 min on VM_1. Therefore, we know that we can use one instance to process both T_1 and T_2. The finest granularity we use is 1 minute.

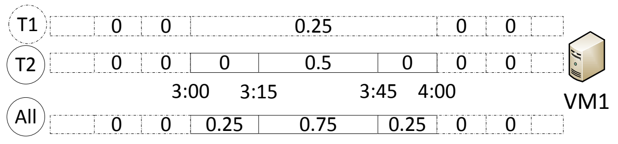

Figure 5. Load vector.

We calculate the load vectors for each task and add them to get m load vectors, one for each machine type. If we can ensure that the number of existing machines is always greater than or equal to the load vector at any time, the tasks will finish within the assigned execution interval. Because cloud VMs may take some time to be ready to use, for scaling-up cases, task load vectors are calculated between interval $[T_0+ lag_v, T_1]$ instead of $[T_0, T_1]$. Scaling-down decisions are also based on the load vector. The auto-scaling mechanism is aware of the acquisition time of every instance. When one instance is approaching multiples of instance-hour operation and the number of instances is greater than the load vector, we can shut down the instance. VM churn is another factor which may affect the decision of shutting down an instance. Too frequent VM acquisition/release will degrade the performance of the auto-scaling mechanism if the instance acquisition lag cannot be accurately estimated. We show the effects of imprecise estimations in the evaluation section. Note, every time when updating the task load vectors, deadline assignments need to be recalculated, because some tasks may be finished earlier than their original assigned deadlines and reassignment can possibly allow later tasks to run on cheaper machines.

Step 4 – Instance consolidation. It is optimal if all tasks can be executed on their most cost-efficient instances and all instances are fully utilized. However, it may not be always feasible to make sure there are no wasted partial instance hours when considering the arrival times and execution times of the tasks. Sometimes, we need to make a decision to run tasks on their non-cost-efficient machines to consolidate partial instance hours, because consolidating instance hours can help users spend less money on cloud resources. We call this process *instance consolidation*. The figure below illustrates this idea. T11 and T12 originally are scheduled on a high-CPU instance and a standard instance, respectively. Because both only consumes a partial instance-hour and there are no other tasks sharing the instances in the same instance hour, a smart scheduling decision is to consolidate the two tasks on the same standard machine and save one high-CPU instance hour (although T11 runs slower and costs more on a standard machine). Of course, a "consolidated task" must still be finished by its original deadline. The process is described in the following pseudo code.

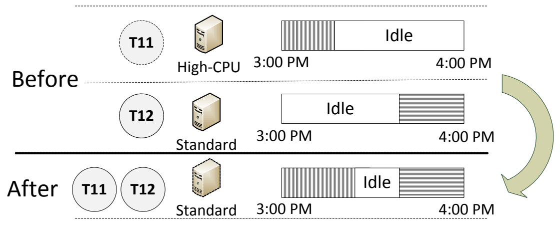

Figure 6. Instance consolidation.

Algorithm 2 – Instance Consolidation

Input – LoadVector LV_m and exiting instance number N_m for each VM type

Output – Updated LoadVector LV_m after instance consolidation

```
1:  For (each $VM_m$ where $LV_m > N_m$)
2:      For (each $VM_n$ where $LV_n <= N_n$)
3:          If ($InstanceNum(LV_n + LV_n(\text{TopTask}[VM_m])) <= N_n$ &&
              tasks following TopTask[$VM_m$] do not change scheduled VM)
4:              $LV_m$ – $LV_m$(TopTask[$VM_m$])
5:              $LV_n$ + $LV_n$(TopTask[$VM_m$])
6:              Schedule TopTask[$VM_m$] to $VM_n$ instances
7:          End If
8:          If ($InstanceNum(LV_m) < = N_m$)
9:              break
10:         End If
11:     End For
12: End For
```

Step 5 – Dynamic scheduling. After determining the number of instances of each VM type, we use the Earliest Deadline First (EDF) algorithm to schedule tasks on each VM type. After deadline assignment and instance consolidation, every task is scheduled to a VM type. We sort the tasks by their deadlines for each VM type, and schedule the task with the earliest deadline whenever an instance is available. Through dynamic scaling, the task facing the deadline miss can be found in time and the auto-scaling mechanism can immediately acquire instances to finish the task within its deadline. In other words, the nature of dynamic scaling ensures that the load vector is always less than 1 for every instance type. In other words, $\sum_i \frac{t_i}{T_{end\_i} - T_{start\_i}} < 1$. We know EDF is the optimal scheduling algorithm in this case [37][38]. Therefore, all the tasks will finish by their assigned deadlines. The following pseudo code describes our auto-scaling solution.

