Preliminary version. Final version appears In Proceedings of 11th ACM/IEEE International Conference on Grid Computing (Grid 2010). Oct 25-28, 2010. Brussels, Belgium.

Cloud Auto-Scaling with Deadline and Budget Constraints

Ming Mao，Jie Li，Marty Humphrey
Department of Computer Science
University of Virginia
Charlottesville, VA, USA 22904
{ming, jl3yh, humphrey}@cs.virginia.edu

Abstract—**Clouds have become an attractive computing platform which offers on-demand computing power and storage capacity. Its dynamic scalability enables users to quickly scale up and scale down underlying infrastructure in response to business volume, performance desire and other dynamic behaviors. However, challenges arise when considering computing instance non-deterministic acquisition time, multiple VM instance types, unique cloud billing models and user budget constraints. Planning enough computing resources for user desired performance with less cost, which can also automatically adapt to workload changes, is not a trivial problem. In this paper, we present a cloud auto-scaling mechanism to automatically scale computing instances based on workload information and performance desire. Our mechanism schedules VM instance startup and shut-down activities. It enables cloud applications to finish submitted jobs within the deadline by controlling underlying instance numbers and reduces user cost by choosing appropriate instance types. We have implemented our mechanism in Windows Azure platform, and evaluated it using both simulations and a real scientific cloud application. Results show that our cloud auto-scaling mechanism can meet user specified performance goal with less cost.**

Keywords-cloud computing; auto-scaling; dynamic scalability; integer programming

I. INTRODUCTION

Clouds have become an attractive computing platform which offers on-demand computing power and storage capacity. Its dynamic scalability enables users to scale up and scale down the underlying infrastructure in response to business volume, performance desire and other dynamic behaviors. To offload cloud administrators' burden and automate scaling activities, cloud computing platforms have also offered mechanisms to automatically scale up and down VM capacity based on user defined policy, such as AWS auto-scaling [1]. Using auto-scaling, users can define triggers by specifying the performance metrics and thresholds. Whenever the observed performance metric is above or below the threshold, a predefined number of instances will be added to or removed from the application. For example, a user can define a trigger like "Add 2 instances when CPU usage is above 60% for 5 minutes".

Such automation largely enhances the cloud dynamic scalability benefits. It transparently adds more resources to handle increasing workload and shuts down unnecessary machines to save cost. In this way, users do not have to worry about capacity planning. The underlying resource capacity can be adaptive to the application real-time workload. However, challenges arise when people look deeper into the mechanisms.

In cloud auto-scaling mechanisms, performance metrics normally include CPU utilization, disk operation and bandwidth usage, etc. Such infrastructure level performance metrics are good indicators for system utilization information. But it cannot clearly reflect the quality of service a cloud application is providing or tell whether the performance meets user's expectation. Choosing appropriate performance metric and finding precise threshold is not a straightforward task, and cases become more complicated if the workload pattern is continuously changing. Moreover, considering individual utilization information only may not robust to scale [9]. For example, a cluster going from 1 to 2 instances can increase capacity by 100%, while going from 100 to 101 instances can only increase capacity by 1%. Current simple auto-scaling mechanisms normally ignore such non-constant effects when adding a fixed number of resources.

Another factor such auto-scaling mechanisms overlook is the time lag to boot a VM instance. Though instance acquisition requests can be made at any time, they are not immediately available to users. Such instance startup lag typically involves finding the right spot for the requested instances in cloud data center, downloading specified OS image, booting the virtual machine, and finishing network setup, etc. Based on our experiences and research [5], it could take as long as 10 min to start an instance in Windows Azure, and such startup lag can change over time. In other words, it's very likely that users may request instances late if they do not consider instance startup time factor.

Cost is also an issue worth careful consideration when using cloud. Cloud computing instances are charged by hours. A fraction of an hour is counted as a whole hour. Therefore, it could be a waste of money for machines shut down before a whole hour operation. In addition to noticing the full hour principal, clouds now usually offers various instance types, such as high-CPU and high I/O instances. Choosing appropriate instance types based on the application workload can further save user money and improve performance. We believe cloud scaling activities can be done better by considering using different instance types than just manipulating instance numbers.

In this paper, we present a cloud dynamic scaling mechanism, which could automatically scale up and scale down underlying cloud infrastructures to accommodate changing workload based on application level performance metric – job deadline. During the scaling activities, the mechanism tries to form a cheap VM startup plan by choosing appropriate instance types, which could save more cost compared to only considering one instance type.

The rest of this paper is organized as following. Section II introduces the related work. Section III identifies cloud scaling characteristics and describes application performance model. Section IV formalizes the problem and details our implementation architecture in Windows Azure platform. Section V evaluates our mechanism using both simulations and a real scientific application. Section VI concludes the paper and describes future works.

