A Reliable and Cost-Efficient Auto-Scaling System for Web Applications Using Heterogeneous Spot Instances

Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya
Cloud Computing and Distributed Systems (CLOUDS) Laboratory,
Department of Computing and Information Systems,
The University of Melbourne, Australia

September 17, 2015

Abstract

Cloud providers sell their idle capacity on markets through an auction-like mechanism to increase their return on investment. The instances sold in this way are called spot instances. In spite that spot instances are usually 90% cheaper than on-demand instances, they can be terminated by provider when their bidding prices are lower than market prices. Thus, they are largely used to provision fault-tolerant applications only. In this paper, we explore how to utilize spot instances to provision web applications, which are usually considered availability-critical. The idea is to take advantage of differences in price among various types of spot instances to reach both high availability and significant cost saving. We first propose a fault-tolerant model for web applications provisioned by spot instances. Based on that, we devise novel auto-scaling polices for hourly billed cloud markets. We implemented the proposed model and policies both on a simulation testbed for repeatable validation and Amazon EC2. The experiments on the simulation testbed and the real platform against the benchmarks show that the proposed approach can greatly reduce resource cost and still achieve satisfactory Quality of Service (QoS) in terms of response time and availability.

1 Introduction

There are three common pricing models in current Infrastructure-as-a-service (IaaS) cloud providers, namely *on-demand*, in which acquired virtual machines (VMs) are billed according to amount of time consumed and predefined unit cost of each VM type, *reservation*, where users pay an amount of up-front fee for each VM to secure availability of usage and cheaper price within a certain contract period, and the *spot*.

The spot pricing model was introduced by Amazon to sell their spare capacity in open market through an auction-like mechanism. The provider dynamically sets the market price of each VM type according to real-time demand and supply. To participate in the market, a cloud user needs to give a bid specifying number of instances for the type of VM he wants to acquire and maximum unit price he can pay. If the bidding price exceeds the current market price, the bid is fulfilled. After getting the required spot VMs, the user only pays the current market prices no matter how much he actually bids, which results in significant cost saving compared to VMs billed in on-demand prices (usually only 10% to 20% of the latter) [1]. However, obtained spot VMs will be terminated by cloud provider whenever their market prices rise beyond the bidding prices.

Such model is ideal for fault-tolerant and non-time-critical applications such as scientific computing, big data analytics, and media processing applications. On the other hand, it is generally believed that applications, like web applications, for which continuity of service and stability of Quality of Service (QoS) are considered extremely important, are not appropriate to be deployed on spot instances.

Adversely in this paper, we illustrate that, with carefully designed policies, it is also possible to reliably scale web applications using spot instances to reach both high QoS and significant cost saving in the same time.

Spot market is similar to a stock market that, though possibly following general trends, each listed item has its distinctive market behaviour according to its own supply and demand. In this kind of market, often differences in price appear with some types of instances sold in expensive prices due to high demand, while some remaining unfavoured leading to attractive deals. Taking Amazon EC2 spot market [1] as an example, a period of whose

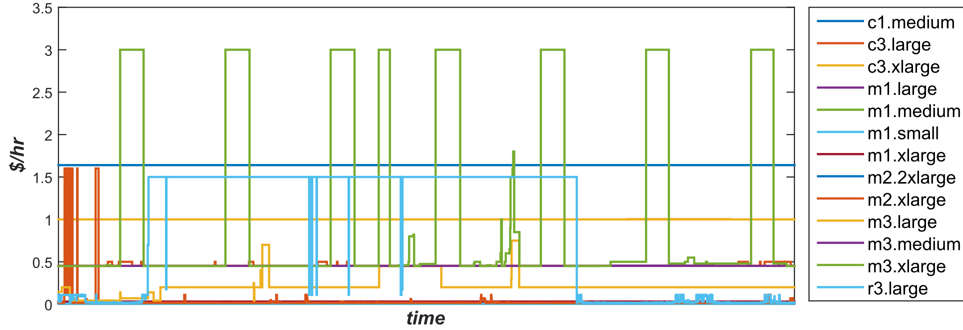


Figure 1: One week spot price history from March 2nd 2015 18:00:00 GMT in Amazon EC2's ***us-east-1d*** Availability Zone

market history is shown in Figure 1, the differences in price are notable during all the recorded period. By exploiting the diversity in this market, cloud users can utilize spot instances as long as possible to further reduce their cost. Recently, Amazon introduced the Spot Fleet API [2], which allows users to bid for a pool of resources at once. The provision of resources is automatically managed by Amazon using combination of spot instances with lowest cost. However, it still lacks fault-tolerant capability to avoid availability and performance impact caused by sudden termination of spot instances, and thus, is not suitable to provision web applications.

To fill in this gap, we aim to build a solution to cater this need. We proposed a reliable auto-scaling system for web applications using heterogeneous spot instances along with on-demand instances. Our approach not only greatly reduces financial cost of using cloud resources, but also ensures high availability and low response time even when some types of spot VMs are terminated unexpectedly by cloud provider simultaneously or consecutively within a short period of time.

The **main contributions** of this paper are:

- a fault-tolerant model for web applications provisioned by spot instances;
- reliable auto-scaling policies using heterogeneous spot instances;
- prototype implementations of the proposed policies on CloudSim [3] and Amazon EC2 platform;
- performance evaluations through both repeatable simulation studies based on historical data and real experiments on Amazon EC2;

The remainder of the paper is organized as follows. We first model our problem in Section 2. In section 3, we propose the base auto-scaling policies using heterogeneous spot instances under hourly billed context. Section 4 explains the optimizations we proposed on the initial polices. Section 5 briefly introduces our prototype implementations. We present and analyze the results of the performance evaluations in Section 6 and discuss the related works in Section 7. Finally, we conclude the paper and vision our future work.

2 System Model

For reader's convenience, the symbols used in this paper are listed in Table 1.

2.1 Auto-scaling System Architecture

As illustrated in Figure 2, our auto-scaling system provisions a single-tier (usually the application server tier) of an application using a mixture of on-demand instances and spot instances. The provisioned on-demand instances are homogeneous instances that are most cost-efficient regarding the application, while spot instances are heterogeneous.

Like other auto-scaling systems, our system is composed of the *monitoring* module, the *decision-making* module, and the *load balancer*. The monitoring module consists of multiple independent monitors that are responsible for fetching newest corresponding system information such as resource utilizations, request rates, spot market prices,

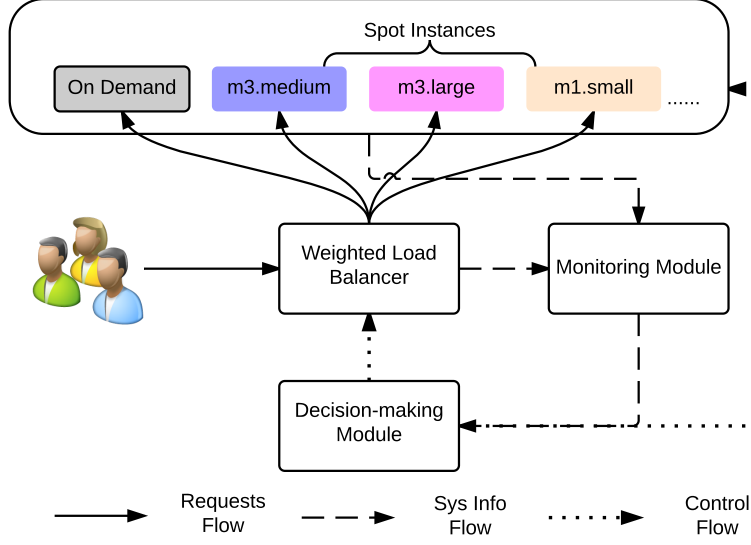
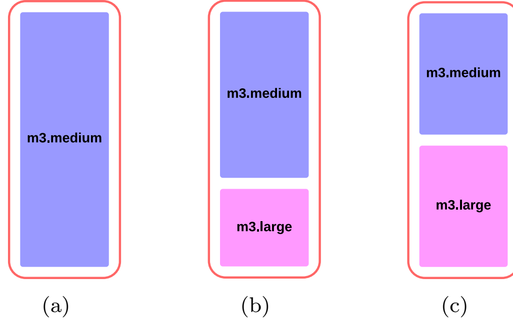


Figure 2: Proposed Auto-scaling system architecture

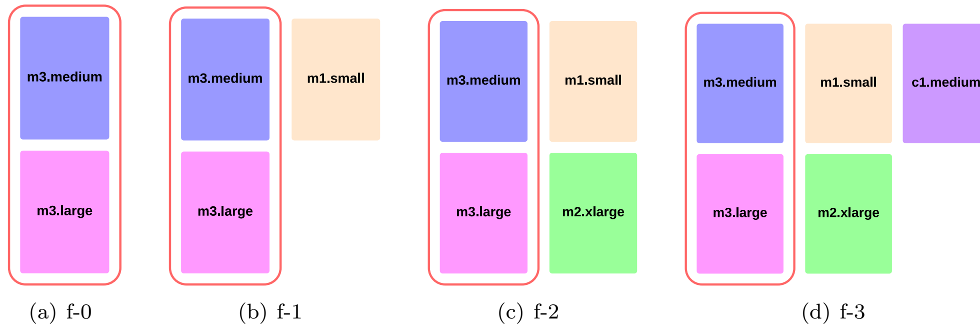
Table 1: List of Symbols

Symbol	Meaning
$\mathbf{T}$	The set of spot types
M_{min}	The minimum allowed resource margin of an instance
M_{def}	The default resource margin of an instance
Q	The quota for each spot group
R	The required resource capacity for the current load
F_{max}	The maximum allowed fault-tolerant level
f	The specified fault-tolerant level
O	The minimum percentage of on-demand resources in the provision
S	The maximum number of selected spot groups in the provision
r_o	The resource capacity provisioned by on-demand instances
s	The number of chosen spot groups
vm	The VM type
vm_o	The on-demand VM type
c_{vm}	The hourly on-demand cost of the vm type instance
$num(c, vm)$	The function returns the number of vm type instances required to satisfy resource capacity c
C_o	The hourly cost of provision in on-demand mode
tb_{vm}	The truthful bidding price of vm spot group
m	The dynamic resource margin of an instance



