SLA-Driven Adaptive Resource Management for Web Applications on a Heterogeneous Compute Cloud

Waheed Iqbal[1], Matthew Dailey[1], and David Carrera[2]

[1] Computer Science and Information Management, Asian Institute of Technology, Thailand

[2] Technical University of Catalonia (UPC) Barcelona Supercomputing Center (BSC) Barcelona, Spain

Abstract. Current service-level agreements (SLAs) offered by cloud providers make guarantees about quality attributes such as availability. However, although one of the most important quality attributes from the perspective of the users of a cloud-based Web application is its response time, current SLAs do not guarantee response time. Satisfying a maximum average response time guarantee for Web applications is difficult due to unpredictable traffic patterns, but in this paper we show how it can be accomplished through dynamic resource allocation in a virtual Web farm. We present the design and implementation of a working prototype built on a EUCALYPTUS-based heterogeneous compute cloud that actively monitors the response time of each virtual machine assigned to the farm and adaptively scales up the application to satisfy a SLA promising a specific average response time. We demonstrate the feasibility of the approach in an experimental evaluation with a testbed cloud and a synthetic workload. Adaptive resource management has the potential to increase the usability of Web applications while maximizing resource utilization.

1 Introduction

Cloud providers such as Google and Amazon offer computational and storage resource rental services to consumers. Consumers of these services host applications and store data for business or personal needs. The key features of these services, on-demand resource provisioning and pay-per-use, mean that consumers only need to pay for the resources they actually utilize. In this environment, cloud service providers must maximize their profits by fulfilling their obligations to consumers with minimal infrastructure and maximal resource utilization.

Although most cloud providers provide Service Level Agreements (SLAs) for availability or other quality attributes, the most important quality attribute for Web applications from the user's point of view, *response time*, is not addressed by current SLAs. The reason for this is obvious: Web application traffic is highly unpredictable, and response time depends on many factors, so guaranteeing a particular maximum response time for any traffic level would be suicide for the

M.G. Jaatun, G. Zhao, and C. Rong (Eds.): CloudCom 2009, LNCS 5931, pp. 243–253, 2009.
© Springer-Verlag Berlin Heidelberg 2009

cloud provider unless it had the ability to dynamically and automatically allocate additional resources to the application as traffic grows.

In this paper, we take steps toward eliminating this limitation of current cloud-based Web application hosting SLAs. We present a working prototype system running on a EUCALYPTUS-based [1] heterogeneous compute cloud that actively monitors the response time of the compute resources assigned to a Web application and dynamically allocates the resources required by the application to maintain a SLA that guarantees specific response time requirements.

There have been several efforts to perform adaptive scaling of applications based on workload monitoring. Amazon Auto Scaling [2] allows consumers to scale up or down according to criteria such as average CPU utilization across a group of compute instances. [3] presents the design of an auto scaling solution based on incoming traffic analysis for Axis2 Web services running on Amazon EC2. [4] demonstrate two software systems, Shirako [5] and NIMO [6]. Shirako is a Java toolkit for dynamic resource allocation in the Xen virtualization environment that allocates virtual resources to guest applications from a pool of available resources. NIMO creates an application performance model using active learning techniques. The authors use NIMO to build performance models by capturing resource requirements, data characteristics, and workload statistics for a guest application. They then use Shirako to allocate the necessary resources to the application. [7] use admission control and dynamic resource provisioning to develop an overload control strategy for secure Web applications hosted on SMP (Symmetric MultiProcessing) platforms. They implement and experiment with a global resource manager for the Linux hosting platform that is responsible for allocating resources to application servers running on it and ensuring desired QoS in terms of performance stability during extreme overload. The server machines in their system are able to automatically adapt to changes in workload.

To the best of our knowledge, our system is the first SLA-driven resource manager for compute clouds based on open source technology. Our working prototype, built on top of a EUCALYPTUS-based compute cloud, provides adaptive resource allocation and dynamic load balancing for Web applications in order to satisfy a SLA that enforces specific response time requirements. We evaluate the prototype on a heterogeneous testbed cloud and demonstrate that it is able to detect SLA violations from individual computational resources and perform adaptive resource management to satisfy the SLA.

There are a few limitations to this preliminary work. We only address the application server tier, not the database tier or network. Our prototype is only able to scale up, although it would also be easy to enable the system to scale down by detecting the ends of traffic spikes. Finally, cloud providers using our approach to response time-driven SLAs would need to protect themselves with a detailed contract (imagine for example the rogue application owner who purposefully inserts delays in order to force SLA violations). We plan to address some of these limitations in future work.