II. Related Work

There have been a number works on dynamic resource provisioning in virtualized computing environment [9][10][12][4]. Feedback control theory has been applied in these works to create autonomic resource management systems. In [9][10], target range is proposed to solve the control stability issue. Further in [9], it focuses on control system design. It points out that resizing instances is a coarse grained actuator when applying control theory in cloud environment and proposed proportional threasholding to fix the non-constant effect problem. These works use infrastructure level performance metrics and mainly focus on control theory application in cloud environment. They do not consider various VM types or total running cost. In [8], dynamic scaling is explored for cloud web applications. They considered web server specific scaling indicators, such as the number of current users and the number of current connections. The work uses simple triggers and thresholds to determine instance number and does not consider VM type information and budget constraints as well. In [4], they considered extending computing capacity using cloud instances and compared the incurred cost of different policies.

Particularly in cloud computing, dynamic scalability becomes more attractive and practical because of the unlimited resource pool. Most cloud providers offer cloud management API to enable users to control their purchased computing infrastructure programmatically, but few of them directly offers a complete solution for automatic scalability activities in cloud. Amazon web service auto-scaling service is one of them. AWS auto-scaling is a mechanism to automatically scale up and down virtual machine instances based on user defined triggers [1]. Triggers describe the thresholds of observed performance metric, which include CPU utilization, network usage and disk operations. Whenever the monitored metric is above the upper limit, a predefined number of instances will be started, and when it is below the lower limit, a predefined number of instances will be shut down. Another work worth mentioning here is RightScale [3]. It works as a broker between users and cloud providers by providing unified interfaces. Users can interact with multiple cloud providers on one screen. The nicely designed user interface, highly customized OS images and many predefined utility scripts enable users to deploy and manage their cloud applications quickly and conveniently. In dynamic scaling, they borrow the idea of "triggers and thresholds" but extend scaling indicator choices broadly. Including system utilization metrics, they further support some popular middle-ware performance metrics, such as Mysql connections, Apache http server requests and DNS queries. However, these scaling indicators may not be able to support all application types and not all of them can directly reflect quality of service requirements. Also, they do not consider cost explicitly. To the best of our knowledge, our work is the first auto-scaling mechanism which addresses both performance and budget constraint in cloud.

III. Cloud Scaling

A. *Cloud Scaling Characteristics and Analysis*

As a computing platform, clouds own distinct characteristics compared to utility computing and grid computing. We have identified the following characteristics which can largely affect the way people use cloud platforms, especially in cloud scaling activities.

Unlimited resources limited budget. Clouds offer users unlimited computing power and storage capacity. Though by default the resource capacity is capped to some number, e.g., 20 computing units per account in Windows Azure, such usage cap is not a hard constraint. Cloud providers allow users to negotiate for more resources. Unlimited resource enables applications to scale to extremely large size. On the other hand, these unlimited resources are not free. Every cycle used and byte transferred are going to appear on the bill. Budget cap is a necessary constraint for users to consider whey they deploy applications in clouds. Therefore, a cloud auto-scaling mechanism should explicitly consider user budget constraints when acquiring resources.

Non-ignorable VM instance acquisition time. Though cloud instance acquisition requests can be made at any time and computing power can be scaled up to extremely large, it does not mean cloud scales fast. Based on our previous experiences and research [5], it could take around 10 more minutes from an instance acquisition request until it is ready to use. Moreover, such instance startup lag could keep changing over the time. On the other side, VM shutting down time is quite stable, around 2-3 minutes in Windows Azure. This implies that users have to consider two issues in cloud dynamic scaling activities. First, count in the computing power of pending instances. If an instance is in pending status, it means it is going to be ready soon. Ignoring pending instances may result in booting more instances than necessary, therefore waste money. Second, count how long the pending instance has been acquired and how long further it needs to be ready to use. If the startup time delay can be well observed and predicted, application admin can acquire machines in advance and prepare early for workload surges.

Full hour billing model. The pay-as-you-go billing model is attractive, because it saves money when users shut down machines. However, VM instances are always billed by hours. Fraction consumption of an instance-hour is counted as a full hour. In other words, 10 minute and 60 minute usage are both billed as 1 hour usage and if an instance is started and shut down twice in an hour, users will be charged for two instance hours. The shutting down time therefore can greatly affect cloud cost. If cloud auto-scaling mechanisms do not consider this factor, it could be easily tricked by fluctuate workloads. Therefore, a reasonable policy is that whenever an instance is started, it is better to be shut down when approaching full hour operation.

Multiple instance types. Instead offering one suit-for-all instance type, clouds now normally offer various instance types for users to choose. Users can start different types of instances based on their applications and performance requirement. For example, EC2 instances are grouped into three families, standard, high-CPU and high-memory. Standard instances are suitable for all general purpose applications. High-CPU instances are well suited for computing intensive application, like image processing. High-memory instances are more suitable for I/O intensive application, like database systems and memory caching applications. One important thing is that instances are charged differently and not necessarily proportional to its computing power. For example, in EC2, c1.medium costs twice as much as m1.small. But it offers 5 times more compute power than m1.small. Thus for computing heavy jobs it is cheaper to use c1.medium instead of the least expensive m1.small. Therefore, users need to choose instance type wisely. Choosing cost-effective instance types can both improve performance and save cost.

