SLA-Driven Automatic Bottleneck Detection and Resolution for Read Intensive Multi-tier Applications Hosted on a Cloud

Waheed Iqbal[1], Matthew N. Dailey[1], David Carrera[2], and Paul Janecek[1]

[1] Computer Science and Information Management, Asian Institute of Technology, Thailand
[2] Technical University of Catalonia (UPC), Barcelona Supercomputing Center (BSC), Spain

Abstract. A Service-Level Agreement (SLA) provides surety for specific quality attributes to the consumers of services. However, the current SLAs offered by cloud providers do not address *response time*, which, from the user's point of view, is the most important quality attribute for Web applications. Satisfying a maximum average response time guarantee for Web applications is difficult for two main reasons: first, traffic patterns are unpredictable; second, the complex nature of multi-tier Web applications increases the difficulty of identifying bottlenecks and resolving them automatically. This paper presents a working prototype system that automatically detects and resolves bottlenecks in a multi-tier Web application hosted on a EUCALYPTUS-based cloud in order to satisfy specific maximum response time requirements. We demonstrate the feasibility of the approach in an experimental evaluation with a testbed cloud and a synthetic workload. Automatic bottleneck detection and resolution under dynamic resource management has the potential to enable cloud providers to provide SLAs for Web applications that guarantee specific response time requirements.

1 Introduction

Cloud service providers allow consumers to rent computational and storage resources on demand and according to their usage. Cloud service providers 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. Guaranteeing response time is a difficult problem for two main reasons. First, web application traffic is highly unpredictable. Second, the complex nature of multi-tier Web applications, in which bottlenecks can occur at multiple points, means response time violations may not be easy to diagnose or remedy. If a cloud provider is to guarantee a particular maximum response time for any traffic level, it must automatically detect bottleneck tiers and allocate additional resources to those tiers 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] cloud that actively monitors the response times for requests to a multi-tier Web application, gathers CPU usage statistics, and uses heuristics to identify the bottlenecks. When bottlenecks are identified, the system dynamically allocates the resources required by the application to resolve the identified bottlenecks and maintain response time requirements. We consider a two-tier Web application consisting of a Web server tier and a database tier to evaluate our proposed approach.

There have been several efforts to perform dynamic scaling of applications based on workload monitoring. In previous work [2] we considered single-tier Web applications, used log-based monitoring to identify SLA violations along, and used dynamic resource allocation to satisfy the SLA. Amazon Auto Scaling [3] allows consumers to scale up or down according to criteria such as average CPU utilization across a group of compute instances. [4] presents a statistical machine learning approach to predict system performance and minimize the number of resources required to maintain the performance of an application hosted on a cloud. [5] dynamically switches between different virtual machine configurations to satisfy changing workloads on an Amazon EC2 cloud. However, none of these solutions address the issues of multi-tier Web applications.

To the best of our knowledge, our system is the first SLA-driven resource manager for clouds based on open source technology. Our working prototype, built on top of a EUCALYPTUS-based compute cloud, provides dynamic resource allocation and load balancing for multi-tier Web applications in order to satisfy a SLA that enforces specific response time requirements. In this paper, we describe our prototype, the heuristics we have developed for multi-tier Web applications, and an evaluation of the prototype on a testbed cloud. We find that the system is able to detect bottlenecks, resolve them using dynamic resource allocation, and satisfy the SLA.

There are a few limitations to this preliminary work. We only address scaling of the application server tier and a read-only database tier. 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. Our proposed system is also not capable of reversing scaling operations that turn out not to be helpful for bottleneck resolution. 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 three components: `VLBCoordinator`, `VLBManager`, and `VMProfiler`. We use Nginx [6] 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 a EUCALYPTUS cloud using Typica [7]. Typica is a simple API written in Java to access a variety of Amazon Web services such as EC2, SimpleDB, and DevPay. The core functions of `VLBCoordinator` are `instantiateVirtualMachine` and `getVMIP`, which are accessible through XML-RPC. `VLBManager` monitors the traces of the load balancer and detects violations of response time requirements. It clusters the requests into static and dynamic resource requests and calculates the average response time for each type of request. `VMProfiler` is used to log the CPU utilization of each virtual machine. It exposes XML-RPC functions to obtain the CPU utilization of specific virtual machine for the last n minutes.