In the rest of this paper, we describe our approach, the prototype implementation, and an experimental evaluation of the prototype.

2 System Design and Implementation

To manage cloud resources dynamically based on response time requirements, we developed two components, `VLBCoordinator` and `VLBManager`, in Java. We use Nginx [8] as a load balancer because it offers detailed logging and allows reloading of its configuration file without termination of existing client sessions.

`VLBCoordinator` interacts with the EUCALYPTUS cloud using Typica [9]. Typica is a simple API written in Java to access a variety of Amazon Web services such as EC2, SQS, SimpleDB, and DevPay. Currently, Typica is not able to interact with EUCALYPTUS-based clouds, so we patched it to allow interaction with EUCALYPTUS. The core functions of `VLBCoordinator` are `instantiateVirtualMachine` and `getVMIP`, which are accessible through XML-RPC.

`VLBManager` monitors the logs of the load balancer and detects violations of response time requirements. It reads the load balancer logs in real time and calculates the average response time for each virtual machine in a Web farm over intervals of 60 seconds. Whenever it detects that the average response time of any virtual machine exceeds the required response time, it invokes the `instantiateVirtualMachine` method of `VLBCoordinator` with the required parameters and obtains a new instance ID. After obtaining the instance ID, `VLBManager` waits for 20 seconds (the maximum time it takes for a VM to boot in our system) then obtains the new instance's IP address using an XML-RPC call to `VLBCoordinator`. After receiving the IP address of the newly instantiated virtual machine, it updates the configuration file of the load balancer then sends it a signal requesting it to reload the configuration file.

`VLBManager` executes as a system daemon. Nginx's proxy log entries record among other information the node that serves each specific request. `VLBManager` reads these log entries for 60 seconds and calculates the average response time of each node. If it finds that the average response time of any node is greater then required response time, it makes an asynchronous call to `VLBCoordinator` that adaptively launch a new virtual machine and add that virtual machine to the Web farm controlled by the Nginx load balancer. Following is the pseudocode for the main use case of `VLBManager`.

1: set $SLA_{rt} = 2.0$ {Maximum Response time (seconds) allowed in SLA}
2: set $isScaling = false$
3: **while** $true$ **do**
4: vlbManager.ReadNginxLogEnteries(60)
5: **for** each $node$ in $nginxWebFarm$ **do**
6: $vlbManager.calculateAvgRT(node)$
7: **end for**
8: **if** Avg_{rt} of any node $> SLA_{rt}$ and $isScaling == false$ **then**
9: $isScaling = true$
10: $instanceId = VLBCoordinator.instantiateVirtualMachine()$
11: $vmip = vlbCoordinator.getVMIP(instanceId)$
12: $vlbManager.addVMtoNginxWebFarm(vmip)$

13: $isScaling = false$
14: **end if**
15: **end while**

The Boolean *isScaling* is used to prevent concurrent invocations of the scale-up procedure. `VLBManager` creates a child thread to interact with `VLBCoordinator` after detection of response time requirements violation at Line 8.

3 Experiments

In this section we describe the setup for an experimental evaluation of our prototype based on a testbed cloud, a sample Web application, and a synthetic workload generator.

3.1 Testbed Cloud

We built a small heterogeneous compute cloud using four physical machines. Table 1 shows the hardware configuration of the machines.

Table 1. Hardware configuration of physical machines used for the experimental compute cloud

Node	Type	CPU	RAM
Front end	Intel Pentium	2.80 GHz	2 GB
Node1	Intel Pentium	2.66 GHz	1.5 GB
Node2	Intel Celeron	2.4 GHz	2 GB
Node3	Intel Core 2 Duo	1.6 GHz	1 GB

We used EUCALYPTUS to establish a cloud architecture comprising one Cloud Controller (CLC), one Cluster Controller (CC), and three Node Controllers (NCs). We installed the CLC and CC on a front-end node attached to both our main LAN and the cloud's private network. We installed the NCs on three separate machines (Node1, Node2, and Node3) connected to the private network.

3.2 Sample Web Application and Workload Generation

The CPU is normally the bottleneck in the generation of dynamic Web content [10]. We therefore built a simple synthetic Web application consisting of one Java servlet that emulates the real behavior of dynamic web applications by alternating between a CPU intensive job and idle periods, simulating access to non-compute resources such as network connections and local I/O. The servlet accepts two parameters, a baseline time and the number of iterations to perform, and performs a matrix calculation up to the baseline time for the given number of iterations. We used `httperf` to generate synthetic workload for our experiments. We generate workload for a specific duration with a required number of user sessions per second.