Algorithm 3 – Auto-Scaling

```
1:  While (true)
2:      Update $LV_m$ for tasks $t_i$
3:      Get existing $N_m$ for each VM type
4:      $LV_m \leftarrow InstanceCosolidation(LV_m , N_m)$
5:      For (each $VM_m$ where $NumNeeded(LV_m) > N_m$)
6:          $Acquire(VM_m , NumNeeded(LV_m) - N_m)$
7:      End For
8:      For (each $VM_m$ where $NumNeed(LV_m) < N_m$)
9:          For (each instance $I_i$ of type $VM_m$ )
10:             If ($I_i$ approaches multiples of full hour operation
                    && $NumShutdown < N_m - NumNeeded(LV_m)$)
11:                 $ShutDown(I_i)$
12:                 $NumShutdown$ ++
13:             End If
14:         End For
15:     End For
16:     EDF-Scheduling($t_i$)
17: End While
```

4. Evaluation

We evaluate our approach using three types of applications with different workload patterns and deadlines. We first compare our auto-scaling mechanism with two other approaches on the cost and the instance utilization of running cloud applications. Then, we analyze the effects of the workload volume. Then, we evaluate the ability of our approach to handle imprecise input parameters, i.e. the estimated task running time and the instance acquisition lag. Finally, we evaluate the mechanism's overhead.

We use three types of representative applications: Pipeline, Parallel and Hybrid. Pipeline applications are simple multiple-stage applications in which tasks need to be processed one by one with precedence constraints. Parallel applications feature a high degree of potential concurrency, because there are only limited precedence constraints. Hybrid applications are mix of pipeline applications and parallel applications. Task dependencies in Hybrid applications can be very complex. Figure 7 illustrates the pipeline application, the parallel application and the hybrid application we use in our evaluation. They are adopted from [16].

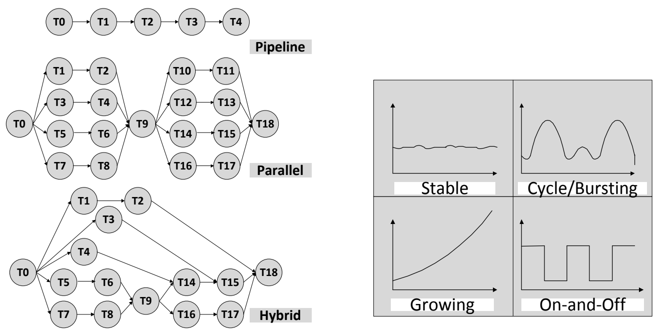

Figure 7. Application models and workload patterns.

In addition to the three application models, we simulate four types of VMs: Micro, Standard, high-CPU and high-Memory instances. The prices (Table 1) of these VMs are the same as Amazon EC2 [41].

Table 1. The VMs

VM Type	*Price*
Micro	\$0.02/hour
Standard	\$0.085/hour
High-CPU	\$0.68/hour
High-Memory	\$0.50/hour

It is a challenge to establish a baseline to compare with our mechanism. Existing research either targets a batch-queue model (which does not handle task dependencies) or a single workflow instance (not a stream of submitted jobs). Moreover, cost-efficiency is usually not a first-class concern. The rule-based trigger mechanism is a reasonable baseline. However, we must determine the scaling indicators and thresholds by ourselves. As argued earlier, the mechanisms do not really solve the performance-resource matching problem and any rules we pick could result in the risk of an "unfair game".

Therefore, we decided to use two existing approaches, *Greedy*[16] [16] and *GAIN* [17], as the baseline for our study. We compare our approach denoted as *Scaling-Consolidation-Scheduling* (*SCS*) with these two approaches. These two approaches are originally designed for the cost-aware single-workflow execution in the utility computing environment. We extend these two approaches to enable them to support continuous workflow submissions and make them aware of the instance-hour billing model in the cloud environment. First, they treat each submitted job independently using their original algorithms. They reserve instances for each job and acquire more instances when the number of live instances is insufficient. Second, acquired instances can only be released when they approach multiples of full hour operation and no jobs need them. In this way, all job requests are served in a timely manner and we reduce the number of wasted partial instance hours. We modified the two approaches to support two operations – *Acquire* and *Release* in the cloud environment. Note, the original *GAIN* approach starts with the cheapest plan and iteratively improves the plan based on the cost-efficiency rank until the budget cap is reached. Here we change the break condition so that the estimated job finishing time is before the deadline. For the Greedy algorithm, we always try to find the cheapest instance among all the live instances first. If it is not sufficient, we then acquire the cheapest instance from the all available instance types.

```
Algorithm 4 – Referenced Algorithm Extension
Acquire
1:  For (each submitted job $j_J$ )
2:    Use Greedy/GAIN to determine the instance type ($j_J^{S_i} \rightarrow VM_m$)
3:    For (each ready $j_J^{S_i}$)
4:      If (one instance $I_i$ of type $VM_m$ is available)
5:        Schedule $j_J^{S_i}$ on $I_i$
6:      Else
7:        Acquire($VM_m$ , 1) && schedule $j_J^{S_i}$ on the instance
8:      End If
9:    End For
10: End For
Release
1:  For (each VM instance $I_i$)
2:    If ($I_i$ approaches multiples of full hour operation
          && no job needs it)
3:      ShutDown($I_i$)
4:    End If
5:  End For
```