RUBiS [8] is an open-source benchmark Web application for auctions. It provides core functionality of an auction site such as browsing, selling, and bidding for items, and provides three user roles: visitor, buyer, and seller. Visitors are not required to register and are allowed to browse items that are available for auction. We used the PHP implementation of RUBiS as a sample Web application to evaluate our system. Since RUBiS does not currently support load balancing over a database tier, we modified it to use round-robin balancing over a set of database servers listed in a database connection settings file, and we developed a server-side component, `DbConfigAgent`, to update the database connection settings file after a scaling operation has modified the configuration of the database tier. The entire benchmark system consists of the physical machines supporting the EUCALYPTUS cloud, a virtual Web server acting as a proxying load balancer for the entire Web application, a tier of virtual Web servers running the RUBiS application software, and a tier of virtual database servers. Figure 1 shows the deployment of our components along with the main interactions.

We use heuristics and active profiling of the CPUs of virtual machine-hosted application tiers for identification of bottlenecks. Our system reads the Web server proxy logs for 60 seconds and clusters the log entries into dynamic content requests and static content requests. Requests to resources (Web pages) containing server-side scripts (PHP, JSP, ASP, etc.) are considered as dynamic content requests. Requests to the static resources (HTML, JPG, PNG, TXT, etc.) are considered as static content requests. Dynamic resources are generated through utilization of the CPU and may depend on other tiers, while static resources are pre-generated flat files available in the Web server tier. Each type of request has different characteristics and is monitored separately for purposes of bottleneck detection. The system calculates the 95^{th} percentile of the average response time. When static

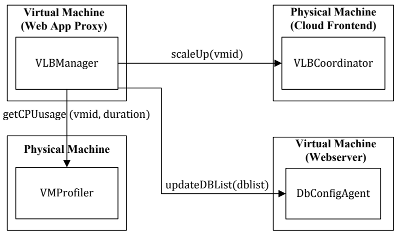

Virtual Machine (Web App Proxy)
VLBManager
scaleUp(vmid)
Physical Machine (Cloud Frontend)
VLBCoordinator
getCPUusage (vmid, duration)
Physical Machine
VMProfiler
updateDBList(dblist)
Virtual Machine (Webserver)
DbConfigAgent

Fig. 1. Component deployment diagram for system components including main interactions

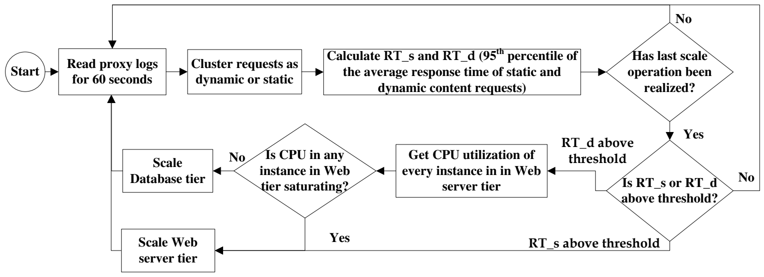

Start
Read proxy logs for 60 seconds
Cluster requests as dynamic or static
Calculate RT_s and RT_d (95th percentile of the average response time of static and dynamic content requests)
Has last scale operation been realized?
No
Yes
Is RT_s or RT_d above threshold?
No
RT_d above threshold
RT_s above threshold
Get CPU utilization of every instance in in Web server tier
Is CPU in any instance in Web tier saturating?
No
Yes
Scale Database tier
Scale Web server tier

