Accepted, to appear in CCGrid 2012

Lightweight Resource Scaling for Cloud Applications

Rui Han, Li Guo, Moustafa M. Ghanem, Yike Guo*
Department of Computing
Imperial College London
London, UK
{r.han10, liguo, mmg, y.guo}@imperial.ac.uk

Abstract— **Elastic resource provisioning is a key feature of cloud computing, allowing users to scale up or down resource allocation for their applications at run-time. To date, most practical approaches to managing elasticity are based on allocation/de-allocation of the virtual machine (VM) instances to the application. This VM-level elasticity typically incurs both considerable overhead and extra costs, especially for applications with rapidly fluctuating demands. In this paper, we propose a lightweight approach to enable cost-effective elasticity for cloud applications. Our approach operates fine-grained scaling at the resource level itself (CPUs, memory, I/O, etc) in addition to VM-level scaling. We also present the design and implementation of an intelligent platform for light-weight resource management of cloud applications. We describe our algorithms for light-weight scaling and VM-level scaling and show their interaction. We then use an industry standard benchmark to evaluate the effectiveness of our approach and compare its performance against traditional approaches.**

Keywords- cloud computing; resource allocation algorithms; lightweight scaling

I. INTRODUCTION

Cloud computing has gained unquestionable commercial success in recent years. Key value propositions promoted by cloud IaaS (Infrastructure-as-a-Service) providers, namely infrastructure owners or resellers such as Amazon AWS [1] and GoGrid [2], include the user's ability to scale up or down resources used based on their computational demand. The approach is typically based on charging the application owners for using one or more Virtual Machines (VMs) per unit time. These VMs are then hosted for execution on different Physical Machines (PMs) at the infrastructure provider's cloud. In addition, many applications hosted in a cloud environment may have varying resource demands as they are expected to serve wide range of workloads, some of which could be predictable while others fluctuate based on using the application. For example, an e-commerce website may have a workload fluctuation during the release of some promotion activities or other events. Within this context, on-demand scaling (also known as dynamic or online resource provisioning) of applications becomes one of the most important features of cloud services. This feature enables real-time acquisition/release of computing resources to adjust the applications performance in order to meet QoS (Quality of Service) requirements, while also minimising the infrastructure provider's operational cost to remain competitive.

Most approaches for on-demand resource scaling, whether used in practice or described in the literature [1, 3, 4, 5, 6, 7], are typically based on controlling (increasing or decreasing) the number of VM instances that host the applications' server components. The approach is appealing for a wide class of applications especially those that are based on multitier architectures, or server-side software platforms. For such applications, scaling up an application typically involves adding an extra software server, and hence an extra (server) VM in the cloud context.

Using VMs as basic units for dynamic resource provisioning for cloud applications incurs considerable overhead and costs. Allocation of new VMs incurs considerable waste of computing resources as each VM consumes resources which are not directly used by the applications. As the number of allocated VMs increases, the total amount of resources used for housing keeping also increases. In addition, creating/shutting down/removing VMs dynamically at run time incurs extra overheads. Based on our experience, however, we noticed that in many real-world scenarios, scaling an application's resource up and down does not always require its underlying computing infrastructure to be changed much by adding/removing VMs constantly. More subtle changes, such as modifying VMs' capacity, can be conducted to achieve the desired effects with less running cost and, usually, in less time.

In this paper, we propose a lightweight approach for achieving more cost-efficient scaling of cloud resources at the IaaS cloud provider's side. This approach offers benefits for multi-tier applications that are already implemented using multiple VMs by improving the resource utilisation between them as application demands vary. In particular, we present:

Fine-grained scaling approach: a novel lightweight scaling (LS) algorithm is introduced to enable fine-grained scaling of an application at the level of underlying resources, namely CPU, Memory and I/O.

Improving resource utilisation: the LS algorithm increases the resources utilisation of PMs by using idle resources to release overloaded resources.

Implementation and experimental evaluation: an intelligent platform based on the LS algorithm is implemented to automate the scaling process of cloud applications. The proposed lightweight approach is tested using an industry standard benchmark [8] on the IC-Cloud infrastructure [9] and the test results show its effectiveness in scaling applications to meet their QoS requirements while reducing infrastructure providers' cost.

The remainder of the paper is organized as follows: Section II provides the motivation and background to our

* Please direct your enquiries to the communication author Professor Yike Guo.

approach; Section III presents an overview of the related work on scaling techniques used for cloud computing; Section IV presents our LS algorithm; Section V introduces the architecture of the Imperial Smart Scaling engine (iSSe) implemented to support our approach; Section VI presents an experimental evaluation of our work, and compares its effectiveness against traditional approaches; and Section VII presents a summary and discussion of the framework and describes avenues for future work.

II. MOTIVATION

This section illustrates the idea of lightweight scaling for cost-effective resource management in a cloud environment. The approach is generic, and we present it using the example of an e-commerce application, which abstracts the typical behaviour of a wide class of applications running on cloud services. Figure 1(a) shows the basic infrastructure of the application that is composed of five tiers of components (servers): the HAProxy and Amoeba load balancing servers, the Apache HTTP server, the Tomcat application server and the MySQL database server. These servers work together to handle end-users' requests.