B. *Cloud Application Performance Model*

In this paper, we consider the problem of controlling cloud application performance by automatically manipulating the running instance types and instance numbers. Instead of using infrastructure level performance metrics, we target application level performance metric, the response time of a submitted job. We believe a direct performance metric can better reflect users' performance requirements, therefore can better instruct cloud scaling mechanisms for precise VM scheduling. At the same time, we introduce cost as the other goal in our cloud scaling mechanism as well. Our problem statement is how to enable cloud applications to finish all the submitted jobs before user specified deadline with as little money as possible. To keep the cloud application performance model general and simple, we consider a single queue model as shown in Fig. 1. Also, we make following assumptions.

- Workload is considered as non-dependent jobs submitted in the job queue. Users don't have knowledge about incoming workload in advance.
- Jobs are served in FCFS manner and they are fairly distributed among the running instances. Every instance can only process a single job at one time.
- All the jobs have the same performance goal, e.g. 1 hour response time deadline (from submission to finish). Deadline can be dynamically changed
- VM instances acquisition requests can be made at any time, but it may take a while for newly requested pending instance to be ready to use. We call such time VM startup delay.
- There could be different classes of jobs, such as computing intensive jobs and I/O intensive jobs. A job class may have different processing time on different instance types. For example, a computing intensive job can run faster on high-CPU machines than high-I/O machines.
- The job queue is large enough to hold all unprocessed jobs and its performance scales well with increasing number of instances.

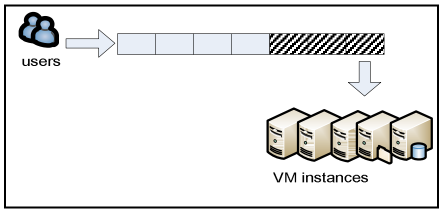

Figure 1. Cloud application performance model

IV. Solution & Architecture

Based on the problem description in previous section, we formalize the problem in this section and present our implementation architecture in Windows Azure.

A. *Solution*

One of the key insights to this problem is that, to finish the all submitted jobs before the deadline, auto-scaling mechanism needs to ensure that the computing power of all acquired VM instances is large enough to handle the workload. We summarize the key variables in the Table. I.

TABLE I. KEY VARIABLES USED IN CLOUD PERFORMANCE MODEL

Variables	Meaning
J_j	the *j*th job class
n_j	the number of J_j submitted in the queue
V	the VM type
I_i	the *i*th instance (running or pending)
c_v	the cost per hour of VM type V
d_v	average startup delay of VM type V
s_i	the time already spent in pending status of I_i
$t_{j,v}$	average processing time of running job J_j on V
D	deadline (e.g. 1 hour or 100 seconds)
C	budget constraint (dollars/hour)
W	Workload – jobs **need** to be finished
P	computing power – jobs **can** be finished

Using the above notations, we define the system workload as a vector *W*. For each job class J_j, there are n_j submitted jobs.

$$W = (J_j, n_j)$$

The computing power of instance I_i can be represented as a vector P_i. The idea is to calculate how many jobs can be finished for each job class before the deadline on instance I_i.We use deadline and individual completion time (assume all the jobs are finished by that instance) ratio to approximate the number of jobs that can be finished.

$$P_i = (J_j, \frac{D \times n_j}{\sum_j t_{j,type(I_i)} n_j})$$

For instance whose status is pending, its computing power can be represented as following, where s_i is the time already spent in starting the instance.

$$P_i = (J_j, \frac{(D-(d_{type(I_i)} - s_i)) \times n_j}{\sum_j t_{j,type(I_i)} n_j})$$

Therefore, the total computing power of current instance can be represented as $\sum_i P_i$. Clearly if W > P, we need to start more instances P_i' ($'$ means new instances) to handle the increased workload. The problem becomes finding a VM instance combination plan P_j', in which

$$\sum_i P_i' \geq W - P$$

At the same, we also want to minimize the cost we spend for these newly added instances.

$$Min(\sum_i c_{type(I_i')})$$

In the cases where there are insufficient budget, the idea to generate as much computing power as possible within the budget constraints

$$Max(\sum P_i')$$

$$\sum_i c_{type(I_i')} \leq C - \sum_i c_{type(I_i)}$$

When one instance I_s is approaching full hour operation, we need to decide whether to shut-down the machine or not. In this case, we can calculate the computing power without instance I_s , and compare with the workload. If the computing power is still big enough to handle the workload, we can remove the instance.

$$\sum_i P_i - P_s \geq W$$

To better explain the problem, we can go through a simple example. Assume we have three job classes (j_1 , j_2 , j_3) and three VM types (V_1 , V_2 , V_3). Currently, the workload in the system is [60, 60, 60] and there are two running instances I_1 and I_2 . Our goal is to find a VM type combination [n_1' , n_2' , n_3'], whose computer power is greater than or equal to target computing power and their cost is minimal among all the possible VM type combinations.