4.1.1 Cloud workload patterns

We first evaluate the performance of our auto-scaling mechanism with four representative workload patterns in the cloud environment [39][40]. The four workload patterns (see Figure 7) are Stable, Growing, Cycle/Bursting and On-and-Off. Each of these workloads represents a typical application or scenario. For example, the Growing workload pattern may represent a scenario in which a news or video suddenly becomes popular and brings in more and more users to hit the button. The workload keeps increasing very fast. The Cyclic/Bursting workload may represent the workload pattern of an online retailer. Daytime has more workload than the night and holiday shopping seasons may handle more traffic than normal. The On-and-Off workload pattern represents the work to be processed periodically or occasionally, such as batch processing and data analysis performed daily or weekly in a research department. These applications have relatively short active period, after which the service can be switched off or be maintained at the lowest service level. In our evaluation, we have randomly generated the task execution time on different types of VMs and tried 50 combinations for each workload pattern. All the test results have shown similar performance. Therefore we will not detail the information of the task running time here. For each application and workload pattern, we simulate a 72-hour period for each test with four different deadlines – 0.5 hour, 1 hour, 1.5 hour and 2 hour.

The test results with different workload patterns are shown in Figure 8, Figure 9, and Figure 10. For each application model and deadline, we show the total running cost and the average instance utilization information. In all the cases, we assume the running time of each task and the lag of instance acquisitions (6.5 min) can be precisely estimated. Therefore, *SCS, Greedy and GAIN* can each finish the jobs before user specified deadlines. We will discuss the performance of handling imprecise parameters in the next subsection. As shown from the figures, for almost all the cases, *SCS* costs the least and produces higher instance utilization (higher utilization implies fewer idle instance hours) compared to the *Greedy* and *GAIN* approaches. The cost savings ranges from 9.8% to 40.4%.

When the deadline is short, these three approaches tend to have similar performance, because all tasks are forced to run on their fastest machines to finish the job within the deadline. In other words, all three approaches generate very similar scheduling/scaling plans and there is not much room optimized for cost saving. When the deadline is longer (the “scheduling slack time” is greater), *SCS* generally can save the most cost, the *Greedy* approach performs the worst and the *GAIN* approach performs in between. The *Greedy* approach always chooses the cheapest machine instead of considering cost-efficiency for each task. In this case, many tasks actually cost more running on the cheapest machines than on their preferred machines. Therefore, the total cost is high. This result explains the importance of choosing suitable types of VMs for different workloads. The *GAIN* approach however always schedules the tasks on their preferred cost-efficient instances. In this way, it achieves cost optimization for each job and saves more cost than the *Greedy* approach, but it has lower instance utilization rate than the *Greedy* approach, which implies more partial instance-hours are wasted. The *SCS* approach not only takes advantage of the task-level cost-efficiency but also better utilizes the partial instance-hours through instance consolidation. Therefore it saves more cost than the *GAIN* approach and also has higher instance utilization rate.

Among the four workload patterns, the Growing case has the highest utilization, because as more and more jobs are submitted very quickly, all instances are filled with tasks and partial instance-hours becomes fewer and fewer. Generally speaking, when the workload volume is high enough, task level cost-efficiency could dominate the overall cost. For this reason, the *SCS* and *GAIN* approaches can always beat the *Greedy* approach. When the deadline is longer, such cost-saving benefits become clearer, because the *Greedy* approach will place most tasks on the cheapest machines and therefore incur more cost than shorter deadlines.

4.1.2 Heavy workload & light workload

In extreme cases, the workload volume can be very low, such as a single workflow instance. In such cases, the idea of placing tasks on their preferred cost-efficient instances may not always save money, because instances are not fully utilized and there could be a large portion of unutilized instances hours. In other words, the benefit of instance consolidation overweighs the cost-efficiency for individual tasks. However, when the workload volume is high enough to fill all the instances, placing tasks cost-efficiently also implies cost-efficiency globally. This is because all the jobs can be processed with the minimum cost and there are very few idle instances. To illustrate this point, we show the evaluation results of the pipeline application with both low (X) and high (10X) workload volume using 1 hour deadline. From