(a) (b) (c)

Figure 3: Naive provisioning using spot instances



(a) f-0 (b) f-1 (c) f-2 (d) f-3

Figure 4: Provisioning for different fault-tolerant levels

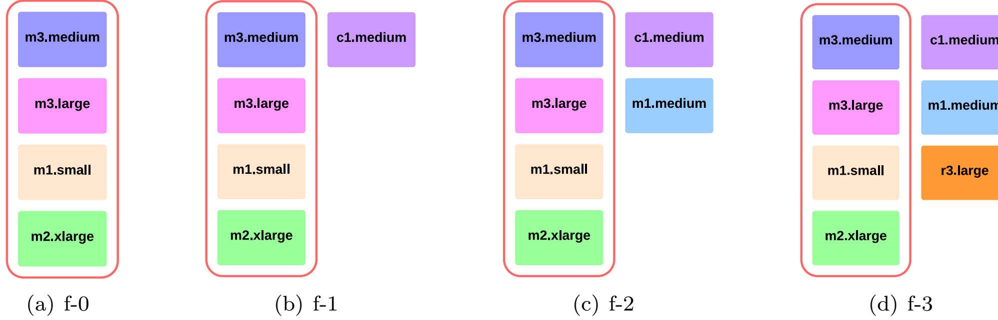
and VMs' statuses into the system. The decision-making module then makes scaling decisions according to obtained information based on the predefined strategies and policies when necessary. Since in our proposed system provisioned virtual cluster is heterogeneous, the load balancer should be able to distribute requests according to capability of each attached VM. The common algorithm used in this case is *weighted round robin*.

2.2 Fault-Tolerant Mechanism

Suppose there are sufficient temporal gaps between price variation events of different types of spot VMs, increasing spot heterogeneity in provision can improve robustness. As illustrated in Figure 3(a), the application is fully provisioned using *m3.medium* spot VMs only, which may lead it to losing 100% of its capacity when *m3.medium*'s market price go beyond the bidding price. By respectively provisioning 70% and 30% of the total required capacity using *m3.medium* and *m3.large* spot VMs in Figure 3(b), it will lose at most 70% of its processing capacity when the price of either chosen type rises above the bidding price. Furthermore, if it is provisioned equal capacity using the two types of spot VMs, like in Figure 3(c), termination of the either type of VMs will only cause it to lose 50% of its capacity.

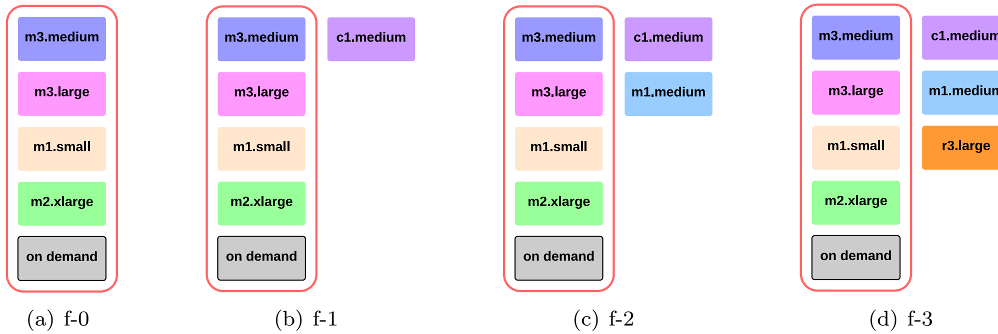
This is still unsatisfactory as we demand application performance intact even when unexpected termination happens. Simply, the solution is to further over-provision the same amount of capacity using another type of spot VMs, as the example illustrated in Figure 4(b), it can be 50% of the required capacity provisioned using *m1.small* instances. In this way, the application is now able to tolerate termination of any involving type of VMs and remain fully provisioned. After detection of the termination, the scaling system can either provision the application using another type of spot VMs or switch to on-demand instances. Application performance will be intact if there is no other termination happens before the scaling operation that repairs the provision fully completes.

However, it takes quite a long time to acquire and boot a VM (around 2 minutes for on-demand instances and 12 minutes for spot instances [4]). Hence, there is substantial possibility that another type of spot VMs could be terminated within this time window. To counter such situation, it requires further over-provision the application



(a) f-0 (b) f-1 (c) f-2 (d) f-3

Figure 5: Provisioning for different fault-tolerant levels using 2 more spot types



(a) f-0 (b) f-1 (c) f-2 (d) f-3

Figure 6: Provisioning for different fault-tolerant levels using mixture of on-demand and spot instances

using extra types of spot VMs. We define the ***fault-tolerant level*** of our auto-scaling system as the maximum number of spot types that can be unexpectedly terminated without affecting application performance before its provision can be fully recovered. Figure 4 respectively shows the provision examples that satisfy fault-tolerant level zero, one, two, and three in our definition with each spot type provisioning 50% of the required capacity.

Note that setting fault-tolerant level to zero is usually not recommended. Though using multiple types of spot instances confines amount of resource loss when failures happen, with no over-provision to compensate resource loss, it may frequently cause performance degradations as failure probability becomes higher when more types of spot instances are involved.

2.3 Reliability and Cost Efficiency

Though the provisions shown in Figure 4(b), 4(c), and 4(d) successfully increase reliability of the application, they are not cost-efficient. The three provisions respectively over-provision 50%, 100%, and 150% of resources required by the application, which greatly diminishes the cost saving of using spot instances.

One possible improvement is to provision the application using more number of spot types. The illustrative provisions in Figure 5 employ two more spot types than that are used in Figure 4 to reach the corresponding fault-tolerant levels. As the result, total over-provisioned capacities for the three cases are reduced to 25%, 50%, and 75%. Though the provisions now might become more volatile with more types of spot VMs involved, the increased risk is manageable by the fault-tolerant mechanism with over-provision.

To reduce over-provision, the other choice is to provision the application with a mixture of on-demand instances and spot instances. Like the demonstrations shown in Figure 6, there are now only 20%, 40%, and 60% over-provisioned capacities if 20% of the required resource capacity is provisioned by on-demand instances. Moreover, using on-demand resources also further confines amount of capacity that could be lost unexpectedly, thus, improving robustness. On the other hand, this method increases financial cost.

We define total capacity that is provisioned by the same type of spot VMs as a ***Spot Group***. In addition to

that, we give definition to ***Quota*** (Q), which is the capacity each spot group needs to provision given the capacity provisioned by on-demand resources (r_o) and the fault-tolerant level (f). It is calculated as:

$$Q = \frac{R - r_o}{s - f} \tag{1}$$

where R represents the required capacity for the current load, and s denotes the number of chosen spot types. The minimum amount of capacity that is required to over-provision then can be calculated as $Q * f$.

We call a provision is ***safe*** if the provisioned capacity of each spot group is larger than Q. Hence, the problem of scaling web applications using heterogeneous spot VMs is transformed to dynamically selecting spot VM types and provisioning corresponding spot and on-demand VMs to keep the provision in safe state with minimum cost when the application workload increases, and timely deprovisioning various types of VMs when they are no longer necessary.

3 Scaling Policies

Based on the previous fault-tolerant model, we propose auto-scaling policies for hourly billed cloud market like Amazon EC2.

3.1 Capacity Estimation and Load Balancing

Our auto-scaling system is aware of multiple resource dimensions (such as CPU, Memory, Network, and Disk I/O). It needs profile of the target application regarding its average resource consumption for all the considered dimensions. Currently, the profiling needs to be performed offline, but our approach is open to integrate dynamic online profiling into it.

With the profile, we can estimate the processing capability of each spot type under the context of the scaling application. Based on that, we can easily determine how to distribute incoming requests to the heterogeneous VMs to balance their loads, in other words, the relative weight of each instance that should be assigned to the load balancer. The estimated capabilities are used in the calculation of scaling plans as well.

3.2 Spot Mode and On-Demand Mode

Our scaling system runs interchangeably in ***Spot Mode*** and ***On-Demand Mode***. Spot Mode provisions application in the way explained in Section 2.3. In Spot Mode, user needs to specify the minimum percentage of required resources provisioned by on-demand instances, symbolized as O. He can also set a limit on the number of selected spot groups in provision, denoted as S. In On-Demand Mode, application is fully provisioned by on-demand instances without over-provision. Switch of modes is dynamically triggered by the scaling policies detailed in the following sections.