$$\begin{array}{l} j_1: \\ j_2: \\ j_3: \end{array} \underbrace{\begin{bmatrix} x \\ y \\ z \end{bmatrix}}_{\sum P'} \geq \underbrace{\begin{bmatrix} 60 \\ 60 \\ 60 \end{bmatrix}}_{W} - \underbrace{\begin{bmatrix} 10 \\ 5 \\ 20 \end{bmatrix}}_{I_1} - \underbrace{\begin{bmatrix} 10 \\ 20 \\ 5 \end{bmatrix}}_{I_2} = \begin{bmatrix} 40 \\ 35 \\ 35 \end{bmatrix}$$

$$\begin{array}{l} j_1: \\ j_2: n_1' \\ j_3: \end{array} \underbrace{\begin{bmatrix} 10 \\ 5 \\ 20 \end{bmatrix}}_{V_1} + n_2' \underbrace{\begin{bmatrix} 10 \\ 20 \\ 5 \end{bmatrix}}_{V_2} + n_3' \underbrace{\begin{bmatrix} 10 \\ 10 \\ 10 \end{bmatrix}}_{V_3} = \underbrace{\begin{bmatrix} x \\ y \\ z \end{bmatrix}}_{\sum P'} \geq \begin{bmatrix} 45 \\ 35 \\ 35 \end{bmatrix}$$

$$Min(c_1 n_1' + c_2 n_2' + c_3 n_3')$$

Where

$$c_1 n_1' + c_2 n_2' + c_3 n_3' + c_{type(I_1)} + c_{type(I_2)} \leq C$$

From the above analysis, our cloud auto-scaling mechanism is reduced to several integer programming problems. We try to minimize the cost or maximize the computing power with either computing power constraints or budget constraints. There are quite a few standard approaches to solve integer programming problems, such as cutting-plane and branch-and-bound methods [13] [14]. We will not duplicate the details here.

In addition to determining the number and type of VM instances, there are some other cases like admission control and deadline miss handling which are also interested to think about in cloud auto-scaling mechanisms. However, our work's intension is not to create a hard real-time cloud system which all jobs' deadline are guaranteed, we focus on automatic resource provisioning based on both performance goals and budget constraints. Deadline is just the metric we choose, because it can better reflect users' performance desire. Therefore, in real practice we believe these are more like policy questions. Users can choose their own policies based on their applications. For example, to maintain service availability and basic computing power, users can decide the minimum number of running instances. In other words, even there is no workload, a cloud application will always have at least 1 running instance. For admission control cases, when there's insufficient budget, auto-scaling mechanism could either accept the job and try to run with maximum computing power within the user budget constraints or users can simply deny the job. In either case, users may want to get notification from the mechanism. For deadline miss handling, users can either leave it alone or allow auto-scaling mechanism to increase as many instances as possible to speed up the remaining processing. In our implementation, we have implemented these policies and let user to configure which policy is most appropriate for their cases, and users are allowed to implement their own policies as well.

B. *Architecture*

We have designed and implemented our cloud auto-scaling mechanism in Windows Azure [3]. Figure 2 shows the architecture of our implementation. The implementation includes four components. They are performance monitor, history repository, auto-scaling decider and VM manager. Performance monitor observes the current workload in the system, collects actual job processing time and arrival pattern information, and updates the history repository. VM manager works as the adapter between our auto-scaling mechanism and cloud providers. It monitors all pending and ready VM instances, and updates history repository with actual startup time of different VM types. Moreover, it executes VM startup plan generated by auto-scaling decider and directly invokes cloud provider resource provisioning APIs. In our case, it is Windows Azure management API. Our intention is that VM manager hides all cloud provider details and can be easily replaced with other cloud adapters. Such information hiding enhances the reusability and customizability of our implementation when working with different cloud providers. History repository contains two data structures. One is the configuration file, which includes application deadline, budget constraint, monitor execution interval information, etc. As shown in Fig. 2, application administrators can dynamically control the behavior of cloud auto-scaling mechanism by changing the configuration file. The other data structure is historical data table, which records the historical job processing time and arrival pattern information provided by performance monitor, and instance startup delay information provided by VM manager. By maintaining historical data, the repository improves the input parameter preciseness and also helps decider to prepare for possible workload surges early. Decider is the core of our cloud auto-scaling mechanism. Relying on real-time workload and VM status information from performance monitor and VM manager, as well as configuration parameters and historical records from history repository, it solves the integer programming problem we formalized in the previous section and generates a VM startup plan for VM manager to execute. The VM startup plan could be empty because the workload may be well handled by exiting instances or it can contain instance type and number pairs to notify VM manager acquire enough computing power. In our current implementation, we use Microsoft Solver Foundation [11] to solve the integer programming problem. Acquiring instance actions are initialed by decider. After every sleep interval, it invokes the logic to determine the VM startup plan. On the other side, releasing instance actions are initialed by VM manager because it monitors which instance is approaching full hour operation and could be the potential shut-down targets. But it has to ask decider to see whether remaining computing power is large enough to handle the workload. We have published our current implementation as a library and plug it in MODIS application [7]. The evaluation of our mechanism in this real scientific application can be found in the next section.