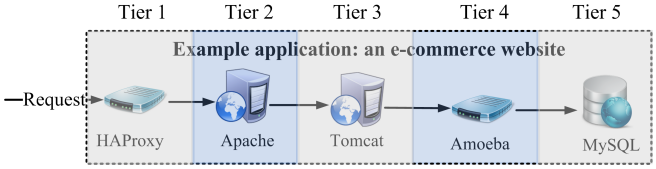

Figure 1. The example of an e-commerce website.

Once deployed, the servers of this application are hosted across different VMs. An interesting point here is that the amount of resources that are consumed by these servers depends on the behaviour of the application itself. For example, if webpage browsing and product previewing are the application users' main actions over a period of time, Apache and Tomcat web servers will be stressed and their resources become saturated. By contrast, the MySQL database component would remain performing well. However, from the application owner's point of view (e-commerce website), the overall performance of the application is affected. Ideally, they may choose to allocate more resources to the bottleneck component of the application, namely the webs servers, while keeping the MySQL database component in idle status. We call such cases "imbalanced resource provisioning" for an application.

Typically, for an application in the cloud environment, different types of imbalanced resource provisioning could exist. To illustrate, consider Figure 2(a), where a single server VM executes on its own PM. In this case, any of the three types of resources (CPU, memory and I/O), could suffer from high or low utilisation based on some metric. For example, one can consider resources with high utilisation (e.g., a memory's utilisation is 0.8 means 80% of the memory is used) to be typically overloaded, and those with low utilisation (e.g., 0.3) to be mostly idle. Such imbalance becomes significant for applications using more than one VM executing on the same PM. Such VMs typically can compete for the same resources (Figure 2(b)). Note that the overall utilisation of resources also varies across different PMs used by the application (Figure 2(c)). The aim of lightweight scaling is to operate at resource level to address such imbalances and improve resource utilisation rather than adding new resources to scale the application up.

Conceptually, lightweight scaling can be based on two techniques. The first, *Self-healing scaling* is applicable in cases when two servers of an application are deployed on one PM, and when the idle resources of one VM can be used to release the overloaded resources in the other. For example, one or more CPUs allocated to a VM with low CPU utilisation can be removed and to another VM which has already saturated existing CPU resources. The second technique, *Resource-level scaling*, is based on using unallocated resources available at a particular PM to scale up a VM executing on it. For example, a PM with low CPU and memory utilisation can allocate these types of resources to one of the VMs executing on it thus scaling it up.

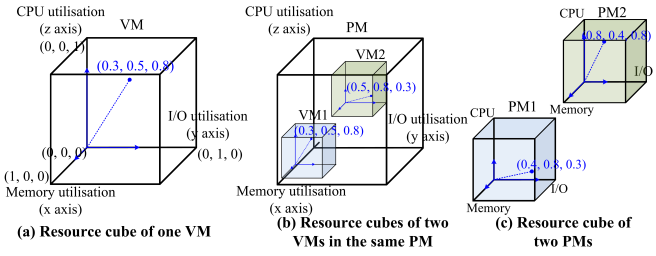

Figure 2. Resource shown as cubes and three types of unbalanced resource provisioning.

Our motivation in this paper is to develop a system and algorithms based on a light-weight scaling approach to support this conceptual framework. The system would be installed at an IaaS cloud provider's side enabling them to support QoS requirements of the application owner without incurring unnecessary costs for either the provider or application owner. The approach is generic and can be applied to any cloud provider for a wide range of applications. However, our focus in this paper is on transaction-based multitier applications where the performance of the application, and hence QoS, can be assessed based on the application's response time to each incoming request.

III. RELATED WORK

A. Scaling Techniques before Clouds

Application scaling has long been studied before clouds. Early work considers single-tier applications and focused on transforming performance targets into underlying resources such as CPU numbers and allocated bandwidth [10]. Further investigations classify an application into multiple tiers and provision each tier with enough resources to meet the application's peak workload [11, 12].

Although scaling of traditional applications on physical servers shares many commonalities with that of cloud applications, scaling in these two environments has different emphases. Conventional techniques mainly focused on how to schedule compute nodes to meet QoS requirement of

applications by predicting their long-term demand changes. In contrast, clouds focus on providing metered resources on-demand and on quickly scaling applications up and down whenever the user demand changes. Further investigations, therefore, are needed to address challenges brought by this requirement for high elasticity.

B. Traditional Scaling Techniques in Clouds

Most IaaS cloud providers (e.g., Amazon AWS [1]) and vendors (e.g., RightScale [3]) assist application owners to manage cloud infrastructure and employ pre-defined policies (rules) to guide application scaling. For example, vendors like RightScale require application owners to manually specify scaling rules after an application is deployed. These rules specify the upper and lower bounds of the number of servers, the conditions to trigger scaling and the number of changed servers in each scaling. The policy-based mechanism assumes that the application owners have particular knowledge of the application being executed so that they can define proper policies. However, this is not always the case.