3.3 Truthful Bidding Prices

Bidding truthfully means the participant in an auction always bids the maximum price he is willing to pay. In order to guarantee cost-efficiency, truthful bidding price for each VM type in our policies is calculated dynamically according to real-time workload and provision. Before computing them, we first calculate the hourly baseline cost if the application is provisioned in On-Demand Mode, which can be represented as:

$$C_o = num(R, vm_o) * c_{vm_o} \tag{2}$$

where function $num(R, vm_o)$ returns the minimum number of on-demand VM type (vm_o) instances required to satisfy the capacity that is capable processing the current workload (R). c_{vm_o} is the on-demand hourly price of on-demand instance type. Then truthful bidding price of spot type vm (tb_{vm}) is derived as follow:

$$tb_{vm} = \frac{C_o - num(r_o, vm_o) * c_{vm_o}}{s * num(Q, vm)} \tag{3}$$

where $num(r_o, vm_o)$ and $num(Q, vm)$ are interpreted similarly to $num(R, vm_o)$ in Equation (2).

This ensures that even in the worst situation that all chosen spot types' market prices are equal to their corresponding truthful bidding prices, total hourly cost of the provision will not exceed that in On-Demand Mode.

Algorithm 1: Find new provision when the system needs to scale up

Input: R : the current workload
Input: n_c : the number of on-demand VMs in current provision
Input: vm_o : the on demand vm type
Input: O : the minimum percentage of on-demand resources
Output: *target_provision*

```
1 min_vm_o ← max(n_c, num(R * O, vm_o));
2 max_vm_o ← num(R, vm_o);
3 candidate_set ← call Algorithm 2 for each integer n in [min_vm_o, max_vm_o];
4 return on-demand provision if candidate_set is empty
5 otherwise the provision with minimum cost in candidate_set;
```

Algorithm 2: Find provision given the number of on-demand instances

Input: n : the number of on-demand VMs
Input: g_c : the set of spot groups in current provision
Input: vm_o : the on-demand vm type
Input: f : the fault-tolerant level
Input: $\mathbf{T}$: the set of spot types
Input: S : the maximum number of chosen spot groups
Output: *new_provision*

```
1  min_groups ← max(|g_c|, f + 1);
2  max_groups ← min(|T|, S);
3  if max_groups < min_groups then
4  |   provision not found;
5  end
6  else
7  |   for s from min_groups to max_groups do
8  |   |   p ← p ∪ (vm_o, n);
9  |   |   compute Q using Equation (1);
10 |   |   compute tb_vm for each vm in T;
11 |   |   p ← p ∪ g_c;
12 |   |   groups ← each group not in g_o and whose tb_vm is higher than market price;
13 |   |   k ← s − |g_c|;
14 |   |   if |groups| ≥ k then
15 |   |   |   p ← p ∪ top k cheapest groups in groups;
16 |   |   |   provisions ← provisions ∪ p;
17 |   |   end
18 |   end
19 end
20 return the cheapest provision in provisions;
```

3.4 Scaling Up Policy

Scaling up policy is called when some instances are terminated unexpectedly or the current provision cannot satisfy resource requirement of the application. By resource requirement, in Spot Mode, it means the provision should be ***safe*** under the current workload, which is defined in Section 2.3. While in On-Demand Mode, it only requires that

Algorithm 3: Find target provision when the billing hour of one on-demand instance is about to end

Input: R : the current workload
Input: n_c : the number of on-demand instances in current provision
Input: vm_o : the on-demand vm type
Input: O : the minimum percentage of on-demand resources
Output: $target_provision$

```
1 if n_c ≤ num(R * O, vm_o) then
2 |   provision not found;
3 end
4 else
5 |   p_1 ← call Algorithm 2 with n_c;
6 |   p_2 ← call Algorithm 2 with n_c − 1;
7 |   return on-demand provision if neither p_1 nor p_2 is found otherwise either provision that is cheaper;
8 end
```

resource capability of the provision should be able to process the current workload.

Algorithm 1 is used to find the ideal new provision when the system needs to scale up. To avoid frequent drastic changes, the algorithm only provisions VMs incrementally. As shown by line 1 in Algorithm 1, it limits the number of on-demand instances provisioned to be at least the maximum of the number of current provisioned on-demand instances and the minimum number of on-demand instances required to satisfy the on-demand capacity threshold (O). For each valid number of on-demand instances, it calls Algorithm 2 to find the corresponding best provision among provisions with various combinations of spot groups. Similarly, in Algorithm 2 (line 11), it retains the spot groups chosen by the current provision and only incrementally adds new groups according to their cost-efficiency (line 15). If there is no valid provision found, the system switches to on-demand mode.

After the target provision is found, the system compares it with the current provision and then contacts the cloud provider through its API to provision the corresponding types of VMs that are in short.

3.5 Scaling Down Policy

Since each instance is billed hourly, it is unwise to shut down one instance before its current billing hour matures. We therefore put the decision of whether each instance should be terminated or not at the end of their billing hours. The specific decision algorithms are different for on-demand instances and spot instances.

3.5.1 Policy for on-demand instances

When one on-demand instance is at the end of its billing hour, we not only need to decide whether the instance should be shut down, but also have to make changes to the spot groups if necessary. The summarized policy is abstracted in Algorithm 3. The algorithm first checks whether enough on-demand instances are provisioned to satisfy the on-demand capacity limit (line 1 and line 2). If there are sufficient on-demand instances, it endeavours to find the most cost-efficient provisions with and without the on-demand instance by calling Algorithm 2 (line 5 and line 6). Suppose the current provision is in On-Demand Mode and no provision is found without the on-demand instance, the provision will remain in On-Demand Mode. Otherwise, if a new provision is found without the current instance, the policy switches the provision to Spot Mode. In the case that the current provision is already in Spot Mode, it picks whichever provision that has lower hourly cost.

3.5.2 Policy for spot instances

When dealing with spot instance whose billing period is ending, in the base policy, we simply shut down the instance when the corresponding spot quota Q can be satisfied without it. Thereafter, the policy will evolve with the introduced optimizations in Section 4.

3.6 Spot Groups Removal Policy

Note that in both scaling up and down policies, we forbid removing selected spot groups from provision. Instead, we evict a chosen spot group when any spot instances of such type is terminated by the provider. Since bidding price of each instance is calculated dynamically, instances within the same spot group may be bid at different prices. This could cause some instances to remain functional even after the corresponding spot groups are removed from provision. We call these instances ***orphans***. Though orphan instances are still in production, they are not enlisted in capacity when making scaling decisions. In the base policies, they are shut down when their billing hour ends. This drawback is addressed by the introduced optimizations in the following section.

4 Optimizations

We have made several optimizations on the above proposed base policies to further improve cost-efficiency and reliability of the system.

4.1 Bidding Strategy

In the scaling policies, spot groups are bid at truthful bidding prices calculated by Equation (3) due to cost-efficiency concern. While focusing on robustness, the system can employ a different strategy to bid higher so as to grasp spot instances as long as possible.

4.1.1 Actual Bidding Strategies

There are two actual bidding strategies, namely truthful bidding strategy and on-demand price bidding strategy embedded in the system.

- **Truthful Bidding Strategy:** the system always bids the truthful bidding price calculated by Equation (3) when new spot instances are launched. Since partial billing hours ended by cloud provider are free of charge, cloud users can save money by letting cloud provider terminate their spot instances once their market prices exceed the corresponding truthful bidding prices. On the down side, it let the system suffer from more unexpected terminations.
- **On-Demand Price Bidding Strategy:** the system always bids on-demand price of the spot instance type when trying to obtain corresponding new spot instances. It will cost cloud users more money but provides a higher level of protection against unexpected terminations.

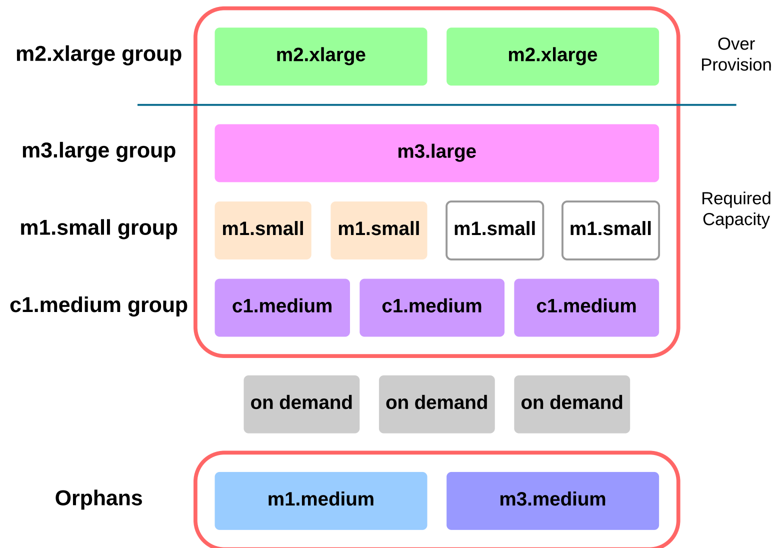
4.1.2 Revised Spot Groups Removal Policy

In the base policies, less cost-efficient spot groups could remain in provision for a long time unless some of their instances are terminated by provider. When the actual bids are higher than the truthful bidding prices, situation could become worse. Instead of just relying on provider terminating uneconomical spot groups, the revised policy actively inspects whether market prices of some spot groups have exceeded their corresponding truthful bidding prices and remove them from the provision. In the meantime, for spot groups whose market prices are still below their truthful bidding prices, it looks for chance to replace them by more economical spot groups that have not been selected. To minimize disturbance to provision, such operations should be conducted in a long interval, such as every 30 minutes in our implementation. Members of removed or replaced spot groups become orphans.