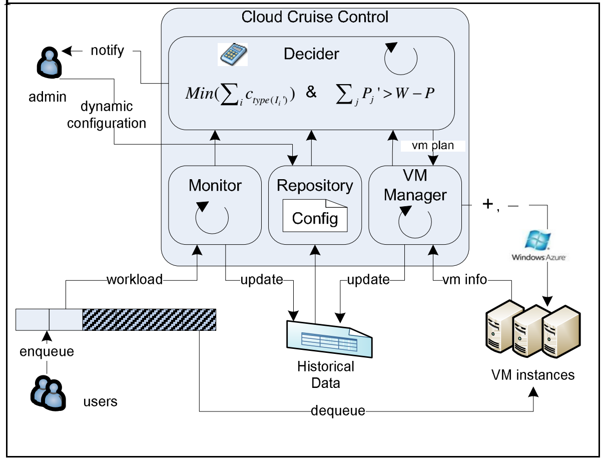

Figure 2. Architecture of Cloud auto-scaling in Azure

V. Evaluation

In this section, we evaluate our mechanism using both simulations and a real scientific application (MODIS) running in Windows Azure. Through simulation framework, we can easily control the input parameters, such as workload pattern and job processing time, which helps to identify the key factors in our mechanism. Moreover, using simulation extensively reduces the evaluation time and cost. The scientific application tests our mechanism's performance in real environment.

In our evaluation, we simulated three types of jobs. They are mix, computing intensive and I/O intensive. At the same time, we simulated three types of machines. They are General, High-CPU and High-I/O machines. We summarize their simulation parameters in Table II. The simulation data is derived from pricing tables and instance descriptions of EC2. For example, in EC2, c1.medium instance costs twice as much as m1.small. But it offers 5 times more compute power than m1.small [1]. In our case, we assume mix jobs are half computation and half I/O. The speedup factor of powerful machines is 4-5.

TABLE II. AVAREAGE PROCESSING TIME

	Mix Avg 30 jobs/hour STD 5 jobs/hour	Computing Intensive Avg 30 jobs/hour STD 5 jobs/hour	I/O Intensive Avg 30 jobs/hour STD 5 jobs/hour
General 0.085$/hour Delay 600s	Average 300s STD 50s	Average 300s STD 50s	Average 300s STD 50s
High-CPU 0.17$/hour Delay 720s	Average 210s STD 25s	Average 75s STD 15s	Average 300s STD 50s
High-IO 0.17$/hour Delay 720s	Average 210s STD 25s	Average 300s STD 50s	Average 75s STD 15s

A. Deadline

For deadline performance goal, we consider two cases. 1) Stable workload with changing deadline. We generate the workload using Table II and plot the job response time in Fig. 3. Every data point in the graph reflects the job response time in every 5 minutes and we record average, minimum and maximum response time for all the jobs finished in that interval. The deadline is first set as 3600s, then changed to 5400s and finally switched back. The purpose is to evaluate our mechanism's reaction to dynamic user performance requirement change. Fig. 3 shows that more than 95% of jobs are finished within the deadline and most of the misses happen at the second deadline change. This is mainly because our auto-scaling mechanism runs every 5 minutes and VM instances can only be ready 10-12 minutes later after acquisition requests. Besides, we also calculate the instantaneous instance utilization rate. Job processing is considered as utilized while all the other cases, such as pending and idling, are considered as unutilized. The high utilization rate (average 94%) shows that our mechanism does not aggressively acquire instances to guarantee the deadline, and 6% of time is spent on VM startups.

2) Changing workload with fixed deadline. In this test, we fix the deadline to 3600s and create three workload peaks. Base workload is 30 mix jobs per hour. The first workload peak adds another 300 mix jobs per hour. The second peak adds 300 computing intensive jobs per hour, and the third one adds 300 I/O intensive jobs per hour. The purpose of this

test is to evaluate our mechanism's reaction to sudden increasing workload and job type changes. Such workload pattern is normally seen in large volume data processing applications, in which data computation and analysis is performed in day time, and data backups and movements are performed in nights and holidays. From Fig. 4, we can see that the deadline goal is well met for all three workload peaks. When workload goes back to normal, the over acquired instances during peak moments quickly reduce job response time. As more and more unnecessary instances are shut down (approaching full hour operation), the response time goes back to average.