Fig. 2. Flow diagram for prototype system that detects the bottleneck tier in a two-tier Web application hosted on a heterogeneous cloud and dynamically scales the tier to satisfy a SLA that defines response time requirements

content response time indicates saturation, the system scales the Web server tier. When the system determines that dynamic content response time indicates saturation, it obtains the CPU utilization across the Web server tier. If the CPU utilization of any instance in the Web server tier has reached a saturation threshold, the system scales up the Web server tier; otherwise, it scales up the database tier. Each scale operation adds exactly one server to a specific tier. Our focus is on read-intensive applications, and we assume that a mechanism such as [9] exists to ensure consistent reads after updates to a master database. Before initiating a scale operation, the system ensures that the effect of the last scale operation has been realized. Figure 2 shows a flow diagram for bottleneck detection and resolution in our prototype system.

3 Experiments

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

3.1 Testbed Cloud

We built a small private heterogeneous compute cloud on five physical machines (Front-end, Node1, Node2, Node3, and Node4) using EUCALYPTUS. Front-end, Node1, and Node4 are Intel Pentium 4 machines with 2.84GHz, 2.66 GHz, and 2.66 GHz CPUs, respectively. Node2 is an Intel Celeron machine with a 2.4 GHz CPU. Node3 is an Intel Core 2 Duo machine with 2.6 GHz CPU. Front-end, Node2 and Node3 have 2 GB RAM while Node1 and Node4 have 1.5 GB RAM.

We used EUCALYPTUS to establish a cloud architecture comprised of one Cloud Controller (CLC), one Cluster Controller (CC), and four 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 four (Node1, Node2, Node3, and Node4) separate machines connected to the private network.

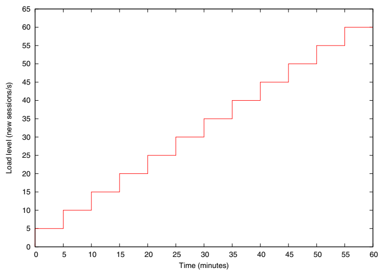


Fig. 3. Workload generation profile for all experiments

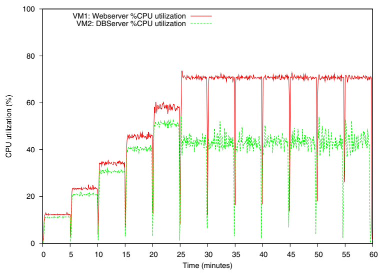


Fig. 4. CPU utilization of virtual machines used during Experiment 1

3.2 Workload Generation

We use `httperf` to generate synthetic workloads for RUBiS. We generate workloads for specific durations with a required number of user sessions per second. A user session emulates a visitor that browses categories and regions and also bids on items up for auction. Every five minutes, we increment the load level by 5, from load level 5 through load level 60. Each load level represents the number of user sessions per second; each user session makes 6 requests to static resources and 5 requests to dynamic resources including 5 pauses to simulate user think time. The dynamic resources consist of PHP pages that make read-only database queries. Figure 3 shows the workload levels we use for our experiments over time.

We performed four experiments based on this workload and RUBiS. Experiments 1, 2, and 3 profile the system's behavior using static allocation with different static resource allocations. Experiment 4 profiles the system's behavior under dynamic resource allocation using the proposed algorithm for bottleneck detection and resolution. We generate the same workload for each experiment except that Experiments 2 and 3 start at load level 25 instead of load level 5.

3.3 Experiment Details

In Experiment 1, we statically allocate one virtual machine to the Web server tier and one virtual machine to the database tier. In Experiment 2, we statically allocate 1 virtual machine for the Web server tier and a cluster of 2 virtual machines for the database tier. As discussed earlier, we modified RUBiS to perform load balancing across the database server instances; in this experiment, RUBiS performs load balancing across the static database server cluster. In Experiment 3, we statically allocate a cluster of 2 virtual machines for the Web server tier and 1 virtual machine for the database tier. In this experiment, Nginx performs load balancing of HTTP traffic across the static Web server cluster.