In addition, a variety of scaling approaches are proposed based on analytical models of the application. In [4], Bacigalupo et al. model an application by a three-tier queueing model, namely application, database and storage disk tiers. Each tier is solved to analyse the servers' mean response time and throughput. A scaling algorithm is then proposed using these analysis results. Similar to [11, 12], researchers in [5] break down an application's end-to-end response time by tier. They then calculate the number of servers to be allocated to each tier in order to meet the response time target. In addition, a method is presented to support the scaling up of a two-tier web application by actively profiling the VMs' CPU usage. The method also realises scaling down using a regression-model-based predictive mechanism [6]. In [7], a network flow model is introduced to analyse applications to assist application owners in making a trade-off between cost and QoS. In [13], Gong et al. propose a scaling approach that conducts accurate resource allocation using two complementary techniques: patterns-driven (they call repeating patterns as signatures) and state-driven (they use discrete-time Markov chain) demand predictions. In [14], Shen et al. further improve the prediction mechanism by introducing two handling schemas, namely online adaptive padding and reactive error correction.

C. Lightweight Cloud Resource Management

At present, some preliminary research has conducted on lightweight management of cloud resources. In [15], He et al. proposed a novel concept called Elastic Application Container (EAC) which is a lightweight resource management approach. It is considered as a substitution of VM virtual infrastructure for hosting cloud applications. Their experiments demonstrate that EAC is more resource-efficient and feasible than VM infrastructure in terms of application migrations and resource utilisation. In addition, a lightweight mechanism is presented to reduce VM migration time by pre-caching memory pages [16]. Researchers in [17] focus on live migration of shared storage databases. They introduce an end-to-end technique to ensure the minimal impact on database performance during migration.

IV. THE LIGHTWEIGHT SCALING ALGORITHM

In this section, we provide an overview of the LS algorithm, and the associated algorithms for scaling up and down.

A. The LS Algorithm Overview

In order to help readers understand the algorithm, Table 1 lists the main parameters used throughout the LS algorithm. Their values are obtained in a variety of ways.

TABLE I. PARAMETERS OF THE LS ALGORITHM

Parameter	Description	Source
r	Three resource dimensions of a VM: $r \in \{CPU, memory, I/O\}$.	
s	A server s hosted on a VM.	
PM(s)	Server s's PM.	
$c(s)$	Server s's running cost.	Cloud providers
$c(s,r)$	One unit of resource r's running cost.	
$u^{su}(r)$	The threshold of utilisation for scaling up resource r.	
$u^{sd}(r)$	The threshold of utilisation for scaling down resource r.	
(t^l, t^u)	Lower bound t^l and upper bound t^u of the required response time.	Application owners
t^o	The observed end-to-end response time	Online measurement
$u(s,r)$	Resource r's utilisation in server s.	

First of all, the IaaS cloud provider needs to specify its running cost of a single server (for convenience, each server is installed on one dedicated VM) or the server's underlying resources, namely CPU, memory and I/O. This cost represents resource consumption and operating cost of cloud providers. Such a cost varies for different cloud providers and the cost function is a research topic in itself [18]. Note that resources are typically charged based on their basic unit. For example, the basic unit of memory is 1GB and it is charged 2 cents/hour. In addition, cloud providers also give two thresholds of utilisation for triggering scaling up (down) of resource r. For example, an application may define the CPU's threshold for scaling up (i.e., $u^{su}(CPU)$) as 0.8. This threshold represents high CPU usage and indicates that an additional CPU should be added. Similarly, the memory's threshold for scaling down (i.e., $u^{sd}(memory)$) is 0.3, which denotes the low memory usage and so some redundant memory can be removed.

In addition, application owners specify the QoS demand as the required response time in a service level agreement (SLA). Without losing generality, this response time includes a lower bound t^l and an upper bound t^u. These two bounds define a range of response times such as 1.8 to 2.0 seconds that are acceptable to service users. Finally, the observed

end-to-end response time of the application and each VM's resource utilisation is collected online by monitors.

Given an m-tier application to be scaled with the LS algorithm, this application has n server components, i.e., its sever set S consists of n servers where $s_i \in S$ and $i = 1, \ldots, n$. Pseudo code for the LS algorithm is given below. The algorithm is started after the application is initially deployed and keeps running until the application is completed (lines 2 to 7). The algorithm triggers a lightweight scaling up (LSU) whenever the observed response time is larger than the upper bound of the required response time (lines 4 and 5), or a lightweight scaling down (LSD) if the observed time is smaller than the lower bound (lines 6 and 7).

```
Algorithm 1: LS
Input: S, (t^l, t^u)
1.  Begin
2.    while (the application is not completed)
3.      Monitor t^o once every few minutes;
4.      if t^o>t^u, then
5.        LSU (S, (t^l, t^u), t^o);
6.      else if t^o<t^l, then
7.        LSD (S, (t^l, t^u), t^o).
8.  End
```

The lightweight scaling here has a twofold meaning. First, the algorithm conducts fine-grained scaling by modifying each VM's resource configuration. This ensures that the algorithm is light weight and consumes insignificant computing resources for resource adjustment. The scaling can be completed in as quick as few milliseconds, which guarantees the period of violating response time acceptable by end users. Second, the algorithm applies an automatic reactive scaling mechanism similar to mechanisms applied by Amazon EC2 and Rightscale. This measurement-based scaling needs no prior knowledge about the applications. Each time an application is scaled, its load balancing servers automatically redistribute incoming requests and the service monitor can instantaneously detect the updated end-to-end response time. The scaling process is completed when the detected response time meets the requirement (i.e., it falls between t^l and t^u). This ensures the just enough resource allocation to the application and minimises cloud providers' cost.