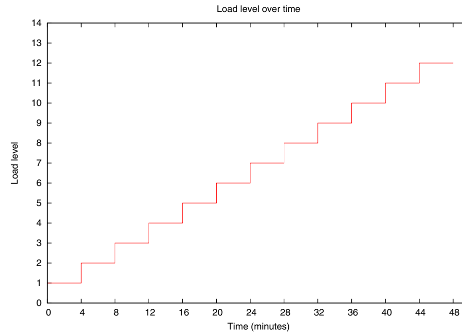


Fig. 1. Workload generation for both experiments. We linearly step the load level from load level 1 through load level 12 every four minutes. Each load level represents the number of user sessions per second, and each session involves 10 requests to the sample Web application.

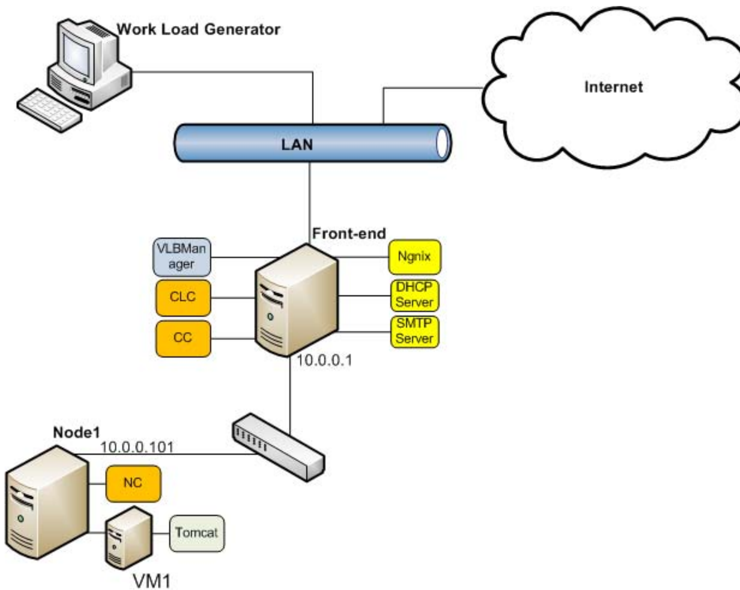


Fig. 2. Experimental setup for Experiment 1 (static resource allocation). The Nginx-based Web farm consists of one virtual machine (VM1) running the Web application. `VLBManager` is only used to obtain the average response time by actively monitoring the Nginx logs.

Each user session makes 10 requests to the application including 4 pauses to simulate user think time. We performed two experiments based on this application. Experiment 1 profiles the system's behavior with static allocation of resources to the application. Experiment 2 profiles the system's behavior under adaptive