In Experiment 4, we use our proposed system to adapt to response time increases and rejection of requests by the Web server. Initially, we started two virtual machines on our testbed cloud. The Nginx-based Web server farm was initialized with one virtual machine hosting the Web server tier. Another single

Table 1. Summary of experiments

Exp.	Description
1	Static allocation using 1 VM for Web server tier and 1 VM for database tier.
2	Static allocation using 1 VM for Web server tier and cluster of 2 VMs for database tier.
3	Static allocation using cluster of 2 VMs for Web server tier and 1 VM for database tier.
4	Dynamic allocation using proposed system for bottleneck detection and resolution.

virtual machine was used to host the database tier. In this experiment, we tried to satisfy a SLA that enforces a one-second maximum average response time requirement for the RUBiS application regardless of load level using our proposed algorithm for bottleneck detection and resolution. The threshold for CPU saturation was set to 85% utilization. Table 1 summarizes the experiments.

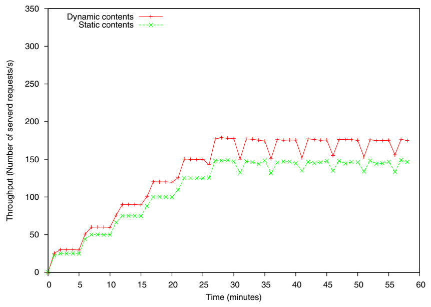
4 Results

4.1 Experiment 1: Simple Static Allocation

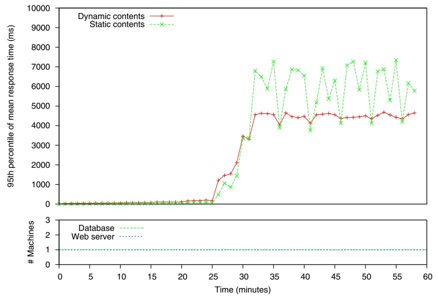
This section describes the results we obtained in Experiment 1. Figure 4 shows the CPU utilization of the 2 virtual machines hosting the application tiers during Experiment 1. The downward spikes at the beginning of each load level occur because all user sessions are cleared between load level increments, and it takes some time for the system to return to a steady state. We do not observe any tier saturating its CPU during this experiment: after load level 25, the CPU utilization remains nearly constant, indicating that the CPU was not a bottleneck for this application with the given workload.

Figure 5(a) shows the throughput of the system during Experiment 1. After load level 25, we do not observe any growth in the system's throughput because one or both of the tiers have reached their saturation points. 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 95^{th} percentile of average response time during Experiment 1. From load level 5 to load level 25, we observe a nearly constant response time, but after load level 25, the arrival rate exceeds the limits of the system's processing capability. One of the virtual machines hosting the application tiers becomes a bottleneck, then requests begin to spend more time in the queue and request processing time increases. From that point we observe a rapid growth in the response time. After load level 30, 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, the system only works efficiently from load level 5 to load level 25.



(a) Throughput of the system.



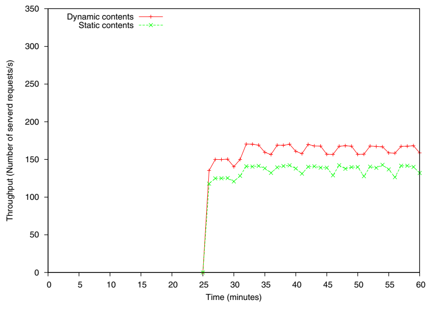
(b) 95th percentile of mean response time.