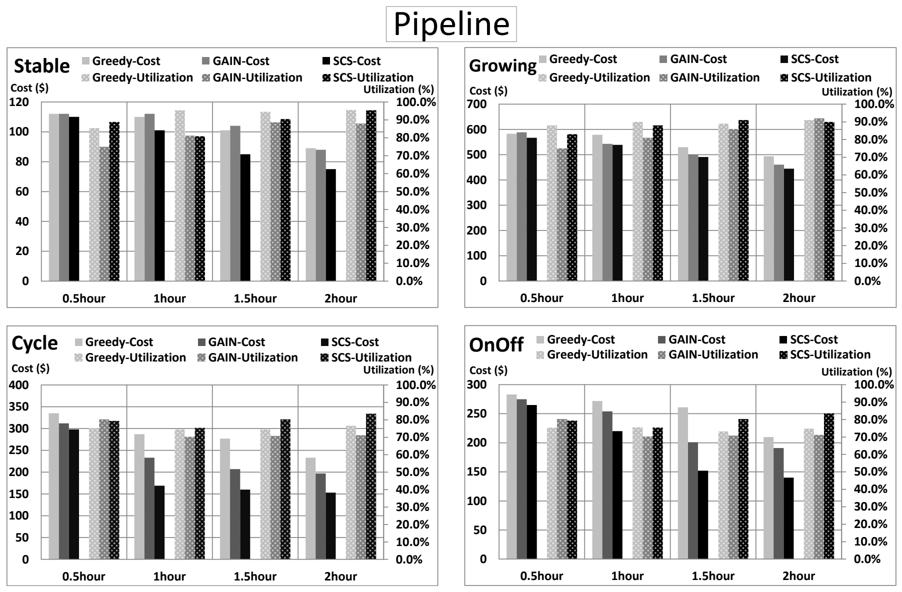

Figure 8. Pipeline application.

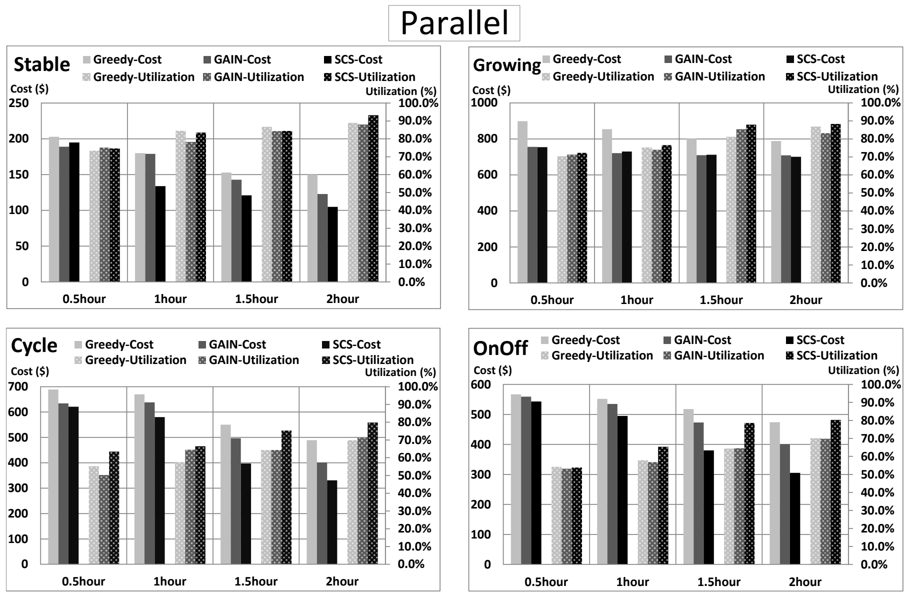

Figure 9. Parallel application.

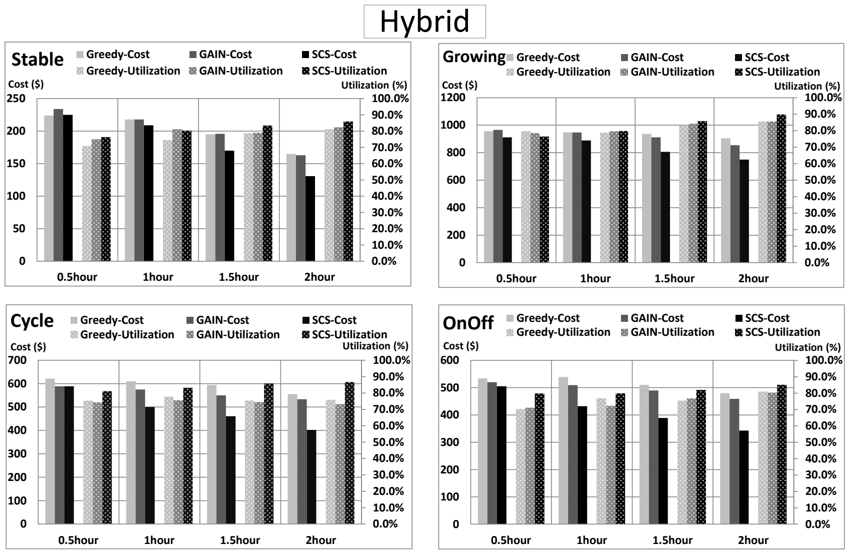

Figure 10. Hybrid application.