B. The LSU Algorithm

The LSU algorithm (Algorithm 2) aims to reduce an application's response time below the upper bound t^u, while minimising the increase of resources or number of VM instances.

```
Algorithm 2: LSU
Input: S, (t^l, t^u), t^o.
Output: updated S.
1.  Begin
2.    Conduct the self-healing scaling;
3.    if t^o>t^u, then
4.      Conduct the resource-level scaling up;
5.    while (t^o>t^u)
6.      Conduct the VM-level scaling up;
7.      Conduct the resource-level scaling up.
8.  End
```

LSU algorithm conducts three types of scaling with different levels of priorities. The self-healing and resource-level scalings first reduce the application's response time by increasing VMs' capacity using the available resources from these servers' hosting PMs (lines 2 to 4). Note that these two scalings have some constraints. For example, a VM instance with a 32-bit operating system can only be allocated 4G memory in maximum. The LSU can be accomplished if any of these two scalings meet the response time target. Otherwise, the VM-level scaling is required to add a VM instance in a new PM to the application (line 6). This new PM may have available resources and so the resource-level scaling can be executed again (line 7).

The basic idea of **self-healing scaling** is that if two VMs of the application's servers are hosted in the same PM, these VMs' resource may complement each other. This scaling uses a candidate list L^s (line 6 in algorithm 3) to store all pairs of eligible servers for VM modification. A pair of eligible server (s_i, s_j) must satisfy: s_i and s_j belong to the same application and PM, s_i's VM has one resource r (e.g., CPU) whose utilisation is larger than its scaling up threshold (e.g., 0.8), and resource r's utilisation in s_j's VM is smaller than its scaling down threshold (e.g., 0.3). One unit of resource r is then removed from s_j's idle VM (line 7) and it is added to s_i's overloaded VM (line 8). The self-healing scaling iteratively executes until the required response time is met (line 2) or no eligible server can be found (i.e., L^s is empty).

```
Algorithm 3: Self-healing scaling
1.  Measure t^o and u(s,r);  //s ∈ S
2.  While (t^o>t^u and L^s ≠ ∅)
3.    L^s={};
4.    For each pair (s_i, s_j) in S
5.      if (u(s_i, r) > u^su(r) and PM(s_i)==PM(s_j) and
    u(s_j,r)< u^sd(r)), then
6.        L^s= L^s ∪{(s_i, s_j)};
7.        Remove one unit of resource r from s_j;
8.        Add one unit of resource r to s_i;
9.      Measure t^o and u(s,r).
```

Resource-level scaling up follows the observation that if the PMs that host the application's server components have available resources, these resources can be used for scaling up while other applications hosted in these PMs are not affected. One possible way to ensure available resources is to reserve some resources in advance. Note that a PM can theoretically generate an unlimited number of CPUs. Hence the availability of CPU resource could be judged according to the PM's CPU utilisation (e.g., the CPU resource is available if the CPU utilisation is lower than 0.8).

In the resource-level scaling up (see algorithm 4), an eligible server s_i's VM must have a saturated resource r (i.e., its utilisation is higher than its scaling up threshold) and this VM's hosted PM must have one unit of available resource r (line 5). This scaling selects all eligible servers into a candidate list L^r (line 3). If multiple servers exist in L^r, the criterion $x^r(s,r)$ is used to select a server s_j and scale up its resource r (lines 7 and 8).

As listed in definition 1, the criterion $x^r(s,r)$ consists of two variables: 1) resource r's exploitable ratio (i.e., 1-$u(s,r)$), which represents the remaining amount of r in server s's VM; 2) r's running cost $c(s,r)$. Using this criterion, the resource with the least remaining amount and cost is selected in the resource-level scaling up.

```
Algorithm 4: Resource-level scaling up
1.  Measure t^o and u(s,r); //s ∈ S
2.  While (t^o>t^u and L^r ≠ ∅)
3.    L^r={};
4.    for (i=1; i<|S|; i++)
5.      if (u(s_i,r)>u^su(r) and
    PM(s_i).hasAvailableResource==true), then
6.        L^r=L^r ∪{s_i};
7.    Select s_j from L^r with the smallest x^r(s_j,r);
8.    Add one unit of resource r to s_j;
9.    Measure t^o and u(s,r).
```

Definition 1 (Resource-level scaling criterion $x^r(s,r)$). *The criterion used to select servers in resource-level scaling, denoted by $x^r(s,r)$, is the product of a parameter p^r (positive constant), server s's exploitable ratio and resource r's running cost:*

$$x^r(s,r)=p^r(1-u(s,r))c(s,r). \quad (1)$$

VM-level scaling up complements the above two scalings and it has the lowest priority to be triggered. This scaling uses the criterion $x^v(S_j)$ to detect the bottleneck tier j from the application for adding a server. As shown in definition 2, criterion $x^v(S_j)$ consists of two variables: 1) the mean value of servers' exploitable ratios at tier j. A server s_i's exploitable ratio is the product of its three resources' exploitable ratios and it comprehensively represents s_i's remaining capacity; 2) the running cost of a server at tier j.