Fig. 5. Throughput and response time in Experiment 1

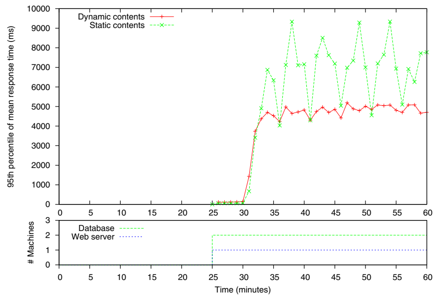
4.2 Experiment 2: Static Allocation Using Database Tier Replication and Load Balancing

We observed in Experiment 1 that after load level 25, the system's throughput and response time saturate. In Experiment 2, to test whether static allocation of an additional database server would resolve the bottleneck, we allocated two virtual machines to the database tier and one virtual machine to the Web server tier. We generated workload from load level 25 to load level 60. Figure 6(a) shows the throughput of the system during Experiment 2. After load level 30, we do not observe growth in the system throughput, because the system requires still more resources to satisfy the needs of the incoming workload.

Figure 6(b) shows the 95th percentile of the average response time during Experiment 2. From load level 25 to load level 30, we observe a nearly constant response time. We see that allocating more resources to the database tier helps the system to support an increase of 5 in the load level, and the system is able to satisfy response time requirements up to load level 30, but after load level 30, the arrival rate exceeds the limit of the system's processing capability and we observe a dramatic increase in system response time.

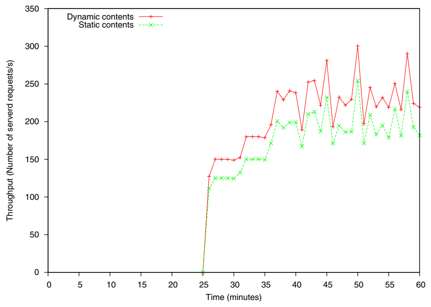


(a) Throughput of the system.

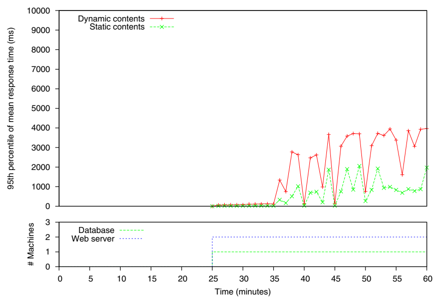


(b) 95th percentile of mean response time.

Fig. 6. Throughput and response time in Experiment 2



(a) Throughput of the system.



(b) 95th percentile of mean response time.

Fig. 7. Throughput and response time in Experiment 3

4.3 Experiment 3: Static Allocation Using Web Server Tier Replication and Load Balancing

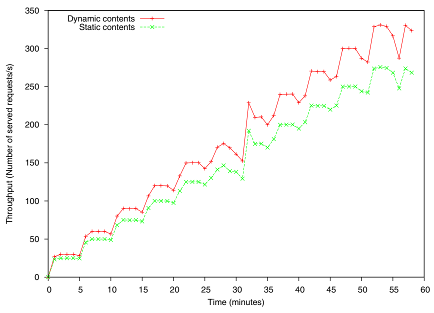
In Experiment 3, to test whether static allocation of an additional Web server would resolve the bottleneck, we allocated two virtual machines to the Web server tier and one virtual machine to the database tier, and generated a workload from load level 25 to load level 60. Figure 7(a) shows the throughput of the system during Experiment 3. We do not observe growth in the system throughput after load level 35, because the system requires still more resources to satisfy the needs of the incoming requests.

Figure 7(b) shows the 95th percentile of average response times during Experiment 3. From load level 25 to load level 35, we observe a nearly constant response time, but after load level 35, the arrival rate exceeds the limit of the system's processing capacity and we observe a dramatic increase in the system response time.

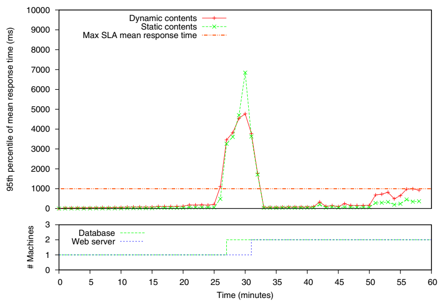
Experiments 1, 2, and 3 show that it is difficult to obtain the best resource allocation for the system to satisfy response time requirements for an undefined load level. Clearly, we cannot provide a SLA guaranteeing a specific response time with an undefined load level for a multi-tier Web application using static resource allocation.

