Adaptive Resource Provisioning for Read Intensive Multi-tier Applications in the Cloud

Waheed Iqbal[a], Matthew N. Dailey[a], David Carrera[b], Paul Janecek[a]

[a] *Computer Science and Information Management, Asian Institute of Technology, Thailand*
[b] *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, current SLAs offered by cloud infrastructure 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 highly dynamic and difficult to predict accurately; second, the complex nature of multi-tier Web applications increases the difficulty of identifying bottlenecks and resolving them automatically. This paper proposes a methodology and presents a working prototype system for automatic detection and resolution of bottlenecks in a multi-tier Web application hosted on a cloud in order to satisfy specific maximum response time requirements. It also proposes a method for identifying and retracting over-provisioned resources in multi-tier cloud-hosted Web applications. We demonstrate the feasibility of the approach in an experimental evaluation with a testbed EUCALYPTUS-based cloud and a synthetic workload. Automatic bottleneck detection and resolution under dynamic resource management has the potential to enable cloud infrastructure providers to provide SLAs for Web applications that guarantee specific response time requirements while minimizing resource utilization.

Keywords: Cloud computing, Adaptive resource management, Quality of Service, Multi-tier applications, Service-level agreement, Scalability

1. Introduction

Cloud providers [1] use the Infrastructure as a Service model to allow consumers to rent computational and storage resources on demand and according to their usage. Cloud infrastructure providers maximize their profits by fulfilling their obligations to consumers with minimal infrastructure and maximal resource utilization.

Although most cloud infrastructure 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 dynamic and difficult to predict accurately. 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. It is also difficult to determine optimal static resource allocation for multi-tier Web applications manually for certain workloads due to the dynamic nature of incoming requests and exponential number of possible allocation strategies. Therefore, if a cloud infrastructure 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 propose a methodology and present a working prototype system running on a EUCALYPTUS-based [2] 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. The system furthermore predicts the optimal configuration for the dynamically varying workload and scales down the configuration whenever possible to minimize resource utilization.

The bottleneck resolution method is purely *reactive*. Reactive bottleneck resolution has the benefit of avoiding inaccurate a priori performance models and pre-deployment profiling. In contrast, the scale down method is necessarily *predictive*, since we must avoid premature release of busy resources. However, the predictive model is built using application performance statis-

tics acquired *while the application is running* under real-world traffic loads, so it neither suffers from the inaccuracy of a priori models nor requires pre-deployment profiling.

In this paper, we describe our prototype, the heuristics we have developed for reactive scale-up of multi-tier Web applications, the predictive models we have developed for scale-down, and an evaluation of the prototype on a testbed cloud. The evaluation uses a specific two-tier Web application consisting of a Web server tier and a database tier. In this context, the resources to be minimized are the number of Web servers in the Web server tier and the number of replicas in the database tier. We find that the system is able to detect bottlenecks, resolve them using adaptive resource allocation, satisfy the SLA, and free up over-provisioned resources as soon as they are not required.

There are a few limitations to this preliminary work. We only address scaling of the Web server tier and a read-only database tier. Our system only perform hardware and virtual resource management for applications. In particular, we do not address software configuration management; for example, we assume that the number of connections from each server in the Web server tier to the database tier is sufficient for the given workload. Additionally, real-world cloud infrastructure providers using our approach to response time-driven SLAs would need to protect themselves with detailed contracts (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 provide related work, then we describe our approach, the prototype implementation, and an experimental evaluation of the prototype.

2. Related Work

There has been a great deal of research on dynamic resource allocation for physical and virtual machines and clusters of virtual machines [3]. In [4] and [5], a two-level control loop is proposed to make resource allocation decisions within a single physical machine. This work does not address integrated management of a collection of physical machines. The authors of [6] study the overhead of a dynamic allocation scheme that relies on virtualization as opposed to static resource allocation. None of these techniques provide a

technology to dynamically adjust allocation based on SLA objectives in the presence of resource contention.

VMware DRS [7] provides technology to automatically adjust the amount of physical resources available to VMs based on defined policies. This is achieved using the live-migration automation mechanism provided by VMotion. VMware DRS adopts a VM-centric view of the system: policies and priorities are configured on a VM-level.

A approach similar to VMware DRS is proposed in [8], which proposes a dynamic adaptation technique based on rearranging VMs so as to minimize the number of physical machines used. The application awareness is limited to configuring physical machine utilization thresholds based on off-line analysis of application performance as a function of machine utilization. In all of this work, runtime requirements of VMs are taken as a given and there is no explicit mechanism to tune resource consumption by any given VM.