Definition 2 (VM-level scaling criterion $x^v(S_j)$). *The criterion used for detecting the bottleneck tier j in VM-level scaling, denoted by $x^v(S_j)$, is the product of a parameter p^v (positive constant) and the mean value of each server's exploitable ratio multiplied by its cost at tier j:*

$$x^v(S_j)=p^v\frac{\sum_{i=1}^{|S_j|}(\prod_{r=CPU,memory,I/O}(1-u(s_i,r)))c(s_i)}{|S_j|}. \quad (2)$$

Where $s_i \in S_j$ and $|S_j|$ is tier j's number of servers.

The VM-level scaling up uses criterion $x^v(S_j)$ to select the tier j with the least remaining resources and the smallest cost to add a server (line 4 in algorithm 5). In other words, adding a server to this tier should reduce each unit of response time while incurring the smallest running cost for cloud providers.

```
Algorithm 5: VM-level scaling up
1.  Measure t^o and u(s,r); //s ∈ S
2.  for (i=1; i< m; i++)
3.    Calculate tier i's x^v(S_i); //i= 1,..., m
4.  Select tier j with the smallest x^v(S_j);
5.  if (Add a server s_j to tier j is feasible),
    then
6.    S= S ∪{s_j}. // add s_j to S
```

Note that the addition of a server must satisfy some constraints (line 5). Examples are the application owner's budget and each tier's maximum number of servers. Servers' own constraints should also be considered. For instance, MySQL Master's replication constraint is 1, which means an application can have one MySQL Master server at most.

C. The LSD Algorithm

The LSD algorithm (algorithm 6) aims to remove as many VMs and resources from an application as possible, while still trying to hold its response time target. The algorithm first performs the VM-level scaling down to identify a bottleneck tier with the largest $x^v(S_j)$ (definition 2) to remove a server. This removal is expected to save the maximum cost per unit of increased response time. The VM-level scaling down keeps running until it is infeasible (line 2): removing a VM would violate response time target. This feasibility can be checked using application profiling and workload predication techniques. The LSD algorithm then conducts the resource-level scaling down. This scaling down selects a resource r with the largest $x^r(s_i,r)$ (definition 1) and removes one unit of r from s_j's VM. The removal implies the resource with the largest available amount and the highest running cost is removed.

```
Algorithm 6: LSD
Input: S, (t^l, t^u), t^o.
Output: updated S.
1.  Begin
2.    Conduct the VM-level scaling down;
3.    Conduct the resource-level scaling down.
4.  End
```

V. A PLATFORM TO SUPPORT LIGTWEIGHT SCALING

This section introduces a platform, called iSSe, to support lightweight scaling of cloud applications and automate this scaling process. As shown in Figure 3, iSSe acts as middleware to be installed on top of an existing IaaS cloud provider's infrastructure. Using the *User Portal*, application owners can specify the required response time for their applications and get notified when these applications are scaled up (down) using the *Lightweight Scaling Service*.

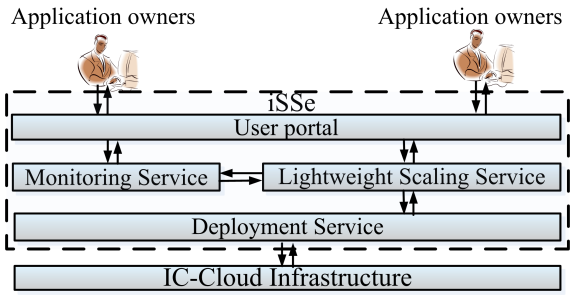

Figure 3. The architecture of iSSe.

In iSSe, the *Monitoring Service* monitors each running application in two ways. First, an entry monitor examines the incoming requests over a finite interval (e.g., 60 seconds). The monitored information includes the requests' arrival rate and the end-to-end response time. This information is used to infer whether a scaling up (or down) is needed. For instance, a scaling up is triggered if the observed response time exceeds the required time. Secondly, a monitor is installed in each server. This monitor examines the resources' utilisation.

When a scaling is triggered, the *Lightweight Scaling Service* applies the LS algorithm to control the whole scaling process and the *Deployment Service* scales the resources of the application upon IC Cloud [9]. In iSSe, each single server is described by a XML based specification using the Open Virtualization Format (OVF) standard [19]. This specification accommodates all aspects of a single server, including its VM resource configuration, software user settings (e.g., Tomcat's user name and password), links to other servers and other constraints. iSSe can automatically executes the scaling process, namely *addition*, *removal* and *resource modification* of servers, by interpreting servers' specifications and calling interfaces of IC Cloud [9]. The automation process and iSSe are detailed in [20].

VI. EXPERIMENTAL EVALUATION

The experimental evaluation is designed to illustrate the effectiveness of our approach in enabling fine-grained scaling up and down of applications (section VI.B). Furthermore, the algorithm's salient feature in supporting cost-effective scaling for cloud providers is demonstrated by comparing with existing techniques (section VI.C).

A. Experimental setup and Benchmark

In the experiment, the iSSe was implemented as a full working system and was tested in one data center running the IC-Cloud platform [9]. This data center has four PMs that share the 4.1 Tb centralised storage and are connected through a switched gigabit Ethernet LAN. Each PM has 8 Quad-Core AMD processors and 32GB memory.