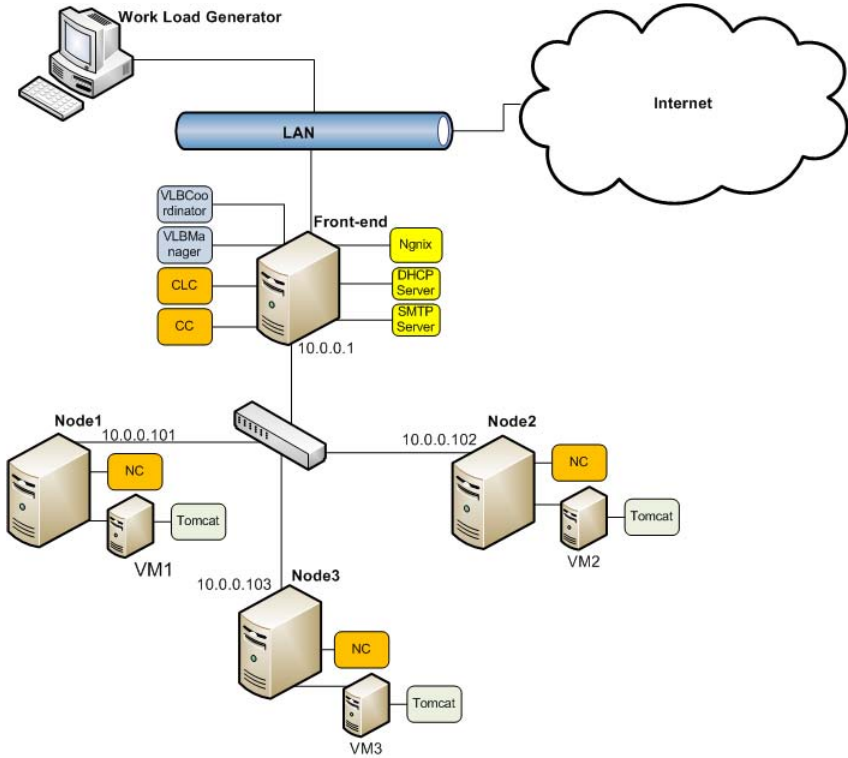


Fig. 3. Experimental setup for Experiment 2 (dynamic resource allocation). The Web farm is initialized with one virtual machine (VM1), while VM2 and VM3 are cached using EUCALYPTUS. `VLBManager` monitors the Nginx logs and detects violations of the SLA. `VLBCoordinator` adaptively invokes additional virtual machines as required to satisfy the SLA.

allocation of resources to the application to satisfy specific response time requirements. The same workload, shown in Figure 1, is generated for both experiments. We linearly step the load level from load level 1 through load level 12 every four minutes. Each load level represents the number of user sessions created per second, and each session involves 10 requests to the sample Web application.

3.3 Experiment 1: Static Allocation

In this experiment, we established the experimental setup shown in Figure 2, in which only one virtual machine (VM1) hosts the Web application. We installed the Nginx load balancer on our front-end node and the Apache Tomcat application server on the virtual machine. The Nginx-based Web farm thus consists of only one virtual machine (VM1). `VLBManager` is only used to obtain the average response time by actively monitoring the Nginx logs.

3.4 Experiment 2: Adaptive Allocation

In this experiment, we used our proposed system to prevent response time increases and rejection of requests by the Web server. Figure 3 shows the experimental setup we established for this experiment. The Nginx-based Web farm is initialized with one virtual machine (VM1), while VM2 and VM3 are cached using EUCALYPTUS. In this experiment, we try to satisfy a Service Level Agreement (SLA) that enforces a two-second maximum average response time requirement for the sample Web application regardless of load level. We use `VLBManager` to monitor the Nginx logs and detect violations of the SLA. We use `VLBCoordinator` to adaptively invoke additional virtual machines as required to satisfy the SLA.

4 Results

4.1 Experiment 1: Static Allocation

This section describes the results we obtained in Experiment 1. Figure 4 shows the CPU utilization of VM1 during Experiment 1. After load level 5, the CPU is almost fully utilized by the Tomcat application server. We observe downward spike in the beginning of each load level because all user sessions are cleared between load levels and it takes some time for the system to return to a steady state.

Figure 5(a) shows the number of requests served by our system. After load level 6, we do not observe any growth in the number of served requests because the sole Web server reaches its saturation point. Although the load level increases with time, the system is unable to serve all requests, and it either rejects or queues the remaining requests. Figure 5(b) shows the number of requests rejected by our system during Experiment 1.

Figure 6 shows the average response time we observed during each load level. From load level 1 to load level 5, we observe a nearly constant response time, but

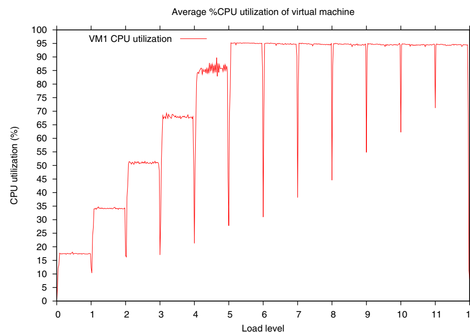
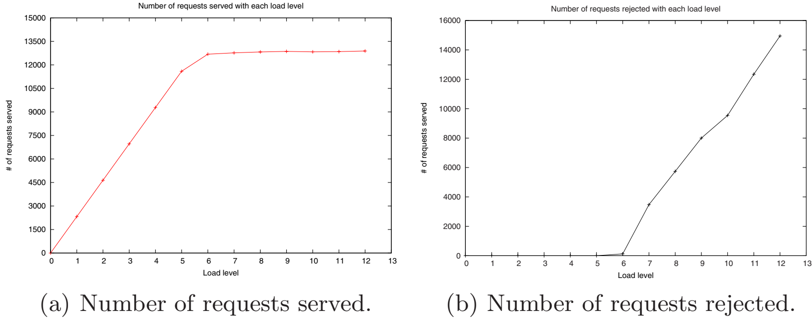


Fig. 4. CPU utilization of VM1 during Experiment 1. The duration of each load level is 4 minutes. The CPU is saturated at load level 5.