Figure 11, we can see that the *Greedy* approach works better than *GAIN* when the workload volume is low and performs worse when the workload volume is high. This is because it schedules as many tasks on the cheapest machines as possible, which is actually an instance consolidation strategy. The *SCS* approach works better than the *Greedy* and *GAIN* approaches in both low and high workload environment. It handles the low volume cases through instance consolidation and has the same advantages of task level cost-efficiency when the workload volume becomes high. Since parallel and hybrid applications show the similar performance, we do not duplicate the results here.

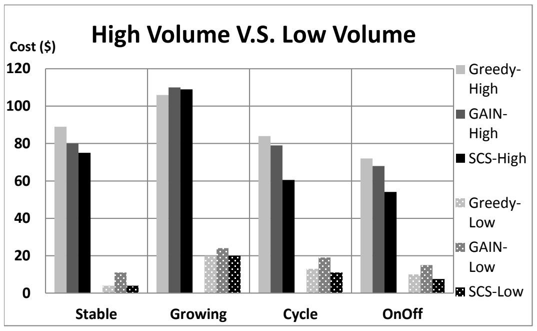

Figure 11. Heavy workload and light workload.

4.1.3 Handling imprecise parameters

We assume we have an estimate of the running time of tasks and the lag time of instance acquisition. Under such assumptions, we have shown that all deadlines can be met through our dynamic scaling and EDF scheduling. However, precise estimation of task execution time may not be always available in practice and the lag of instance acquisition is actually not under a user's control. In this section, we evaluate the performance of our approach on handling imprecise input parameters. We first allow the real task running time to be the estimated running time but with 20% variance and test the deadline non-miss rate for the pipeline application with 0.5 hour deadline (See Figure 12). The dynamic scaling nature of *SCS* handles the imprecise task runtime estimation pretty well compared to the *Greedy* and the *GAIN* approaches. It can complete more than 90% of jobs within their deadlines, which is much better than the other two approaches. In fact, as the users allow longer deadlines, *SCS* can have an even higher deadline non-miss rate. We then test a similar 20% variance for real instance acquisition lag time and test the deadline non-miss rate for the pipeline application with 1 hour deadline (See Figure 13). The lag of instance acquisition can affect the performance more than the task running time estimation (deadline non-miss rate is below 80%), because the auto-scaling mechanism reacts to the dynamic changes through instance acquisition. This is the core function of the auto-scaling mechanism. Imagine in the extreme cases, in which the acquired instance will never be ready to process user tasks, all jobs will not be finished and miss the deadlines. Another factor to determine whether the lag of instance acquisition is acceptable is actually the user's performance requirement. If the user has a very urgent deadline and needs to acquire new instances, 1 minute and 10 minute lag do not make a difference when the criteria is no deadline-miss. On the other hand, if the deadline is 2 hours for a short job, even if the instance acquisition lag is 30 minutes, the deadlines can still be met.

For all the workload patterns, we can see that the Growing case is the worst performance case. This is because it acquires more instances than other test cases, and therefore is affected more by the imprecise lag estimation. This test case shows that the instance acquisition lag plays a very important role in the performance of an auto-scaling mechanism. Workload prediction techniques are needed to prepare early for instance acquisitions. Reducing the frequent operations (e.g. parallelism reduction) of instance acquisition and release is also beneficial.

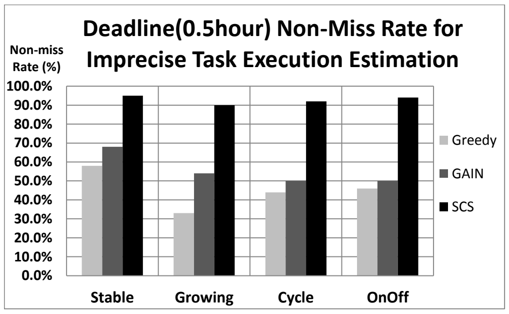

Figure 12. Imprecise task execution time.