Figure 1's e-commerce website and its five types of servers were implemented. For convenience, each server is installed on one dedicated VM running Linux Centos 5.4. Note that two versions of the MySQL database, namely MySQL Master and Slave, are implemented to support the Master-Slave data replication model.

Two workloads of the application were tested: 1) the "browsing" workload represents the browsing behaviour of customers such as searching and ranking products as well as online enquiry; 2) the "ordering" workload denotes the ordering actions such as logging in and making orders, which make heavy database queries.

We developed a client emulator to generate a sequence of session-based requests based on the TPC benchmark [8]. After setting the test period, this emulator continuously generates a sequence of interactions initiated by multiple active sessions. After an interaction is completed, the emulator waits for a random interval before initiating the next interaction to simulate customers' thinking time. As shown in Figure 4, the varying number of active sessions represents the fluctuating demand. For each type of workload, a trial is performed and it lasts 3600 seconds. Each trial can be divided into six stages, in which three changes in demand pattern are simulated: when stages 2, 4 and 6 start, the number of active sessions increases to 300, 400 and 600, respectively. Each increased period of demand lasts 600 seconds and then the level falls down to 200 sessions.

In the trial, the lower and upper bound of the required average response time is 1.8 and 2.0 seconds; the threshold of utilisation for scaling up is 0.8 and for scaling down is 0.3; other parameters of the application are monitored once every 60 seconds. Initially (stage 1), six servers with basic VM configurations are deployed on a PM and newly added servers are deployed on another PM. Cloud providers' costs (cost unit: cents/hour) for running these servers are: 35 for HAProxy and Amoeba, 60 for MySQL Master and 16 for the other three servers. In lightweight scaling, these costs for the three types of resource are: 3 for 10Mb/second bandwidth (i.e., I/O resource), 3 for 1 GB memory and 2 for one CPU.

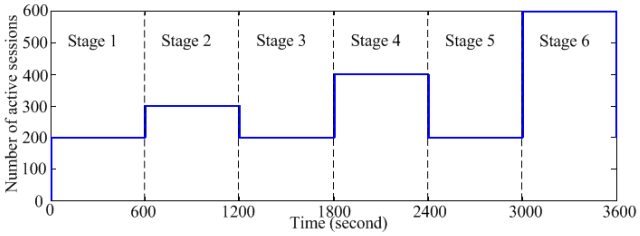

Figure 4. The number of active sessions during six stages of the experiment.

B. Effectiveness of the LS Algorithm

Figure 5 demonstrates the fluctuation of the observed end-to-end response time for two different workloads using the LS algorithm. Each time the demand (i.e., the number of sessions) changes, the response time violates the required upper or lower bound. This violation is detected when the LS algorithm monitors the application every 60 seconds and a scaling up (down) is triggered. In most of the cases, this scaling can be completed in a few milliseconds. One exception is at time=3000 seconds, new VMs are added and this incurs the scaling up time of 1 or 2 minutes.

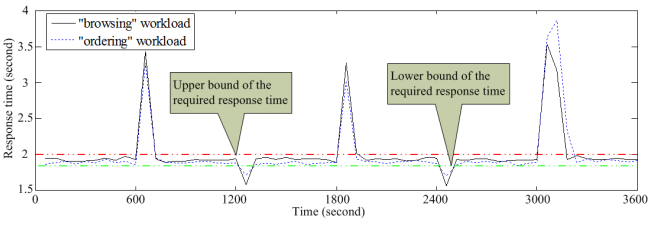

Figure 5. The observed end-to-end response time.

Figure 6 displays the variation of the assigned resources in the trial of the "browsing" and "ordering" workloads. The "browsing" workload mainly consumes memory and CPU resources, so the LS algorithm mainly modifies the configurations of these resources to handle the changing demand (Figure 6(a) and (c)). Stages 2, 4 and 6 show the

results of the self-healing, resource-level and VM-level scaling up operations. These three scalings correspond to the small, middle and large increases in demand, respectively. In particular, at stage 6, one Tomcat and one Apache servers are added. By contrast, the "ordering" workload primarily uses the I/O resources and the bandwidth configuration of VMs is modified (Figure 6(b)).

Results: Our lightweight scaling approach can adapt to different fluctuating demands and perform the fine granularity scaling up (down) of applications to meet response time targets.

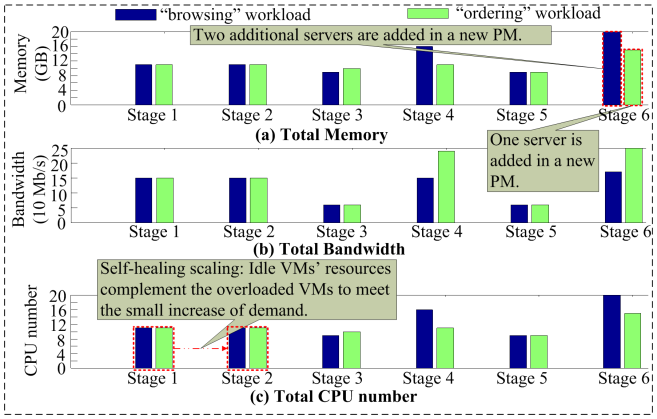

Figure 6. Total resource configuration in the trial of the two workloads.