(a) Number of requests served. (b) Number of requests rejected.

Fig. 5. Number of served and rejected requests during Experiment 1. The duration of each load level is 4 minutes. After load level 6, we do not observe any growth in the number of served requests, because the Web server reaches its saturation point. As the load level increases with time, the system is increasingly unable to serve requests and rejects more requests.

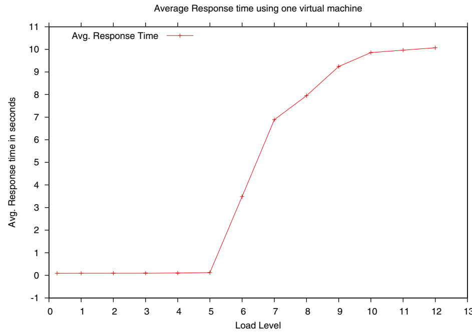
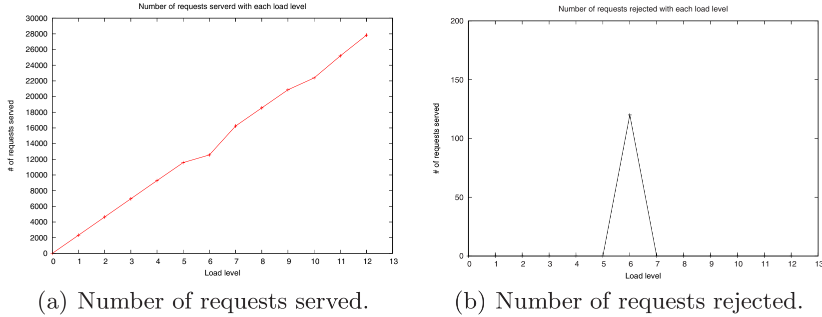


Fig. 6. Average response time for each load level during Experiment 1. The duration of each load level is 4 minutes. From load level 1 to load level 5, we observe a nearly constant response time. After load level 5, we see rapid growth in the average response time. When the Web server reaches the saturation point, requests spend more time in the queue, and the system rejects some incoming requests. From load level 6 to load level 10, some requests spend time in the queue and few requests get rejected. After load level 10, the Web server queue is also saturated, and the system rejects most requests.

after load level 5, the arrival rate exceeds the limit of the Web server's processing capability, so requests spend more time in the queue, and response time grows rapidly. From load level 5 to load level 10, requests spend more time in the queue and relatively few requests get rejected. After load level 10, however, the queue also becomes saturated, and the system rejects most requests. Therefore we do not observe further growth in the average response time.

Clearly, we cannot provide a SLA guaranteeing a specific response time with an undefined load level for a Web application using static resource allocation.



(a) Number of requests served. (b) Number of requests rejected.

Fig. 7. Number of served and rejected requests by system during Experiment 2 with adaptive resource allocation. The number of requests served by system grows linearly with the load level while only 120 requests are rejected during the first violation of response time requirements.

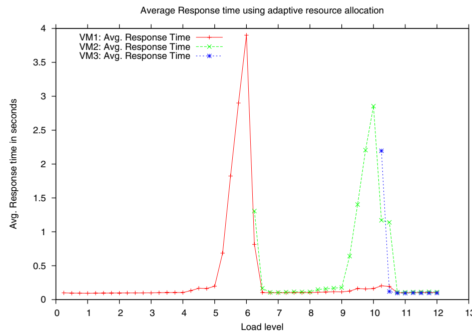


Fig. 8. Average response time for each virtual machine during Experiment 2. Whenever the system detects a violation of response time requirements in any virtual node, it dynamically creates another virtual machine and adds it to the Web farm.

4.2 Experiment 2: Adaptive Allocation

This section describes the results of Experiment 2. Figure 7(a) shows the number of requests served by the system over time. We observe linear growth in the number of requests served by the system with each load level. Figure 7(b) shows the number of requests rejected during Experiment 2. Only 120 requests are rejected during the first violation of response time requirements.

Figure 8 shows the average response time of each virtual machine as it is adaptively added to the Web farm. Whenever the system detects a violation of the response time requirement from any virtual machine, it dynamically invokes another virtual machine and adds it to the Web farm. We observe continued violation of the required response time for a period of time due to the latency of virtual machine boot-up.