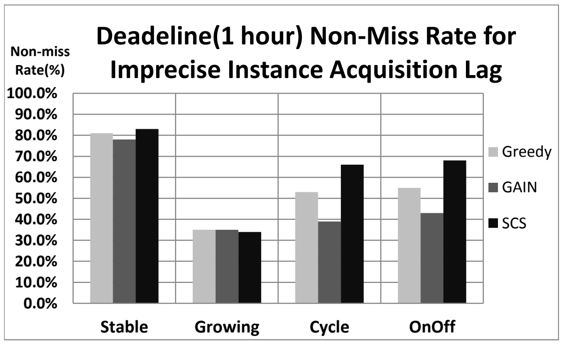

Figure 13. Imprecise instance acquisition lag.

4.1.4 Mechanism Overhead

Finally, we evaluate the overhead of our mechanism. In this test, we ignore the overhead of runtime information monitoring and update, such as the number of newly submitted jobs and the progress of running tasks. In practice, such information monitoring functionality may be provided by the cloud provider, such as Windows Azure Diagnostics [42] and AWS CloudWatch [43] or implemented by the application itself. The overhead therefore largely depends on the way it is implemented. Here we only focus on the performance of the core scheduling and scaling algorithm instead of the complete monitor-control loop. The test is run on a desktop with Intel P4 2.4G CPU, 4G memory and 512G storage. We measure the time of updating the Load Vector, consolidating partial instance-hours and making scaling/scheduling decisions for different number of jobs (from 10 to 100000). In the test, we use 100 hybrid job classes with 16 different VM types. In Figure 14, we see the overhead is low and the performance scales linearly based on the job number (note, the X-axis is exponential). We achieve the low mechanism overhead through the following two techniques. The most time-consuming part, deadline assignment, can be done in preprocessing and the result can be cached after the first calculation. The later jobs of the same class do not really need to recalculate the deadline assignments each time. Secondly, the other time-consuming part, updating the Load Vector, can be implemented using pair-wise data structures instead a big array for each task, which saves both memory and compute time.

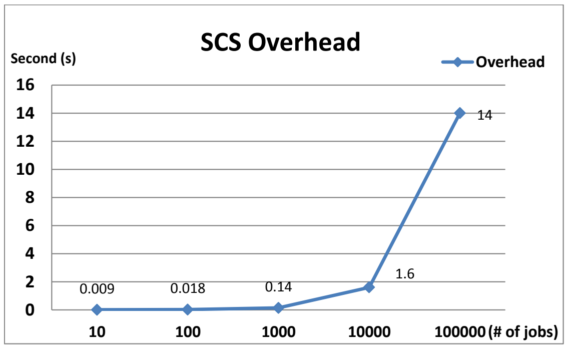

Figure 14. SCS overhead.

5. CONCLUSION

Elastic compute resources are one cornerstone of cloud computing. The dynamic scalability enables users to quickly scale up and scale down the underlying resources in response to the business volume, performance requirements and other dynamic behaviors. In this paper, we described a new auto-scaling mechanism for workflow application models. Our auto-scaling mechanism finishes all jobs by user specified deadlines in a cost-efficient way. Our solution is based on a monitor-control loop that adapts to dynamic changes such as the workload bursting and delayed instance acquisitions. Evaluation results show that our approach helps users minimize costs for various applications models and workload patterns. The cost-saving ranges from 9.8% to 40.4% compared to the other approaches. Our instance consolidation process not only improves the instance utilization rate but also efficiently handles both high and low workload volume. It successfully considers job-level and global-level cost-efficiency together. Moreover, the monitor-control loop can help handle imprecise input parameters and make fast responses to dynamic changes.

In the future, we will continue our cloud auto-scaling research in the following directions. Although dynamic scaling and EDF can ensure that all the tasks will be finished before deadline, improvements can still be made by treating the problem as a bin-packing problem, in which each instance hour can be considered as a bin. The problem is to determine the minimal number of instance hours to process all the jobs. In this way, we can try to save more instance hours by considering "fitting" first. Other global optimization techniques, such as back tracking, can also be explored. Furthermore, although workload prediction is not our focus in this paper, workload prediction techniques could enable the auto-scaling mechanism to prepare early for the incoming workload bursting, reduce the effects of delayed instance acquisitions and avoid VM churn. In addition, task and VM failures will be considered.

Also, in this work, we assume users have unlimited budget and the goal is to minimize the cloud spending while maintaining the target level of service. Our next step is to maximize user benefits/utility under a budget constraint. For example, given a monthly budget approved by the CFO, the IT department uses the auto-scaling mechanism to minimize the response time of customer requests to improve the user experience.

Our final area of future research is cloud auto-scaling for data intensive applications. In this paper, data transfer between tasks is not directly considered, as we assume data uploading and downloading is part of each task and tasks communicate temporary results through a central storage. However, this may not always be the case. In a data intensive application or computing framework, like MapReduce[44], Hadoop[45] and DryAd[46], data movement activities can dominate both the performance and the cost. The trade-off between data transfer performance and cost needs to be carefully considered.