C. A Comparison with Existing Scaling Techniques

We repeat the previous section's trial to compare the LS algorithm with traditional VM-level scaling algorithms. Typically, existing scaling techniques can be divided into two groups. First, the Policy-based scaling (PBS) algorithm scales applications using pre-defined rules [1, 3]. These rules are triggered to execute scaling according to monitoring results. In the experiment, if the resource utilisation of one type of server (e.g., Apache) is larger than 0.8 for 1 minute, additional servers are added. If this utilisation is less than 0.3 for 1 minute, redundant servers are removed. Secondly, in the Tier-Dividing Scaling (TDS) algorithm, the upper bound of the required response time (2.0 seconds in the trial) is broken down into three per-tier response times, which are 40%, 20%, and 40% for the tiers of Apache, Tomcat, MySQL Master (Slave), respectively. The TDS algorithm then calculates the suitable number of servers for each tier using analytical models [5, 6, 11, 12].

Figure 7 shows the scaling results of three algorithms in the trial of two workloads. The three algorithms are compared from three aspects.

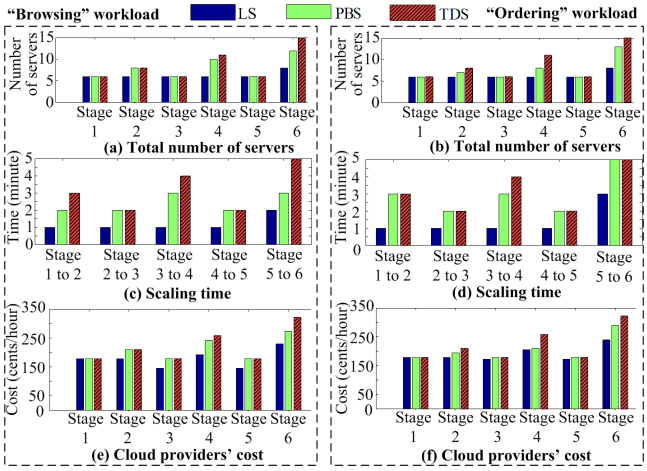

Figure 7. Comparing results of the three scaling algorithms.

1) *Total number of servers*: Figure 7a(b) display the application's total number of servers at each stage of the trial. The PBS algorithm increases the number of overloaded servers in scaling up and decreases the number of idle servers in scaling down. Depending on different types of workloads, different tiers of the application are scaled in the PBS algorithm. The TDS conducts worst-case scaling and allocates each tier the same number of servers for both types of workloads. By contrast, the LS algorithm only needs to increase the number of VMs in the fifth scaling and the application's number of servers is the smallest in this algorithm. This implies our algorithm can meet the same QoS requirement using the smallest amount of resources, thereby improving resource utilisation of the application's hosting PMs.

2) *Scaling time*: Five scaling up (down) times in each trial are listed in Figure 7c(d). Each scaling time represents the reaction time to an increase (decrease) of demand. For example, the scaling time of "stage 1 to 2" denotes the reaction time when the number of sessions changes from 200 to 300. Each scaling lasts at least 1 minute because the monitor needs this amount of time to detect the demand change. In both workloads, our LS algorithm has the shortest scaling time. This is because our algorithm needs the least addition/removal operations of VMs. In cloud platforms, the duration of these operations (several minutes) is much longer than that of the resource modification operations (a few milliseconds).

3) *IaaS Cloud providers' cost*: Figure 7e(f) compare cloud providers' cost for running the application. Experimental results show that the LS algorithm incurs the smallest cost by deploying as few VMs as possible in the three scaling up operations (stages 2, 4 and 6). In the scaling down (stage 3 and 5), the PBS and TDS algorithms revert the application to its initial deployment. By contrast, the LS algorithm further reduces VMs' capacity to save cost by removing their idle resources.

Results: Our LS algorithm can perform agile scaling up and down of applications as well as save cloud providers' cost by improving resource utilisation.

VII. Conclusion and Discussions

In this paper, we argued that conventional VM-level scaling of cloud services may over-use resources and increase cloud providers' operating costs to meet QoS requirements for user applications. Based on this argument, we proposed a lightweight approach that operates fine-grained scaling at resource level and improves resource utilisation for cloud service operations. The approach can efficiently scale cloud application's resources up and down in order to meet the given QoS requirements while reducing cloud providers' costs. An intelligent platform based on the lightweight scaling approach has been designed to achieve this cost-effective scaling. The practical effectiveness of the approach has been successfully tested using industry standard benchmarks.

The overall approach presented in this paper integrates both fine-grained scaling at the resource level itself (CPUs, memory, I/O, etc.) in addition to VM-level scaling, and can be used effectively within a single application. There are multiple avenues for extending the work and we discuss two possible directions below.