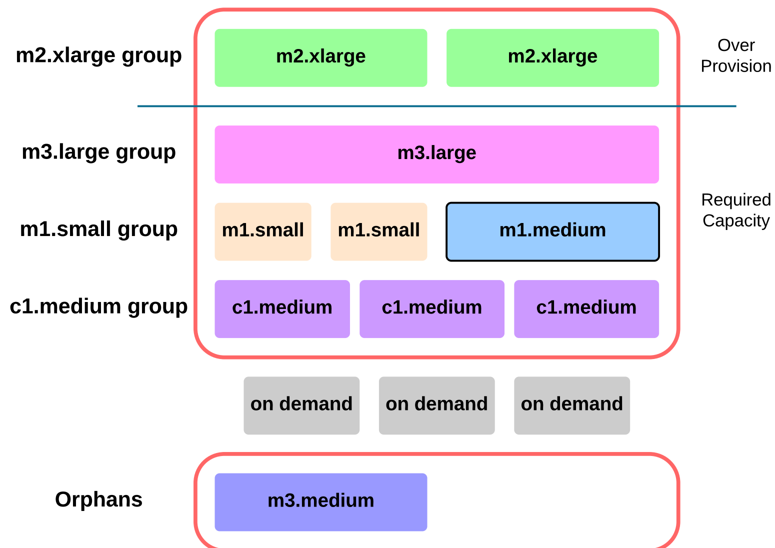
4.2 Utilizing Orphans

After removing or replacing some spot groups, if the system simply lets members of these spot groups become orphans and immediately start instances for newly chosen spot groups, the stability of provision will be affected. Furthermore, as orphans are not considered as valid capacity in the base polices, during the transition period, it has to provision more resources than necessary, which results in monetary waste.

To alleviate this problem, we aim to utilize as many orphans in provision as possible to deter the time to provision new VMs. As a result, resource waste can be reduced and cost-efficiency is improved.



(a) launching 2 new m1.small instances for m1.small spot group



(b) using one m1.medium orphan to temporarily substitute 2 m1.small instances

Figure 7: Provisioning with orphans under fault-tolerant level one

We modify the proposed fault-tolerant model to allow a spot group temporarily accept instances that are heterogeneous to the spot group type under certain conditions. Figure 7 illustrates such provision. In Figure 7(a), the *m1.small* group does not have sufficient instances to satisfy its quota. Instead of launching 2 new *m1.small* spot instances, the policy now temporarily move the available orphan, one *m1.medium* instance, to *m1.small* group to compensate the deficiency of its quota. Even though *m1.small* group becomes heterogeneous in this case, it does not violate the fault-tolerant semantic that losing any type of spot instances will not influence the application performance. However, in some situations, the heterogeneity in spot groups could cause violation of the fault-tolerant semantic, for example, there might be case that three *m1.medium* orphans are spread across three spot groups and the total capacity of the three instances exceeds the spot quota. Then losing the three *m1.medium* instances will violate the fault-tolerant semantic. Fortunately, such cases are very rare as orphans are usually small in numbers and are expected to be shut down in a short time.

With this relaxation of the fault-tolerant model, the previous scaling up and scaling down policies need to be revised to efficiently utilize capacity of orphans.

4.2.1 Revised Scaling Up Policy

The new scaling up policy uses the same algorithm (Algorithm 1) to find the target provision. However, instead of simply launching instances to reach the target provision, the new policies take a deeper thought whether it can utilize existing orphans to meet the quota requirements in the target provision.

The new policy first checks whether the target provision chooses new spot groups. If there are orphans that are either within orphan queue or other spot groups, whose types are the same to the newly chosen groups, they are immediately moved to the corresponding new spot groups. After that, the policies endeavour to insert non-utilized orphans from the orphan queue into spot groups that have not met their quota requirement. If all the orphans have been utilized and some groups still cannot satisfy their quota, new spot instances of the corresponding types then will be launched.

4.2.2 Revised Scaling Down Policy

Regarding policy for on-demand instances that are close to their billing hour, the new policy utilizes the same mechanism in the revised scaling up policy to provision any changes between the current provision and the target provision.

For spot scaling down policy, if the spot instance is in orphan queue, it is immediately shut down. Suppose it is within the spot group of the same type, it is shut down when the spot quota can be satisfied without it. In the case that the instance is an orphan within other spot group, the new policy shuts down the instance and in the meantime starts certain number of spot instances of the spot group type to compensate the capacity loss.

4.3 Reducing Resource Margin

For applications running on traditional auto-scaling platform, administrator usually leaves a margin at each instance to handle short-term workload surge and provide a buffer to buy time for booting up new instances. This margin empirically ranges from 20 to 25% of the instance's capacity.

With over-provision already in place in our system, this margin can be reduced under Spot Mode provision. We devise a mechanism that dynamically changes the margin according to current fault-tolerant level. Since higher fault-tolerant level leads to more over-provision, we can be more aggressive in reducing the margin of each instance. In detail, the dynamic margin is determined by the formula:

$$m = \frac{M_{def} - M_{min}}{F_{max}} * f + M_{min} \tag{4}$$

where M_{min} means the minimum allowed margin, e.g., 10%, M_{def} is the default margin used without dynamic margin reduction, e.g., 25%, and F_{max} is the maximum allowed fault-tolerant level.

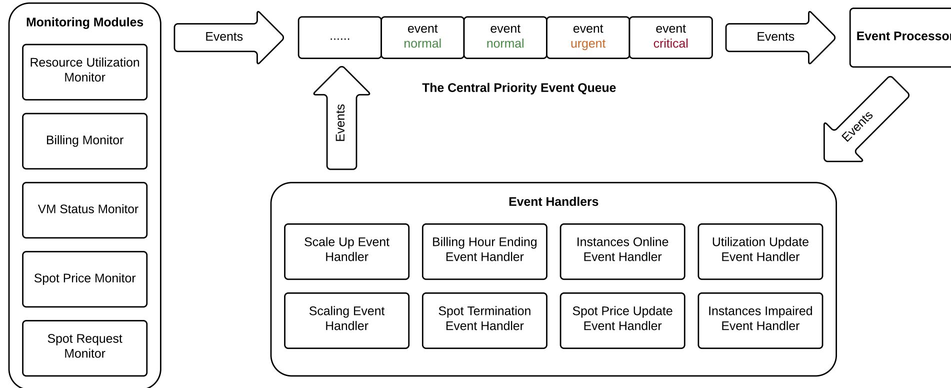


Figure 8: Components of the Implemented Auto-scaling System

5 Implementation

We implemented a prototype of the proposed auto-scaling system on Amazon EC2 platform using Java, the components of which are illustrated in Figure 8. It employs an event-driven architecture with the monitoring modules continuously generating events according to newly obtained information, and the central processor consuming events one by one. Monitoring modules produce corresponding events with predefined critical levels to the central priority event queue. They include the *resource utilization monitors* that watch all dimensions of resource consumption of running instances, the *billing monitor* that gazes billing hour of each requested VM, the *VM status monitor* that reminds the system that some instances are online or offline, the *spot price monitor* that records newest spot market prices for each considered spot type, and *the spot request monitor* that surveillances any unexpected spot termination. On the other side, the central event processor fetches events from the event queue and assigns them to the corresponding event handlers that realize the proposed policies to make scaling decisions or perform scaling actions.

The implementation provides a general interface for users to plug different load balancer solutions into the auto-scaling system. For our case, we use *HAProxy* with weighted round robin algorithm. We also implemented the interface to allow users to automatically customize configurations of VMs according to their own available resources after they have been booted.

For quick concept validation and repeatable evaluation of the proposed auto-scaling policies, we also created a simulation version of the system. The same code base is transplanted onto CloudSim [3] toolkit which provides the underlying simulated cloud environment. Assuming bids from user impose negligible influence on market prices, the simulation tool is able to provide quick and economical validation of the proposed polices using historic data of the application and the spot market prices as input.

For more details about the implementation, please refer to the released code[1].

6 Performance Evaluation

6.1 Simulation Experiments

As stated in Section 5, to allow repeatable evaluation, we developed a simulation version of the system that allows us to compare the performances of different configurations and policies using traces from real applications and spot markets.

6.1.1 Simulation Settings

We use one week trace of English Wikipedia from Sep 19th 2009 to Sep 26th 2009 as the workload, which is depicted in Figure 9. We consider 13 spot types in Amazon EC2. Their spot prices are simulated according to one week Amazon's spot prices history from March 2nd 2015 18:00:00 GMT in the relatively busy ***us-east*** region. The involving spot types and their corresponding history market prices are illustrated in Figure 1.