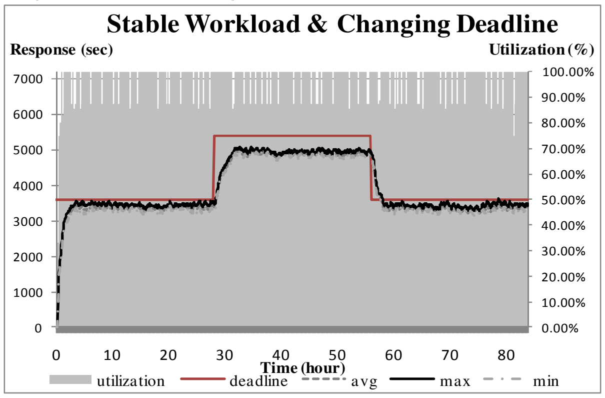

Figure 3. Stable workload with changing deadline

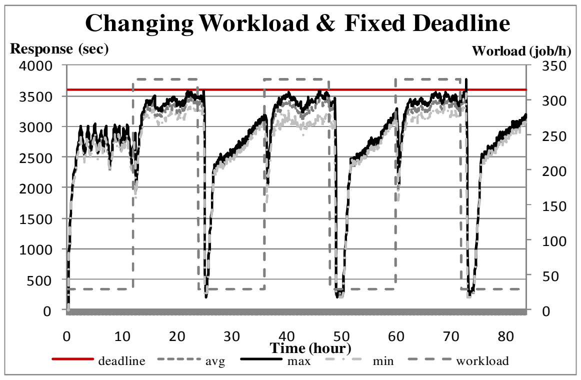

Figure 4. Changing workload with fixed deadline

B. Cost

Using the same evaluation as we did for changing workload fixed deadline, we compare the cost of using different types of VM instance. The VM type combinations are illustrated in Table III. Fig. 5 shows the comparison result.

TABLE III. INSTANCE TYPE

	VM Types	Total Cost ($) % more than optimal
Choice #1	General	98.52$ (43%)
Choice #2	High-CPU	128.86$ (87%)
Choice #3	High-IO	129.71$ (88%)
Choice #4	General, High-CPU, High-IO	78.62$ (14%)
Optimal	General, High-CPU, High-IO	68.85$

To evaluate the performance of our mechanism, in addition to the four choices, we also calculate the possible optimal cost for the same workload and compare our solution with it. The optimal solution can be obtained because we know the workload in advance and we assume we can always put a job to the most cost-effective machines, e.g., put computing intensive jobs on High-CPU instances for processing. From Fig. 5, we can see that by considering all available instance types (Choice #4), our mechanism can adapt to the workload changes and choose cost-effective instances. In this way, the real-time cost is always close to the optimal cost. On the other side, General instances always performs on average for all three workload peaks, while High-CPU and High-IO can only save cost on its preferred workload surges. Fig. 6 shows the accumulated cost. Choice #4 incurs 14% more cost than the optimal solution and saves 20% cost compared to General instance choice, 45% compared to High-CPU and High-IO. Because of symmetry, High-CPU and High-IO instances end up with almost the same cost. General instances has lower cost on average, therefore, in the long run, it outperforms High-CPU and High-IO cases. By choosing appropriate instance types, choice #4 can incur less cost in all three workload peaks like the optimal solution, hence, it outperforms all the other cases. There are two reasons why our solution cannot make the optimal decision. Auto-scaling decider does not know the future workload and can only make decisions locally. Second, it cannot control the running instance for processing a job.

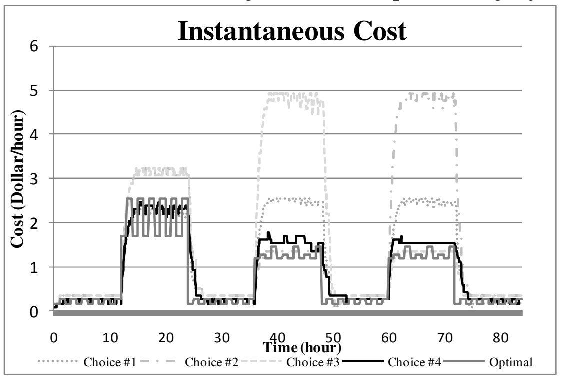

Figure 5. Instantaneous cost of changing workload & fixed deadline

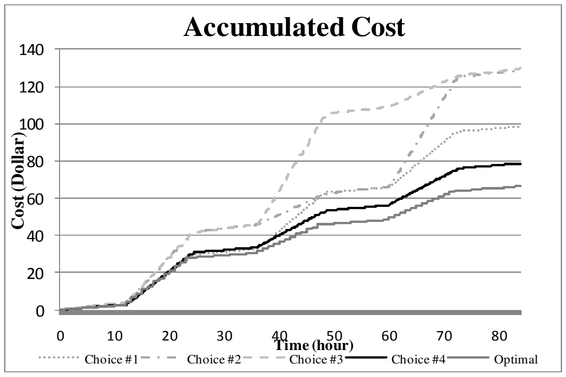

Figure 6. Accumulated cost of changing workload & fixed deadline

C. *MODIS*

In addition to simulations, we also have applied our approach to a real scientific cloud application MODIS [7]. MODIS is a cloud application built in Windows Azure platform for large volume biophysical data processing. It integrates data from ground-based sensors with the Moderate Resolution Imaging Spectroradiometer satellite data. It is now used by biometeorology lab, UC Berkeley. We first introduce MODIS workload and some configuration parameters applied. MODIS workload can be understood in the following way. 200X indicates the year, Terra and Aqua represent satellite images, and (x-y) represents the period from day x to day y. For all our tests, we use all available 15 tile images in MODIS system for a single day data processing. For example, Terra 2004 (10-12) means processing all 15 tiles of Terra images from 2004 Jan 10th to Jan 12th. This implies that totally 45 (15×3) jobs are submitted at once. In our evaluation, we find the actual job processing times range from 10 sec to 13 min with average 5 min and jobs are processed most cost-effectively in small instance types. We set the performance monitor interval as 1 min, decider interval as 5 min, initial average VM delay as 15min and we only notify the users when deadline is missed.