We have described self-healing scaling technique in the context of a single application. This technique coordinates resource usage within an application by using the idle resource of a server to release the overloaded resource in another server hosted in the same PM. The major advantage of this type of scaling is that it scales up the application without incurring any cost on either the cloud provider or the application owner. In this paper, we have considered this type of scaling when two servers belong to the same application. This single-application requirement could be relaxed and investigated in the context of multiple applications belonging to the same user. This is appealing for two reasons. First, coordinating resources within a user's applications does not affect other users' applications. Second, scaling up and down of resources can be completed within milliseconds and the approach could enable real-time scheduling to allow multiple applications to meet their QoS requirements simultaneously. However, further investigations are needed to consider how resources can be scheduled between applications with different QoS requirements in this case.

We note that, in our approach, when resource-level scaling is conducted, an application can only be scaled up when its hosting PMs have available resources. This may not be the case always. One possible way to provide available resources is to reserve some resources in these PMs that can only be used by resource-level scaling. Obviously, the higher proportion of the PMs' resource is reserved, the larger chance that resource-level scaling can be performed and so high-cost VM-level scaling can be avoided. However, excessive resource reservation causes high cost for the IaaS cloud providers as well. Further investigations, therefore, are needed to understand the trade-offs between reservation cost and the risk of high running cost incurred by VM-level scaling. Another approach to provide available resources is to schedule the VMs among the PMs within a data center. For example, one VM in an over-loaded PM can be migrated to another light-loaded PM to release some resources. Existing VM migration/placement techniques, therefore, can be applied in lightweight scaling to help cloud providers achieve resource efficiency in the whole data center and solve conflicts between applications during scaling.

Finally we also note that we have only considered transaction-based applications that execute at a single cloud provider. Considering more complex applications high performance scientific applications that can be scheduled, or coordinated, across different providers in a Grid computing fashion [21] and how to address their requirements would also be another interesting avenue for extending this research.

References

[1] Amazon elastic compute cloud (EC2): http://aws.amazon.com/ec2/ (29.10.2011).

[2] GoGrid: http://www.gogrid.com/ (29.10. 2011).

[3] RightScale: http://www.rightscale.com/ (29.10. 2011).

[4] D. A. Bacigalupo, J. van Hemert, et al, "Managing dynamic enterprise and urgent workloads on clouds using layered queuing and historical performance models," Simulation Modelling Practice and Theory, vol. 19, pp. 1479-1495, 2011.

[5] J. Bi, Z. Zhu, R. Tian, and Q. Wang, "Dynamic Provisioning Modeling for Virtualized Multi-tier Applications in Cloud Data Center," in CLOUD'10, Miami, Florida, 2010, pp. 370-377.

[6] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," Future Generation Computer Systems, vol. 27, pp. 871-879, 2011.

[7] J. Z. Li, "Fast Optimization for Scalable Application Deployments in Large Service Centers," Doctor of Philosophy thesis, Department of Systems and Computer Engineerin, Carleton University, 2011.

[8] TPC Transaction Processing Performance Council: http://www.tpc.org/ (29.10. 2011).

[9] L. Guo, Y. Guo, and X. Tian, "IC Cloud: A Design Space for Composable Cloud Computing," in CLOUD'10, 2010, pp. 394-401.

[10] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat, "Model-based resource provisioning in a web service utility," in USITS'03, 2003, vol. 4, pp. 5-5.

[11] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier Internet applications," ACM Transactions on Autonomous and Adaptive Systems (TAAS), vol. 3, pp. 1-25, 2008.

[12] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," in ICAC'05, Seattle, Washington, 2005, pp. 217 - 228

[13] Z. Gong, X. Gu, and J. Wilkes, "PRESS: PRedictive Elastic ReSource Scaling for cloud systems," in CNSM'10, Niagara Falls, Canada, 2010, pp. 9 - 16.

[14] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "CloudScale: elastic resource scaling for multi-tenant cloud systems," in SOCC'11, Cascais, Portugal, 2011.

[15] S. He, L. Guo, Y. Guo, M. Ghanem, R. Han, and C. Wu, "Elastic Application Container: A Lightweight Approach for Cloud Resource Provisioning," in The 26th IEEE International Conference on Advanced Information Networking and Applications (AINA-2012), Fukuoka, Japan, 2012, in press.

[16] T. Hirofuchi, H. Nakada, S. Itoh, and S. Sekiguchi, "Enabling instantaneous relocation of virtual machines with a lightweight VMM extension," in 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid'11), Melbourne, VIC, Australia, 2010, pp. 73-83,

[17] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration," Proceedings of the VLDB Endowment, vol. 4, pp. 494-505, 2011.

[18] M. Mihailescu and Y. M. Teo, "Dynamic Resource Pricing on Federated Clouds," in 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid'11), Melbourne, VIC, Australia, 2010, pp. 513-517.

[19] VMware OVF: http://www.vmware.com/appliances/getting-started/learn/ovf.html (29.10. 2011).

[20] R. Han, L. Guo, Y. Guo, and S. He, "A Deployment Platform for Dynamically Scaling Applications in The Cloud," in 3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom'11), Athens, Greece, 2011, pp. 506-510.

[21] M. Ghanem, Y. Guo, A. Rowe, P. Wendel, "Grid-based knowledge discovery services for high throughput informatics," in Proceedings. 11th IEEE International Symposium on High Performance Distributed Computing, (HPDC'02), Edinburgh, UK, 2002, p. 416.