[1] `https://github.com/quchenhao/spot-auto-scaling`

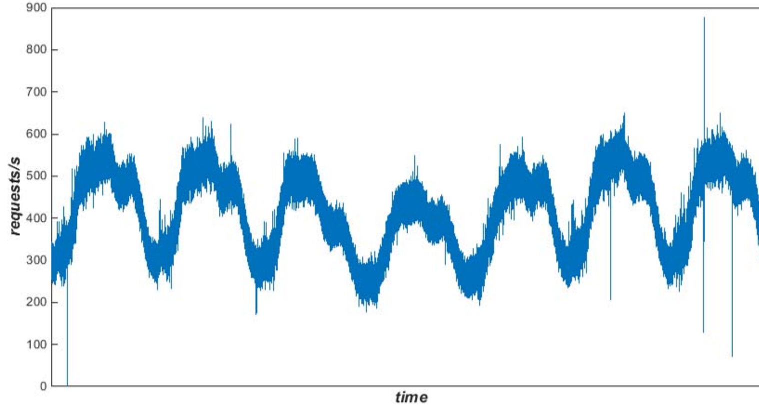


Figure 9: The English Wikipedia workload from Sep 19th 2009 to Sep 26th 2009

We set requests timeout at 30 seconds. In addition, we respectively set minimum allowed resource margin (M_{min}) and default resource margin (M_{def}) at 10% and 25%. The *c3.large* instance type, which is the most cost-efficient type for the Wikipedia application among all the considered VM types according to the resource specification released by Amazon, is selected to provision all the on-demand resources in the experiments. All simulation experiments start with 5 *c3.large* on-demand instances. Length of simulated requests are generated following a pseudo Gaussian distribution with mean of 0.07 ECU[2] and standard deviation of 0.005 ECU so that different tests using the same random seed are receiving exactly the same workload. The VM start up, shut down, and spot requesting delays are generated in the same way using pseudo Gaussian distribution. The means of the above three distributions are respectively 100, 100, 550 seconds, and the standard deviations are set at 20, 20, 50 seconds. The test results are deterministic and repeatable on the same machine.

We tested our scaling policies with various fault-tolerant levels and different least amounts of on-demand resources, which are represented respectively as "$f - x$" and "y% on-demand" in the results. We also tested the polices using the two embedded bidding strategies and static/dynamic resource margins.

We concentrate on two metrics, real-time response time of requests (average response time per second reported) and total cost of instances, in all the experiments.

6.1.2 Benchmarks

We compare our scaling policies with two benchmarks:

- **On-Demand Auto-scaling:** This benchmark only utilizes on-demand instances. It is implemented by restricting the auto-scaling system always in On-Demand Mode.
- **One Spot Type Auto-scaling:** The auto-scaling policies used in this benchmark, like the proposed policies, provision a mixture of on-demand resources and spot resources. The benchmark also has a limit on minimum amount of on-demand resources provisioned. However, for spot instances, it only provisions one spot group that is the most cost-efficient at the moment without over-provision. If the provisioned spot instances are terminated, a new spot group then is selected and provisioned. Suppose a more economic spot group is found, the old spot group is gradually replaced by the new one. It is implemented by setting fault-tolerant level to zero and limiting at most one spot group can be provisioned.

6.1.3 Response time

Figure 10, 11, 12, and 13 respectively depict real-time average response time of requests using on-demand, one spot, and our approach with truthful bidding strategy and dynamic resource margin. From the results, the on-demand auto-scaling produced smooth response time all along the experimental duration except for a peak that was resulted

[2]It means the request takes 70ms seconds to finish if it is computed by the VM equipped with vCPU as powerful as 1 Elastic Computing Unit (ECU)

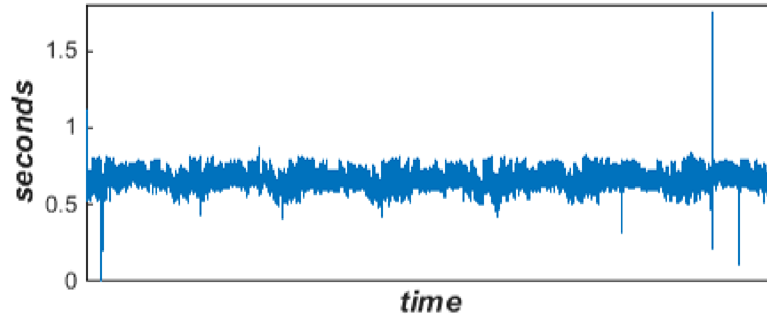
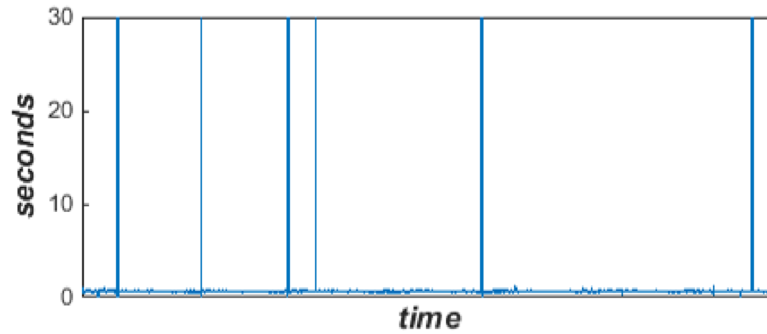
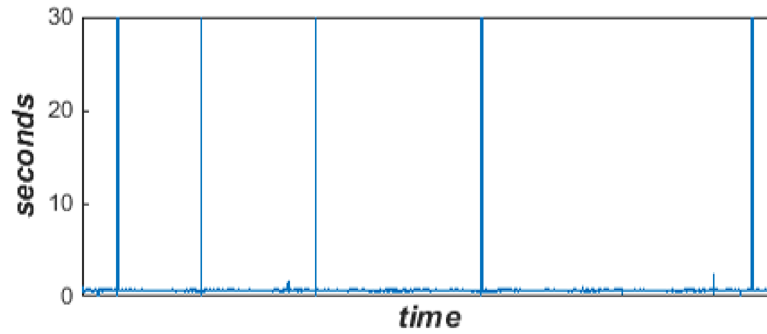


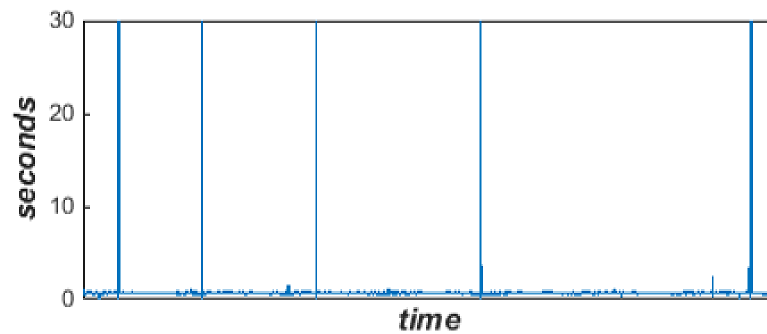
Figure 10: Response time for on-demand auto-scaling



(a) 0% on-demand resources

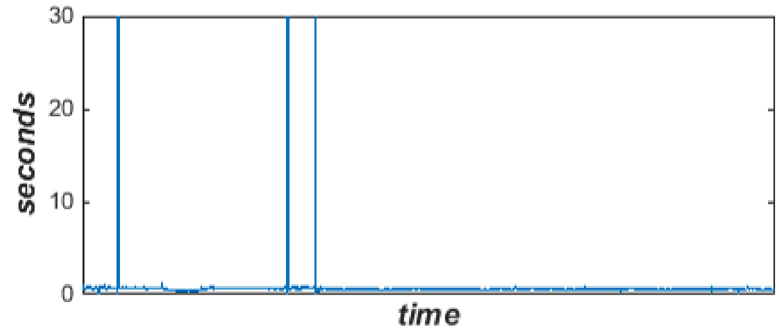


(b) 20% on-demand resources

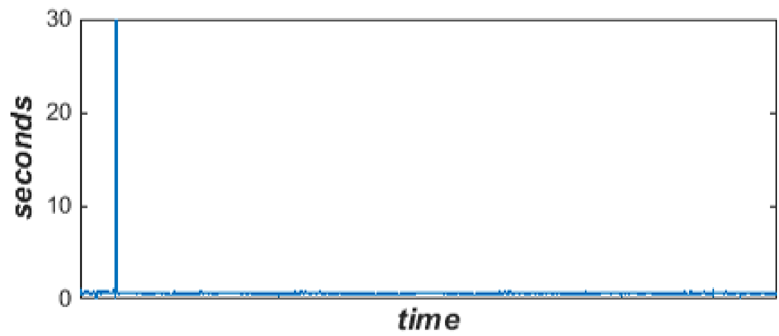


(c) 40% on-demand resources

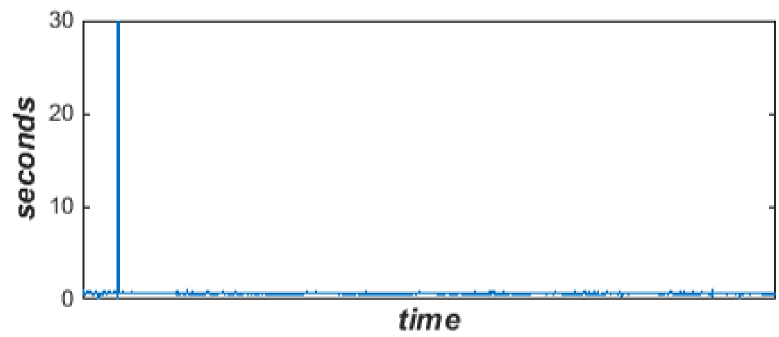
Figure 11: Response time of one spot type auto-scaling with various percentage of on-demand resources and truthful bidding strategy