4.4 Experiment 4: Bottleneck Detection and Resolution under Dynamic Allocation

This section describes the results of Experiment 4. Figure 8(b) shows the 95th percentile of the average response time during Experiment 4 using automatic bottleneck detection and dynamic resource allocation. The bottom graph shows the adaptive addition of instances in each tier after bottleneck detection during the experiment. Whenever the system detects a violation of the response time requirements, it uses the proposed algorithm to identify the bottleneck tier and dynamically add another virtual machine to the farm of that bottleneck tier. We observe violation of the required response time for a period of time due to the latency of virtual machine boot-up and the time required to observe the effects



(a) Throughput of the system.



(b) 95^{th} percentile of mean response time.

Fig. 8. Throughput and response time in Experiment 4

of previous scale operations. Figure 8(a) shows the system throughput during the experiment. We observe linear growth in the system throughput through the full range of load levels.

Figure 9 shows the CPU utilization of all virtual machines during the experiment. Initially, the system was configured with VM1 and VM2. After load level 25, VM3 is dynamically added to the system, and after load level 30, VM4 is dynamically added to the system. The differing steady-state levels of CPU utilization for the different VMs reflects the use of round-robin balancing and differing processor speeds for the physical nodes. We observe the same downward spike at the beginning of each load level as in the earlier experiments due to the time for the system to return to steady state after all user sessions are cleared.

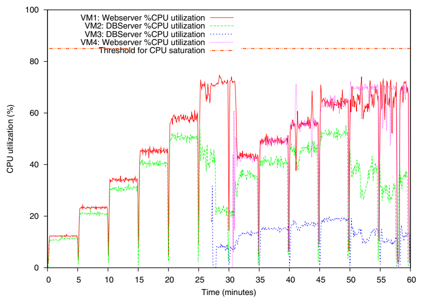


Fig. 9. CPU utilization of virtual machines during Experiment 4

5 Discussion and Conclusion

In this paper, we have described a prototype system that automatically identifies and resolves bottlenecks in a multi-tier application hosted on a EUCALYTPUS-based cloud. Experimental results show that automatic bottleneck detection and resolution can enable us to offer SLAs that define specific maximum response time requirement for multi-tier Web applications.

We are extending our system to support n-tier clustered applications hosted on a cloud, and we are planning to predict bottlenecks in advance to overcome the virtual machine boot-up latency problem.

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. Iqbal, W., Dailey, M., Carrera, D.: SLA-driven adaptive resource management for web applications on a heterogeneous compute cloud. In: Jaatun, M.G., Zhao, G., Rong, C. (eds.) CloudCom 2009. LNCS, vol. 5931, pp. 243–253. Springer, Heidelberg (2009)
3. Amazon Inc.: Amazon Web Services auto scaling (2009), http://aws.amazon.com/autoscaling/
4. Bodik, P., Griffith, R., Sutton, C., Fox, A., Jordan, M., Patterson, D.: Statistical machine learning makes automatic control practical for internet datacenters. In: HotCloud 2009: Proceedings of the Workshop on Hot Topics in Cloud Computing, San Diego, CA (2009)
5. Liu, H., Wee, S.: Web server farm in the cloud: Performance evaluation and dynamic architecture. In: Jaatun, M.G., Zhao, G., Rong, C. (eds.) CloudCom 2009. LNCS, vol. 5931, pp. 369–380. Springer, Heidelberg (2009)
6. Sysoev, I.: Nginx (2002), http://nginx.net/
7. Google Code: Typica: A Java client library for a variety of Amazon Web Services (2008), http://code.google.com/p/typica/
8. OW2 Consortium: RUBiS: An auction site prototype (1999), http://rubis.ow2.org/
9. xkoto: Gridscale (2009), http://www.xkoto.com/products/