Foster et al. [9] address the problem of deploying a cluster of virtual machines with given resource configurations across a set of physical machines. Czajkowski et al. [10] define a Java API permitting developers to monitor and manage a cluster of Java VMs and to define resource allocation policies for such clusters.

Unlike [7] and [8], our system takes an application-centric approach; the virtual machine is considered only as a container in which an application is deployed. Using knowledge of application workload and performance goals, we can utilize a more versatile set of automation mechanisms than [7], [8], [9], or [10].

Network bandwidth allocation issues in the deployment of clusters of virtual machines has also been studied in [11]. The problem there is to place virtual machines interconnected using virtual networks on physical servers interconnected using a wide area network. VMs may be migrated, but the emphasis is rather than resource scaling, to allocate network bandwidth for the virtual networks. In contrast, our focus is on data center environments, in which network bandwidth is of lesser concern.

There have been several efforts to perform dynamic scaling of Web applications based on workload monitoring. Amazon Auto Scaling [12] allows consumers to scale up or down according to criteria such as average CPU utilization across a group of compute instances. [13] presents the design of an auto-scaling solution based on incoming traffic analysis for Axis2 Web services running on Amazon EC2. [14] 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. [15] monitors the CPU and bandwidth usage of virtual machines hosted on an Amazon EC2 cloud, identifies the resource requirements of applications, and dynamically switches between different virtual machine configurations to satisfy the changing workloads. However, none of these solutions address the issues of multi-tier Web applications or database scalability, a crucial step to dynamically manage multi-tier workloads.

Thus far, only a few researchers have addressed the problem of resource provisioning for multi-tier applications. [16] presents an analytical model using queuing networks to capture the behavior of each tier. The model is able to predict the mean response time for a specific workload given several parameters such as the *visit ratio*, *service time*, and *think time*. However, the authors do not apply their approach toward dynamic resource management on clouds. [17] presents a predictive and reactive approach using queuing theory to address dynamic provisioning for multi-tier applications. The predictive approach is to allocate resources to applications on large time scales such as days and hours, while the reactive approach is used for short time scales such as seconds and minutes. This allows the system to overcome the "flash crowd" phenomenon and correct prediction mistakes made by the predictive model. The technique assumes knowledge of the resource demands of each tier. In addition to the queuing model, the authors also provide a simple black-box approach for dynamic provisioning that scales up all replicable tiers when bottlenecks are detected. However, this work does not address database scalability or releasing of application resources when they are not required. In contrast, our system classifies requests as either dynamic or static and uses a black box heuristic technique to scale up and scale down only one tier at a time. Our scale-up system is reactive in resolving bottlenecks and our scale-down system is predictive in releasing resources.

The most recent work in this area [18] presents a technique to model dynamic workloads for multi-tier Web applications using k-means clustering. The method uses queuing theory to model the system's reaction to the workload and to identify the number of instances required for an Amazon EC2 cloud to perform well under a given workload. Although this work does model system behavior on a per-tier basis, it does not perform multi-tier dynamic resource provisioning. In particular, database tier scaling is not considered.

In our own recent work [19], we consider single-tier Web applications, use log-based monitoring to identify SLA violations, and use dynamic resource

allocation to satisfy SLAs. In [20], we consider multi-tier Web applications and propose an algorithm based on heuristics to identify the bottlenecks. This work uses a simple reactive technique to scale up multi-tier Web applications to satisfy SLAs. The work described in the current paper is an extension of this work. We aim to solve the problem of dynamic resource provisioning for multi-tier Web applications to satisfy a response time SLA with minimal resource utilization. Our method is reactive for scale-up decisions and predictive for scale-down decisions. Our method uses heuristics and predictive models to scale each tier of a given application, with the goal of requiring minimal knowledge of and minimal modification of the existing application. 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.

3. System Design and Implementation Details

3.1. Dynamic provisioning for multi-tier Web applications

Here we describe our methodology for dynamic provisioning of resources for multi-tier Web applications, including the algorithms, system design, and implementation. A high-level flow diagram for bottleneck detection, scale-up decision making, and scale-down decision making in our prototype system is shown in Figure 1.