(a) 0% on-demand resources

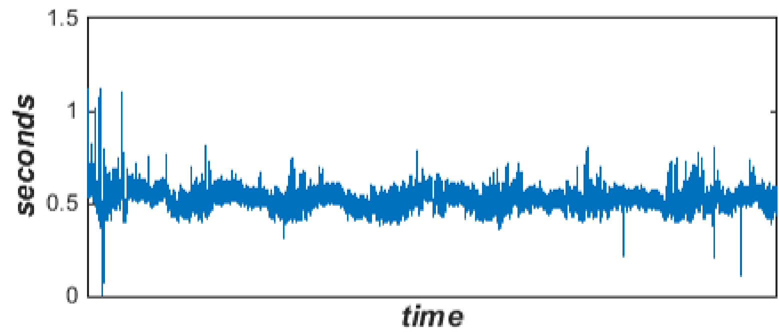


(b) 20% on-demand resources

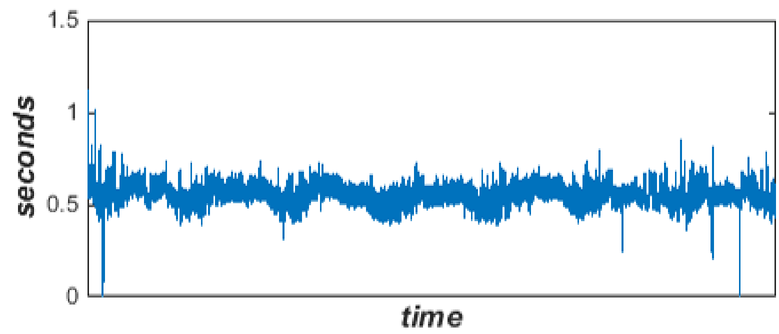


(c) 40% on-demand resources

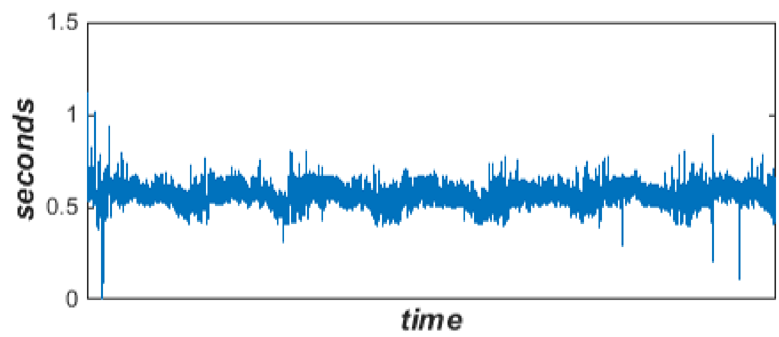
Figure 12: Response time of $f-0$ with various percentage of on-demand resources, truthful bidding strategy, and dynamic resource margin



(a) 0% on-demand resources

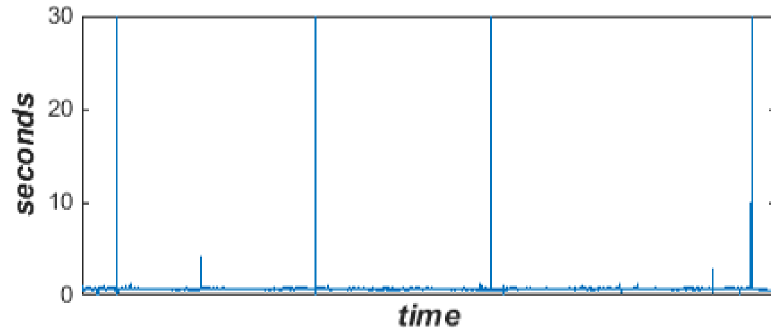


(b) 20% on-demand resources

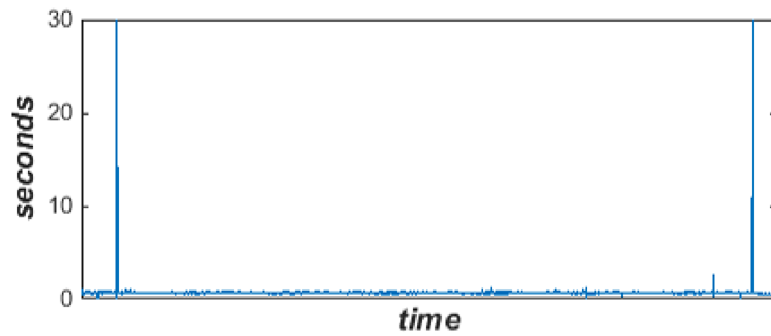


(c) 40% on-demand resources

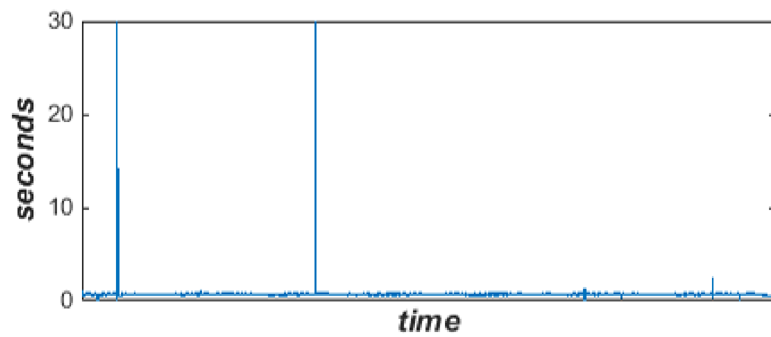
Figure 13: Response time of $f-1$ with various percentage of on-demand resources, truthful bidding strategy, and dynamic resource margin



(a) 0% on-demand resources



(b) 20% on-demand resources



(c) 40% on-demand resources

Figure 14: Response time of one spot type auto-scaling with various percentage of on-demand resources and on-demand bidding strategy

by the corresponding peak in the workload. All experiments employing one spot type auto-scaling experienced periods of request timeouts caused by termination of spot instances, and only increasing the amount of on-demand resources cloud not improve the situation. While our approach greatly reduced such unavailability of service even using $f-0$ with no over-provision of resources. By using $f-1$, we were able to completely eliminate the timeouts under the recorded spot market traces. We omit the results for tests using $f-2$ and $f-3$ as they reveal similar results as Figure 13.

To show the effect of different bidding strategies, we compare the response time results of one spot type auto-scaling using two proposed bidding strategies as they reveal the most significant difference. As Figure 11 and Figure 14 present, it is obvious that service availability can be much improved with higher bidding prices under extreme situation using one spot type auto-scaling. On the other hand, the remaining timeouts also indicate that increasing bidding prices alone is not enough to guarantee high availability.

6.1.4 Cost

Table 2 lists the total costs produced by all the experiments. Comparing to the cost of on-demand auto-scaling, we managed to gain significant cost saving using all other configurations. Tests using one spot type auto-scaling with 0% on-demand resources realized the most cost saving up to 80.87% regardless of its availability issue.

Horizontally comparing the results, the amount of on-demand resources has a significant influence on cost saving. Because our scaling polices can handle termination of spot instances, the best practice is to allow the system to reduce the amount of on-demand resources to zero so as to gain more cost saving. It also can be noted that higher fault-tolerant level incurs extra cost. Though optimal configuration of fault-tolerant level is always application specific, according to our results, configuration using $f-1$ with 0% on-demand resource is the best choice under current market situation in regards of both financial cost and service availability.

The cost differences for configurations using truthful bidding and on-demand price bidding strategies are generally minute. Therefore, it is better to bid higher to improve availability if user's bidding has negligible impact on the market price.

As dynamic resource margin is only applicable when application is over-provisioned, we give the results for tests using dynamic resource margin when fault-tolerant level is higher than zero. According to the results, dynamic resource margin can bring extra cost saving and the amount of cost saving increases when the required over-provision becomes higher (i.e., higher fault-tolerant level). Though the resulted cost saving is not significant, it is safely achieved without sacrificing availability and performance of the application.

6.2 Real Experiments

We conducted two real tests on Amazon EC2 respectively using on-demand auto-scaling policies and the proposed auto-scaling policies with configuration of $f-1$ and 0% on-demand. Other parameters are defined the same to the simulation tests.

We set up the experimental environment to run the Wikibench [5] benchmark tool. It is composed of three components:

- a client driver that mimics clients by continuously sending requests to the application server according to the workload trace;
- a stateless application server installed with the Mediawiki application;
- a mysql database loaded with the English Wikipedia data by the date of Jan 3rd, 2008.

Our aim is to scale the application-tier. Thus, we inserted a HAProxy load balancer layer into the original architecture in order to let the client driver talk to a cluster of servers. The architecture of the testbed is illustrated in Figure 15. We picked the first 3 days of the Wikipedia workload (Figure 9) and scaled it down to half of its original rate as the workload for testing because Amazon limits the number of instances each account can launch.