REFERENCES

[1] Amazon EC2. http://aws.amazon.com/ec2/

[2] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez and M. Humphrey. 2010. Early Observations on the Performance of Windows Azure. *In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. Chicago, Illinois, June 21, 2010. DOI=http://dx.doi.org/10.1145/1851476.1851532

[3] M. Mao, J. Li and M. Humphrey. 2010. Cloud Auto-Scaling with Deadline and Budget Constraints. *In Proceedings of 11th ACM/IEEE International Conference on Grid Computing*. Brussels, Belgium, Oct 25-28, 2010.

[4] H. Lim, S. Babu, J. Chase, and S. Parekh. 2009. Automated Control in Cloud Computing: Challenges and Opportunities.

In 1st Workshop on Automated Control for Datacenters and Clouds. June 2009. DOI=http://dx.doi.org/10.1145/1555271.1555275

[5] P. Padala, K. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. 2007. Adaptive Control of Virtualized Resources in Utility Computing Environments. *EuroSys*, 2007. DOI=http://dx.doi.org/10.1145/1272998.1273026

[6] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. 2005. Dynamic provisioning of multi-tier internet applications. *In 2nd International Conference on Autonomic Computing*, Seattle, WA, USA, June 2005. DOI=http://dx.doi.org/10.1109/ICAC.2005.27

[7] Q. Zhang, L. Cherkasova, E. Smirni. 2007. A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. *In Proceedings of the Fourth International Conference on Autonomic Computing*, Jacksonville, Florida, USA, June 2007. DOI=http://dx.doi.org/10.1109/ICAC.2007.1

[8] A. Chandra, W. Gong and P. Shenoy. 2003. Dynamic Resource Allocation for Shared data centers using online measurements. *In Proceedings of the 11th International Workshop on Quality of Service*, 2003. DOI=http://dx.doi.org/10.1145/885651.781067

[9] J. Chase, D. Irwin, L. Grit, J. Moore, and S. Sprenkle. 2003. Dynamic Virtual Clusters in A Grid Site Manager. *In Proceedings of the 12th High Performance Distributed Computing*, Seattle, Washington, June 22-24, 2003.

[10] S. Park and M. Humphrey. 2008. Feedback-Controlled Resource Sharing for Predictable eScience. *In Proceedings IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC08)*, Austin, Texas, 2008.

[11] S. Park and M. Humphrey. 2010. Predictable High Performance Computing using Feedback Control and Admission Control. *IEEE Transactions on Parallel and Distributed Systems(TPDS)*, Mar, 2010. DOI=http://dx.doi.org/10.1109/TPDS.2010.100

[12] P. Ruth, P. McGachey and D. Xu. 2005. VioCluster: Virtualization for Dynamic Computational Domains. *Cluster Computing, 2005. IEEE International, pages 1-10*, Sep 2005.

[13] P. Marshall, K. Keahey and T. Freeman. 2010. Elastic Site Using Clouds to Elastically Extend Site Resources. *In the 10th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, Melbourne, Victoria, Australia, 2010. DOI=http://dx.doi.org/10.1109/CCGRID.2010.80

[14] M.D. Assuncao, A.D. Costanzo, and R. Buyya. 2009. Evaluating the Cost-benefit of Using Cloud Computing to Extend the Capacity of Clusters. *In Proceedings of the 18th ACM international symposium on High performance distributed computing*, Munich, Germany, June 11-13, 2009. DOI=http://dx.doi.org/10.1145/1551609.1551635

[15] D. Menasc and E. Casalicchio. 2004. A Framework for Resource Allocation in Grid Computing. *In Proc. of the 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, pp. 259-267*, 2004.

[16] J. Yu, R. Buyya and C. Tham. 2005. A Cost-based Scheduling of Scientific Workflow Applications on Utility Grids. *In Proceedings of the First IEEE International Conference on e-Science and Grid Computing*. Melbourne, Australia, Dec. 2005, pp. 140-147. DOI=http://dx.doi.org/10.1109/E-SCIENCE.2005.26

[17] R. Sakellariou and H. Zhao. 2005. Scheduling Workflows with Budget Constraints. *Integrated Research in Grid Computing. CoreGrid series*, Springer-Verlag, 2005.

[18] J. Yu and R. Buyya. 2006. Scheduling Scientific Workflow Applications with Deadline and Budget Constraints using Genetic Algorithms. *Sci. Program*, 2006, 14(3-4), 217–230,

[19] J. Yu and R. Buyya. 2006. A Budget Constraint Scheduling of Workflow Applications on Utility Grids using Genetic Algorithms. *In Proceedings of the Workshop on Workflows in Support of Large-Scale Science* (WORKS06). Paris, France, 2006.