In MODIS evaluation, we run both moderate scale (up to 20 instances) and large scale (up to 90 instances) tests. In moderate scale evaluation, two test cases are randomly selected. One is Terra satellite 2004 (10-12) and the other one is Aqua 2008 (30-32). We record the test results in Table IV, including both performance and instance hours consumed (or cost). The table shows that 2 and 3 hour deadline goals are better met than 1 hour deadline for same workloads. After investigating the VM instance startup history, we find this is largely because instance startup delay is out of our expectation. For example, in 1 hour deadline tests, the average startup delay is around 22 minutes. Some instances even took 50 minutes to be ready. There is little time left for our mechanism to react in such cases. On the contrary, in longer deadline tests, our mechanism acquired fewer instances and hence the result is less affected by the startup delay variances. In both test cases, the theoretical computing power needed is 4 instance hours (all jobs are processed by a single instance). All tests actually acquired more than this, e.g. 9 or 10 instances hours for 1 hour deadline test cases. This is caused by VM startup delay make up and impreciseness of initial job processing time configuration. With longer deadlines, such over acquisition is corrected because fewer instances are acquired and job processing time is also updated by the historical table. Therefore, longer deadline test cases also incur less cost.

TABLE IV. MODIS MODERATE SCALE EVALUATOIN

	1hour deadline	*2hour deadline*	*3hour deadline*
Terra 2004(10-12) Total 45 jobs 4 C.H.* or 0.48$	18 min late 9 C.H.or 1.08$	8 min early 6 C.H or 0.72$	20 min early 5 C.H.or 0.6$
Aqua 2008(30-32) Total 45 jobs 4 C.H. or 0.48$	15min late 10 C.H or 1.2$	20 min early 7 C.H.or 0.84$	29 min early 5 C.H.or 0.6$

* C.H. – computing hour 1C.H. = 0.12$ in Windows Azure

For large scale (up to 90 instances) MODIS evaluations, we performed two tests and recorded the results in Table V. Similar to moderate scale evaluations, longer deadline tests show better results. Again, unexpected VM startup delay is the dominating factor. We find Windows Azure has longer VM startup delay and larger variances in large size instance acquisition cases. For example, in Terra & Aqua 2006 (1-75) 2 hour deadline test, the average VM startup delay is 40 minutes and there's one instance which is still not ready 2 hours later. For 2006 (1-125) 2 hour deadline test, our decider calculation shows 95 instances are needed, which is beyond our resource limit. This job is successfully identified and denied.

TABLE V. MODIS LARGE SCALE EVALUATOIN

	2 hour deadline	*4 hour deadline*
Terra & Aqua 2006(1-75) Total 1125 jobs 93 C.H. or 11.16$	20min late 170 C.H. or 20.4$	6 min early 132 C.H. or 15.84$
Terra & Aqua 2006(1-150) Total 2250 jobs 185 C.H. or 22.2$	Admission Denied	22 min early 243 C.H. or 29.16$

To better demonstrate our mechanism working details, we present instance acquisition and release information for test case Terra & Aqua 2006 (1-75) 4 hour deadline in Fig. 7. This test totally includes 1125 jobs and is submitted at time 0. As shown in the figure, after around 4 minutes, the decider started 34 instances (instance 1 - 34) to handle the workload. The real instance acquisition time took much longer than we configured. Therefore, around 1.5 hours later, the decider started another 6 instances (instance 35 - 40) to make up for such unexpected startup delay. After approaching 2 full hour operation, these 6 instances were shut down due to decreased workload. After all jobs are finished, instance 1 to instance 34 were shut down when they approached 4 hour operation. At that time, only instance 0 was kept alive to maintain service availability. In this case, the theoretical job processing times needed is 93 hours. The real instance hours consumed is 132 hours with 36 hours spent on VM startup. Both moderate and large scale tests show that longer deadline has better performance and incurs less cost. This is because longer deadline tests are less affected by VM startup delay and have more chances to use the updated job processing time.