The testing environment resided in Amazon ***us-east-1d*** zone which is in a relatively busy region with higher degree and frequency of price fluctuations. Regarding each component, we launched one *c4.large* instance acting as the client driver, one *m3.medium* instance running the HAProxy load balancer, and one *c4.2xlarge*[3] instance serving

[3] The 4th generation instances were introduced between the time we performed the simulations and the real experiments. To be consistent, we only consider the 13 spot types listed in Figure 1 for both the simulations and the real experiments

Table 2: Total Costs for Experiments with Various Configurations

Policies \ USD$	Total Cost			
on-demand	116.34			
	Truthful Bidding		**On-Demand Bidding**	
one spot with 0% on-demand	22.26		23.14	
one spot with 20% on-demand	46.50		47.33	
one spot with 40% on-demand	63.17		63.43	
$f-0$ **with 0% on-demand**	32.00		32.30	
$f-0$ **with 20% on-demand**	54.45		56.10	
$f-0$ **with 40% on-demand**	68.34		69.64	
	Static Resource Margin	**Dynamic Resource Margin**	**Static Resource Margin**	**Dynamic Resource Margin**
$f-1$ **with 0% on-demand**	41.57	39.32	43.17	41.66
$f-1$ **with 20% on-demand**	60.21	59.52	61.82	61.06
$f-1$ **with 40% on-demand**	72.09	72.55	72.96	73.08
$f-2$ **with 0% on-demand**	50.48	47.38	51.67	49.38
$f-2$ **with 20% on-demand**	67.72	65.52	68.71	66.09
$f-2$ **with 40% on-demand**	78.01	76.74	78.3	76.74
$f-3$ **with 0% on-demand**	67.87	62.61	68.79	61.50
$f-3$ **with 20% on-demand**	83.27	78.33	81.18	76.19
$f-3$ **with 40% on-demand**	89.86	85.57	88.09	84.46

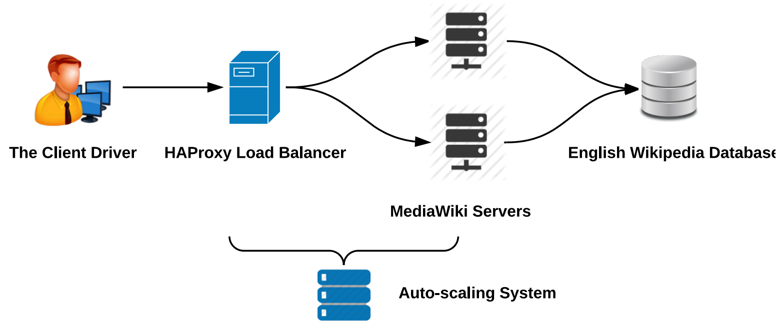


Figure 15: The Testbed Architecture

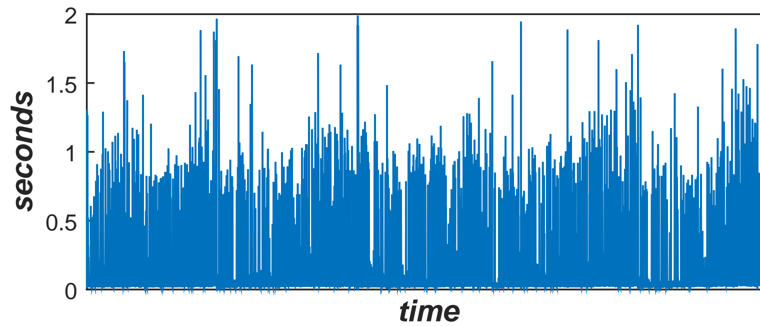


Figure 16: Response time for on-demand auto-scaling on Amazon

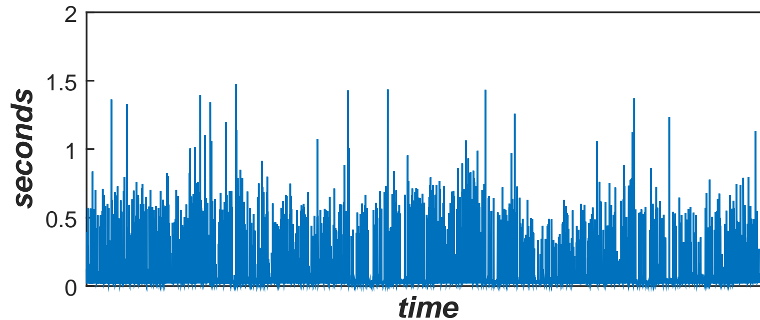


Figure 17: Response time for spot auto-scaling on Amazon

the mysql database requests. The auto-scaling system itself is running on a local desktop computer remotely in Melbourne. Before the tests, we profiled each component to make sure none of them become the bottleneck of the system.

The test using the proposed approach started at 3:30am September 9, 2015, Wednesday, US east time. Its testing period spanned across three busy weekdays from Wednesday to Friday.

Figure 16 and 17 presents real-time response time results of the two experiments. Both results suffer from peaks of high response time. By studying the recorded log, we confirmed they were not caused by shortage of resources as resource utilizations of all the involving VMs were never beyond safe threshold during both tests. Various other reasons can be the culprits, such as cold cache, short term network issues, interference from the shared virtualized environment, and garbage collection [6]. We encountered three unexpected terminations during the test of our approach. Thanks to the fault-tolerant mechanism and policies, we managed to avoid service interruption and performance degradation during those periods. In addition, because resources are tighter in on-demand auto-scaling, it generally performs worse in response time compared to the proposed approach.

Regarding cost, we calculated the total cost of application servers in both experiments. Table 3 presents the

Table 3: Cost of the Experiments

	Cost(USD$)
on-demand	19.01
$ft-1$ **and** 0% **on-demand**	5.69

results. The proposed approach reaches 70.07% cost saving.

6.3 Discussion

Even with high fault-tolerant level, the proposed approach cannot guarantee 100% availability, and no solution can ever manage to assure absolute service continuity due to the nature of spot market. What our system offers is a best effort to counter large scale surges of market prices of the selected spot types in a short time, which is highly unlikely under current market condition. In fact, we have not encountered any case that more than one spot group fail simultaneously during simulations, real experiments, and testing phases. However, market condition could change. Hence, application provider should adjust configuration of the auto-scaling system dynamically according to real-time volatility of the spot market. In addition, the nature of the application also affects the decision. If the application is availability-critical, higher fault-tolerant level is always desirable. Adversely, for some applications, such as analytical jobs, even one spot type auto-scaling is acceptable.

The presented results in Section 6 only indicates the cost saving potential of a certain application considering a selected set of spot types under the recorded spot market prices and workload traces. Thanks to the dynamic truthful bidding price mechanism, even in competitive market condition, we can ensure that the cost reduction gained by our approach will not vanish but only diminish. To reach more cost saving, the application provider can take into account a broader set of spot types, which is available in Amazon's offering.

To save cost and time for testing, application providers can tune the parameters of the auto-scaling system in a similar way as we did by first utilizing simulation for fast validation and then test the system in production environment.

There are also differences in price among the same spot types across different availability zones. It is trivial to extend the current fault-tolerant model to utilize spot groups from multiple availability zones. Currently, the auto-scaling system limits the selection of spot groups within the same availability zone due to charges of traffic across availability zones. If the application provider has already adopted a multi-availability-zone deployment, such extension is able to realize more cost saving.

7 Related Work

7.1 Auto-scaling Web Applications

Auto-scaling web applications have been extensively studied and applied. Basically, auto-scaling techniques for web applications can be classified into three categories: *reactive approaches*, *proactive approaches*, and *mixed approaches*. Reactive approaches scale applications in accordance of workload changes. Proactive approaches predicts future workload and scale applications in advance. Mixed approaches can scale applications both reactively and proactively.

Most industry auto-scaling systems are reactive-based. Among them, the most frequently used service is Amazon's Auto Scaling Service [7]. It requires user to first create an auto-scaling group, which specifies type of VMs and image to use when launching new instances. Then user should define his scaling policies as rules like "add 2 instances when CPU utilization is larger than 75%". Another popular service is offered by RightScale. Their service is based on a voting mechanism that lets each running instance decide whether it is necessary to grow or shrink the size of the cluster based on their own condition [8].

Other than just using simple rules to make scaling decisions, researchers have developed scaling systems based on formal models. These models aim to answer the question that how many resources are actually required to serve certain amount of incoming workload under QoS constraints. Such model can be simply obtained using profiling techniques as we did in this paper. Other commonly adopted approaches include queueing models [9, 10, 11] that

either abstract the application as a set of parallel queues or a network of queues, and online learning approaches such as reinformacement learning [12, 13].

Proactive auto-scaling is desirable because time taken to start and configure newly started VMs creates a resource gap when workload suddenly increases to the level beyond capability of available resources. To satisfy strict SLA, sometimes it is necessary to provision enough resources before workload actually surges. As workloads of web applications usually reveal temporal patterns, accurate prediction of future workload is feasible using state-of-art time-series analysis and pattern recognition techniques. A lot of them have been applied to auto-scaling of web applications [14, 15, 10, 16, 17].

Most auto-scaling systems only utilize homogeneous resources. While some, including our system, have explored using heterogeneous resources to provision web applications. Upendra et al. [18], and Srirama and Ostavar [19] adopt integer linear programming (ILP) to model the optimal heterogeneous resource configuration problem under SLA constraints. Fernandez et al. [20] utilizes tree paths to represent different combinations of heterogeneous resources and then searches the tree to find the most suitable scaling plan according to user's SLA requirements. Different from these works, our work focuses on using spot instances to reliably auto-scale web applications.

7.2 Application of Spot Instances

There have been a lot of attempts to employ spot instances under various application context in order to reduce resource cost. Resource provision problems using spot instances have been studied for fault-tolerant applications [21, 22, 23, 24, 25, 26, 27, 28, 29] such as high performance computing, data analytics, MapReduce, and scentific workflow.