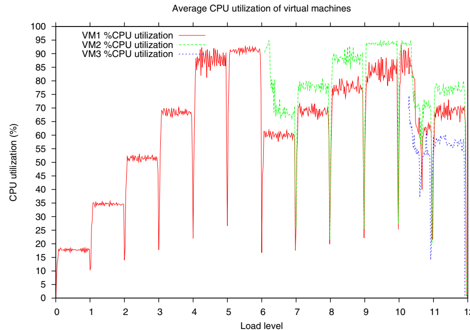


Fig. 9. CPU utilization of virtual machines during Experiment 2. The duration of each load level is four minutes. After load level 6, VM2 is adaptively added to the Web farm to satisfy the response time requirement. After load level 10, VM3 is adaptively added to the Web farm. Different load levels for different VMs reflect the use of round robin balancing and differing processor speeds for the physical nodes.

Figure 9 shows the CPU utilization of each virtual machine over the experiment. After load level 6, VM2 is adaptively added to the Web farm to satisfy the response time requirement. After load level 10, VM3 is adaptively added to the Web farm. Different load levels for different VMs reflect the use of round-robin balancing and differing processor speeds for the physical nodes.

The experiments show that adaptive management of resources on compute clouds for Web applications would allow us to offer SLAs that enforce specific response time requirements. To avoid continued violation of the SLA during VM boot-up, it would be better to predict response time requirement violations rather than waiting until the requirement is violated.

5 Conclusion and Future Work

In this paper, we have described a prototype system based on EUCALYPTUS that actively monitors the response time of a Web application hosted on a cloud and adaptively scales up the compute resources of the Web application to satisfy a SLA enforcing specific response time requirements. Adaptive resource management in clouds would allow cloud providers to manage resources more efficiently and would allow consumers (owners of Web applications) to maintain the usability of their applications.

We use the log-based approach to monitor Web applications and detect response time violations on the basis of the actual time it takes to service requests. The main benefit of this approach is that it does not require any modification of the application or adding components to the user's virtual machines. An event-based approach such as CPU utilization monitoring could be used, but this would not guarantee satisfaction of the SLA. CPU utilization or other event based approaches, while not sufficient in isolation, could be used in tandem with

log-based response time monitoring to help predict future increases in response time.

We are extending our system to scale down Web applications when appropriate, and we are planning to predict VM response times in advance to overcome the virtual machine boot-up latency problem. We also plan to use fair balancing instead of round robin balancing to eliminate the need to check each VM's response time. Finally, we plan to port our system to the Amazon Web Services infrastructure.

Acknowledgments

This work was partly supported by a graduate fellowship from the Higher Education Commission (HEC) of Pakistan to WI.

References

1. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The EUCALYPTUS Open-source Cloud-computing System. In: CCA 2008: Proceedings of the Cloud Computing and Its Applications Workshop, Chicago, IL, USA (2008)
2. Amazon Inc: Amazon web services auto scaling (2009), `http://aws.amazon.com/autoscaling/`
3. Azeez, A.: Auto-scaling web services on amazon ec2 (2008), `http://people.apache.org/~azeez/autoscaling-web-services-azeez.pdf`
4. Shivam, P., Demberel, A., Gunda, P., Irwin, D., Grit, L., Yumerefendi, A., Babu, S., Chase, J.: Automated and on-demand provisioning of virtual machines for database applications. In: SIGMOD 2007: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 1079–1081. ACM, New York (2007)
5. Irwin, D., Chase, J., Grit, L., Yumerefendi, A., Becker, D., Yocum, K.G.: Sharing networked resources with brokered leases. In: ATEC 2006: Proceedings of the Annual Conference on USENIX 2006 Annual Technical Conference, p. 18. USENIX Association, Berkeley (2006)
6. Shivam, P., Babu, S., Chase, J.: Active and accelerated learning of cost models for optimizing scientific applications. In: VLDB 2006: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB Endowment, pp. 535–546 (2006)
7. Guitart, J., Carrera, D., Beltran, V., Torres, J., Ayguadé, E.: Dynamic CPU provisioning for self-managed secure Web applications in SMP hosting platforms. Computer Network 52(7), 1390–1409 (2008)
8. Sysoev, I.: Nginx (2002), `http://nginx.net/`
9. Google Code: Typica: A Java client library for a variety of Amazon Web Services (2008), `http://code.google.com/p/typica/`
10. Challenger, J.R., Dantzig, P., Iyengar, A., Squillante, M.S., Zhang, L.: Efficiently serving dynamic data at highly accessed web sites. IEEE/ACM Transactions on Networking 12, 233–246 (2004)