[20] Y. Yazir, C. Matthews, R. Farahbod, S. Neville, etc. 2010. Dynamic Resource Allocation in Computing Clouds using Distributed Multiple Criteria Decision Analysis. *In 3rd International Conference on Cloud Computing*, Miami, Florida, USA, 2010. DOI=http://dx.doi.org/10.1109/CLOUD.2010.66

[21] M. Mazzucco, D. Dyachuk and R. Deters. 2010. Maximizing Cloud Providers Revenues via Energy Aware Allocation Policies. *In 3rd International Conference on Cloud Computing*, Miami, Florida, USA, 2010. DOI=http://dx.doi.org/10.1109/CLOUD.2010.68

[22] I. Goiri, J. Guitart and J. Torres. 2010. Characterizing Cloud Federation for Enhancing Providers' Profit. *In 3rd International Conference on Cloud Computing*, Miami, Florida, USA, 2010. DOI=http://dx.doi.org/10.1109/CLOUD.2010.32

[23] F. Chang, J. Ren and R. Viswanathan. 2010. Optimal Resource Allocation in Clouds. *In 3rd International Conference on Cloud Computing*, Miami, Florida, USA 2010. DOI= http://dx.doi.org/10.1109/CLOUD.2010.38

[24] E. Deelman, G. Singh, M. Livny, B Berriman, and J. Good. 2008. The Cost of Doing Science on the Cloud: The montage example. *In Proceeding SC '08 Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. pages 1-12. 2008.

[25] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi and C. Krintz. 2010. See Spot Run: Using Spot Instances for MapReduce Workflows. *In 2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud* 2010. Boston, MA. June 2010.

[26] R. Bossche, K. Vanmechelen and J. Broeckhove. 2010. Cost-Optimal Scheduling in Hybrid IaaS Clouds for Deadline Constrained Workloads. *In 3rd International Conference on Cloud Computing*, Miami, Florida, USA, 2010. DOI=http://dx.doi.org/10.1109/CLOUD.2010.58

[27] A. Oprescu and T. Kielmann. 2009. Bag-of-Tasks Scheduling under Budget Constraints. *In 2nd International Conference on Cloud Computing*, Los Angeles, USA, 2009.

[28] RightScale. http://rightscale.com

[29] enStratus. http://www.enstratus.com

[30] Scalr. https://www.scalr.net

[31] AzureScale. http://archive.msdn.microsoft.com/azurescale

[32] Windows Azure. http://www.microsoft.com/windowsazure

[33] G. Nudd, D. Kerbyson, E. Papaefstathiou, S Perry, J. Harper and D. Wilcox. 2000. PACE- A Toolset for the performance Prediction of Parallel and Distributed Systems. *In International Journal of High Performance Computing Applications*, Volume 14 Issue 3, August 2000. DOI=http://dx.doi.org/10.1177/109434200001400306

[34] K. Cooper et al. 2004. New Grid Scheduling and Rescheduling Methods in the GrADS Project. *In NSF Next Generation Software Workshop, International Parallel and Distributed Processing Symposium*, Santa Fe, IEEE CS Press, Los Alamitos, CA, USA, April 2004. DOI=http://dx.doi.org/10.1007/s10766-005-3584-4

[35] S. Jang et al. 2004. Using Performance Prediction to Allocate Grid Resources. *Technical Report 2004-25, GriPhyN Project*, USA. 2004.

[36] W. Smith, I. Foster, and V. Taylor. 1998. Predicting Application Run Times Using Historical Information. *In Workshop on Job Scheduling Strategies for Parallel Processing, 12th International Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP '98)*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

[37] Earliest Deadline First Scheduling. http://en.wikipedia.org-/wiki/Earliest_deadline_first_scheduling

[38] Earliest Deadline First Scheduling. http://retis.sssup.it-/~lipari/courses/str06/10.edf.pdf

[39] Cloud Workload Patterns. http://bramveen.com/2010/07/13/-workload-patterns-for-cloud-computing/

[40] Cloud Workload Patterns. http://www.microsoftpdc.com-/2009/SVC54

[41] Amazon EC2 Price. http://aws.amazon.com/ec2/#pricing

[42] Windows Azure Diagnostics. http://archive.msdn.microsoft-.com/WADiagnostics

[43] AWS CloudWatch. http://aws.amazon.com/cloudwatch/

[44] J. Dean and S. Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *In Communications of the ACM - 50th anniversary issue: 1958 - 2008*, Volume 51 Issue 1, January 2008. DOI=http://dx.doi.org/10.1145/1327452.1327492

[45] Hadoop. http://hadoop.apache.org/

[46] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly. 2007. *In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, Lisbon, Portugal, 2007. DOI=http://dx.doi.org/10.1145/1272998.1273005