For these applications, the fault-tolerant mechanism is often built on checkpointing. Multiple novel checkpointing mechanisms [30, 31, 32] have been developed to allow these applications to harness the power of spot instances. For web applications, checkpointing is not feasible as web requests are short-lived.

Regarding web applications, Han et al. [33] proposed a stochastic algorithm to plan future resource usage with a mixture of on-demand and spot instances. Except that they only use homogeneous resources, their problem is also different from ours as they aim to plan the resource usage with the knowledge of a horizon of future time while we provision resources dynamically without this knowledge. Mazzucco and Dumas [34] also explored using a mixture of homogeneous on-demand instances and spot instances to provision web applications. Instead of building a reliable auto-scaling system, their target is to maximize web application provider's profit by using an admission control mechanism at the front end to dynamically adapt to sudden changes of available resources.

Recently, Amazon EC2 introduced a new feature, called Spot Fleet API [2]. It allows user to bid for a fixed amount of capacity possibly constituted by instances of different spot types. It continuously and automatically provisions the capacity using the combination of instances that incurs the lowest cost. However, as its decision is based only on cost without considering reliability, it is not suitable to provision web applications.

8 Conclusions and Future Work

In this paper, we explored how to reliably and cost-efficiently auto-scale web applications using a mixture of on-demand and heterogeneous spot instances. We first proposed a fault-tolerant mechanism that can handle unexpected spot terminations using heterogeneous spot instances and over-provision. We then devised novel auto-scaling policies under hourly-billed cloud context with the defined fault-tolerant model. We implemented a prototype of the proposed auto-scaling system on Amazon EC2 and a simulation version on CloudSim [3] for repeatable and fast validation. We conducted both simulations and real experiments to demonstrate the effectiveness of our approach by comparing the results with the benchmark approaches.

In the future, we plan to further optimize our system by incorporating the following features:

- selection of spot groups according to predicted spot prices in near future;
- dynamic decision of fault-tolerant level and proportion of on-demand instances according to volatility of the spot market;
- an interface that allows web application providers to plug in different workload prediction techniques into the auto-scaling system to achieve proactive auto-scaling; and

- utilization of spot groups across different availability zones.

Acknowledgment

We thank Dr. Adel Nadjaran Toosi, Xunyun Liu, Yaser Mansouri, Bowen Zhou, and Nikolay Grozev for their valuable comments in improving the quality of the paper.

References

[1] Amazon, Amazon ec2 spot instances.
URL `http://aws.amazon.com/ec2/spot-instances/`

[2] Amazon, Amazon spot fleet api.
URL `https://aws.amazon.com/blogs/aws/new-resource-oriented-bidding-for-ec2-spot-instances/`

[3] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, R. Buyya, Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Software: Practice and Experience 41 (1) (2011) 23–50.

[4] M. Ming, M. Humphrey, A performance study on the VM startup time in the cloud, in: Proceedings of 2012 IEEE 5th International Conference on Cloud Computing (CLOUD), pp. 423–430.

[5] E.-J. van Baaren, Wikibench: A distributed, wikipedia based web application benchmark, Master's thesis, VU University Amsterdam.

[6] J. Dean, B. Luiz A., The tail at scale, Commun. ACM 56 (2) (2013) 74–80.

[7] Amazon, Auto scaling (2015).
URL `http://aws.amazon.com/autoscaling/`

[8] RightScale, Understanding the voting process (2015).
URL `https://support.rightscale.com/12-Guides/RightScale_101/System_Architecture/RightScale_Alert_System/Alerts_based_on_Voting_Tags/Understanding_the_Voting_Process/`

[9] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, T. Wood, Agile dynamic provisioning of multi-tier internet applications, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 3 (1) (2008) 1.

[10] J. Jiang, J. Lu, G. Zhang, G. Long, Optimal cloud resource auto-scaling for web applications, in: Proceedings of 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE, pp. 58–65.

[11] A. Gandhi, P. Dube, A. Karve, A. Kochut, L. Zhang, Adaptive, model-driven autoscaling for cloud applications, in: Proceedings of the 11th International Conference on Autonomic Computing (ICAC 14), USENIX Association, Philadelphia, PA, 2014, pp. 57–64.

[12] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, I. Truck, Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow, in: Proceedings of the Seventh International Conference on Autonomic and Autonomous Systems (ICAS 2011), pp. 67–74.

[13] E. Barrett, E. Howley, J. Duggan, Applying reinforcement learning towards automating resource allocation and application scalability in the cloud, Concurrency and Computation: Practice and Experience 25 (12) (2013) 1656–1674.

[14] N. Roy, A. Dubey, A. Gokhale, Efficient autoscaling in the cloud using predictive models for workload forecasting, in: Proceedings of 2011 IEEE International Conference on Cloud Computing (CLOUD), IEEE, pp. 500–507.

[15] Y. Jingqi, L. Chuanchang, S. Yanlei, M. Zexiang, C. Junliang, Workload predicting-based automatic scaling in service clouds, in: Proceedings of 2013 IEEE Sixth International Conference on Cloud Computing (CLOUD), pp. 810–815.

[16] E. Caron, F. Desprez, A. Muresan, Pattern matching based forecast of non-periodic repetitive behavior for cloud clients, Journal of Grid Computing 9 (1) (2011) 49–64.

[17] N. R. Herbst, N. Huber, S. Kounev, E. Amrehn, Self-adaptive workload classification and forecasting for proactive resource provisioning, Concurrency and Computation: Practice and Experience 26 (12) (2014) 2053–2078.

[18] S. Upendra, P. Shenoy, S. Sahu, A. Shaikh, A cost-aware elasticity provisioning system for the cloud, in: Proceedings of 2011 31st International Conference on Distributed Computing Systems (ICDCS), pp. 559–570. doi:10.1109/ICDCS.2011.59.

[19] S. N. Srirama, A. Ostovar, Optimal resource provisioning for scaling enterprise applications on the cloud, in: Proceedings of 2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom), pp. 262–271. doi:10.1109/CloudCom.2014.24.

[20] H. Fernandez, G. Pierre, T. Kielmann, Autoscaling web applications in heterogeneous cloud infrastructures, in: Proceedings of 2014 IEEE International Conference on Cloud Engineering (IC2E), pp. 195–204.

[21] S. Costache, N. Parlavantzas, C. Morin, S. Kortas, Themis: Economy-based automatic resource scaling for cloud systems, in: Proceedings of 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), pp. 367–374.

[22] C. Binnig, A. Salama, E. Zamanian, M. El-Hindi, S. Feil, T. Ziegler, Spotgres - parallel data analytics on spot instances, in: Proceedings of 2015 31st IEEE International Conference on Data Engineering Workshops (ICDEW), pp. 14–21.

[23] D. Poola, K. Ramamohanarao, R. Buyya, Fault-tolerant workflow scheduling using spot instances on clouds, Procedia Computer Science 29 (2014) 523–533.

[24] L. Sifei, L. Xiaorong, W. Long, H. Kasim, H. Palit, T. Hung, E. F. T. Legara, G. Lee, A dynamic hybrid resource provisioning approach for running large-scale computational applications on cloud spot and on-demand instances, in: Proceedings of 2013 International Conference on Parallel and Distributed Systems (ICPADS), pp. 657–662. doi:10.1109/ICPADS.2013.117.

[25] W. Voorsluys, R. Buyya, Reliable provisioning of spot instances for compute-intensive applications, in: Proceedings of 2012 IEEE 26th International Conference on Advanced Information Networking and Applications (AINA), pp. 542–549. doi:10.1109/AINA.2012.106.

[26] C. Changbing, L. Bu Sung, T. Xueyan, Improving hadoop monetary efficiency in the cloud using spot instances, in: Proceedings of 2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom), pp. 312–319.

[27] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, C. Krintz, See spot run: using spot instances for mapreduce workflows, in: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, USENIX Association, pp. 7–7.

[28] M. Zafer, S. Yang, L. Kang-Won, Optimal bids for spot vms in a cloud for deadline constrained jobs, in: Proceedings of 2012 IEEE 5th International Conference on Cloud Computing (CLOUD), pp. 75–82. doi:10.1109/CLOUD.2012.59.

[29] C. Hsuan-Yi, Y. Simmhan, Cost-efficient and resilient job life-cycle management on hybrid clouds, in: Proceedings of 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS), pp. 327–336.

[30] I. Jangjaimon, T. Nian-Feng, Effective cost reduction for elastic clouds under spot instance pricing through adaptive checkpointing, IEEE Transactions on Computers 64 (2) (2015) 396–409.

[31] Y. Sangho, A. Andrzejak, D. Kondo, Monetary cost-aware checkpointing and migration on Amazon cloud spot instances, IEEE Transactions on Services Computing 5 (4) (2012) 512–524.

[32] D. Jung, S. Chin, K. Chung, H. Yu, J. Gil, An Efficient Checkpointing Scheme Using Price History of Spot Instances in Cloud Computing Environment, Vol. 6985 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, book section 16, pp. 185–200.

[33] Z. Han, P. Miao, L. Xinxin, L. Xiaolin, F. Yuguang, Optimal resource rental planning for elastic applications in cloud market, in: Proceedings of 2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS), pp. 808–819.

[34] M. Mazzucco, M. Dumas, Achieving performance and availability guarantees with spot instances, in: Proceedings of 2011 IEEE 13th International Conference on High Performance Computing and Communications (HPCC), pp. 296–303.