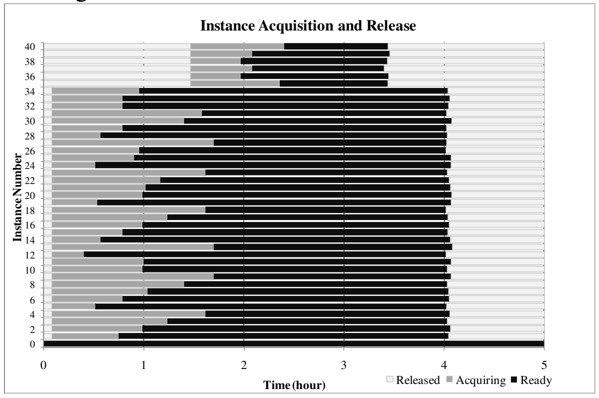

Figure 7. Instance acquistion and release

VI. CONCLUSION & FUTURE WORKS

In this paper, we present a mechanism to dynamically scale cloud computing instances based on deadline and budget information. The mechanism automatically scales up and scales down VM instances by considering two aspects of a cloud application - performance and budget. From performance perspective, our cloud auto-scaling mechanism enables cloud applications to finish all submitted jobs within the desired deadline by acquiring enough VM instances. From cost perspective, it reduces user cost by acquiring appropriate instance types which incurs less money and shuts down unnecessary instances when they approach full hour operation. We interpreted the instance startup plan generation as an optimization problem and used integer programming to solve it. We have designed and implemented our mechanism in Windows Azure platform, and have evaluated it using both simulations and a real scientific application MODIS. Evaluation results show that our mechanism can provision enough instances to meet user deadline performance goals. Even in the cases of dynamic deadline change or sudden workload surge, it can well adapt to the outside behaviors. More than 90% percent of submitted jobs can meet the deadline. In our solution, integer programming is used to identify the most cost-effective instance types based on the job composition information of incoming workload, and therefore, our approach can incur less cost compared to fixed instance type choices. The cost comparison shows that choosing appropriate instance type can save 20% - 45% compared to fixed instance types and incur 15% more compared to the optimal cost. MODIS evaluation shows that VM startup delay plays quite an important role in cloud auto-scaling mechanisms. Long unexpected VM startup delay could not only affect the performance, but can also dominate the utilization rate, and therefore the cost, especially for short deadline cases. Workload and job processing time are also very important factors in our mechanism, because these two directly affect the number and type of provisioned instances. We use history repository to improve their preciseness in our implementation.

In the future, one extension of our work is to support job class level deadlines and extend cloud application performance model into multi-tier architecture. By considering job class individually and controlling its execution instance, better performance can be achieved through running jobs on the most cost-effective instance types and save more money than fair job distribution. Currently, we are trying to use multiple queues to submit jobs by class. In multi-tier application environment, the amount of resources needed to achieve their QoS goals might be different at each tier and may also depend on availability of resources in other tiers. In both cases, a global view of the application is needed to generate optimized resource provisioning plans. Second, including on-demand pay-as-you-go instances, clouds now offer other types of instances as well, such as spot instances and reserved instances. Spot instances cost around 1/3 of regular instance prices, e.g., the average price of a m1.small spot instance is 3 cents an hour. It costs 8.5 cents an hour for the same type of on demand instance. The cheaper cost comes from that cloud providers can automatically shutdown users' spot instances if the spot price is above predefined bid price. Reserved instances are even cheaper in the long run by paying a contract fee in advance. Complexities are added if cloud auto-scaling consider these cheaper instances. Because based on our experiences, spot instances take even longer and more non-deterministic time to start. Auto-scaling controller needs to consider all these factors to make a VM instance scheduling decision. To maintain service availability, reserved instances can be considered as the always running instances. The other direction we are working on is workflow execution in Cloud. In this paper, we model the workload as submitted jobs in a queue. The cost-saving VM startup plan can only be considered during an interval instead of globally, because users can never know the future workload in advance. In workflow context, however, it is different. Users can foresee all the jobs and their decencies; therefore, a globally optimized VM startup plan can be generated. Besides, data movement cost could make it a more interesting problem. We also consider extending our evaluations to other real applications, like well-known internet workload traces, to see how our mechanism works in different workload contexts.

REFERENCES

[1] AWS auto-scaling. http://aws.amazon.com/autoscaling/

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

[3] RightScale. http://www.rightscale.com

[4] M. Assuncao et al., Evaluating the Cost-Benefit of Using Cloud Computing to Extend the Capacity of Clusters, 18th ACM International Symposium on High performance Distributed Computing (HPDC 2009), pp. 141-150.

[5] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey, Early Observations on the Performance of Windows Azure, 1st workshop on Scientific Cloud Computing, 2010.

[6] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat, Model-Based Resource Provisioning in a Web Service Utility, in Proceedings of the USENIX Symposium on Internet Technologies and Systems, 2003.

[7] J. Li, D. Agarwal, M. Humphrey, C. Ingen, K. Jackson, Y. Ryu, eScience in the Cloud: A MODIS Satellite Data Reprojection and Reduction Pipeline in Windows Azure Platform, IPDPS, 2010

[8] Trieu C. Chieu, Ajay Mohindra, Alexei A. Karve, Alla Segal: Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment. ICEBE 2009: 281-286

[9] H. Lim, S. Babu, J. Chase, and S. Parekh. Automated Control in Cloud Computing: Challenges and Opportunities. In 1st Workshop on Automated Control for Datacenters and Clouds, June 2009.

[10] P. Padala, K. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive Control of Virtualized Resources in Utility Computing Environments. EuroSys, 2007

[11] Microsoft Solver Foundation. http://code.msdn.microsoft.com/solver foundation

[12] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier internet applications. ICAC, 2005.

[13] B. Rountree, D. Lowenthal, S. Funk, V. Freeh, B. Supinski, and M. Schulz, Bounding energy consumption in large-scale mpi programs. SC 2007, November 10-16, 2007.

[14] V Swaminathan. and K. Chakrabarty. Real-time task scheduling for energy-aware embedded systems. In IEEE Real-Time Systems Symposium, November 2000.