3.1.1. Reactive model for scale-up

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 t 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

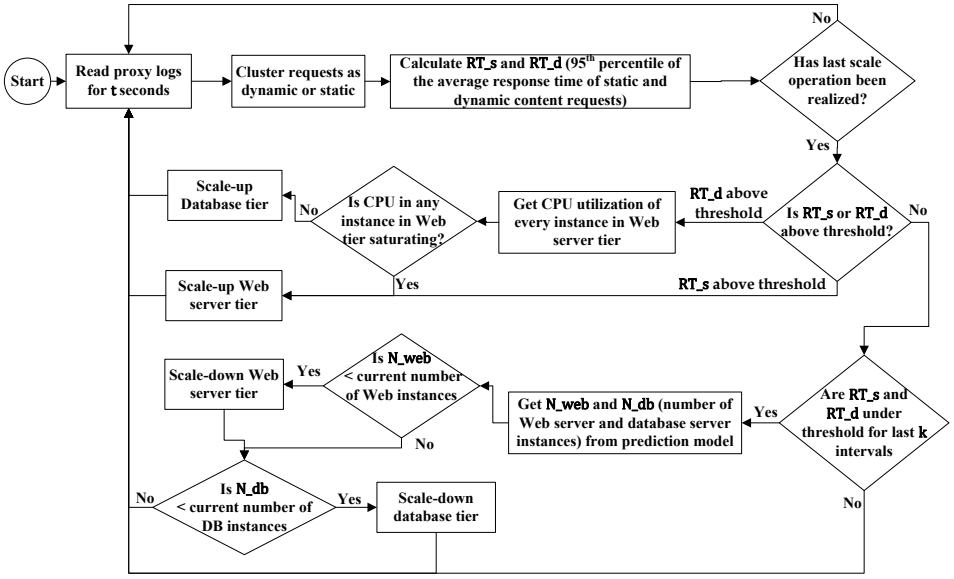


Figure 1: Flow diagram for prototype system that detects the bottleneck tier in a two-tier Web application hosted on a heterogeneous cloud and dynamically scales the tier to satisfy a SLA that defines response time requirements and ensures to release over-provisioned resources.

monitored separately for purposes of bottleneck detection. The system calculates the 95^{th} percentile of the average response time. When static 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 up operation adds exactly one server to a specific tier. Our focus is on read-intensive applications, and we assume that a mechanism such as [21] 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. If the system satisfies the response time requirements for k consecutive intervals, it uses the predictive model to identify any over-provisioned resources and if appropriate, scales down the over-provisioned tier(s). The predictive model is explained next.

3.1.2. Predictive model for scale down

To determine when to initiate scale down operations, we use a regression model that predicts, for each time interval t, the number of Web server instances n_t^{web} and number of database server instances n_t^{db} required for the current observed workload. We use polynomial regression with polynomial degree two. Our reactive scale-up algorithm feeds training observations to the model as appropriate. We retain training observations for every interval of time that satisfies the response time requirements. Each observation contains the observed workload for each type of request and the existing configuration of the tiers for the last 60-second interval. We can express the model as follows:

$$n_t^{web} = a_0 + a_1(h_t^s + h_t^d) + a_2(h_t^s + h_t^d)^2 + \epsilon_t^{web} \quad (1)$$

$$n_t^{db} = b_0 + b_1 h_t^d + b_2(h_t^d)^2 + \epsilon_t^{db}, \quad (2)$$

where h_t^s and h_t^d are the number of static and dynamic requests received during interval t. We assume noise $\epsilon_t^{web} \sim N(0, (\sigma^{web})^2)$ and $\epsilon_t^{db} \sim N(0, (\sigma^{db})^2)$.

Since both static and dynamic resource requests hit the Web server tier, we assume that n_t^{web} (the number of Web server instances required, Equation 1) depends on both h_t^s and h_t^d. To keep the number of model parameters to be estimated small, we use a single parameter for the sum of the two load levels. Since the database server only handles database queries, which are normally only invoked by dynamic pages, we assume that n_t^{db} (the number of database server instances required, Equation 2) depends only on h_t^d.

The regression coefficients a_0, a_1, a_2, b_0, b_1, and b_2 are recalculated after updating the sufficient statistics for all of the historical data every time a new observation is received. (The sufficient statistics are the sums and sums of squares for variables n_t^{web}, n_t^{db}, h_t^s, and h_t^d over the training set up to the current point in time.) The most recent predictive model is used as shown in the flow diagram of Figure 1 to identify over-provisioned resources for the current workload and retract them from the current configuration.

3.2. System components and implementation

To manage cloud resources dynamically based on response time requirements, we developed three components: `VLBCoordinator`, `VLBManager`, and `VMProfiler`. We use Nginx [22] as a load balancer because it offers detailed logging and allows reloading of its configuration file without termination of

existing client sessions. `VLBCoordinator` and `VLBManager` are our service management [23] components.

`VLBCoordinator` interacts with a EUCALYPTUS cloud using Typica [24]. 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.

Every Web application has an application-specific interface between the Web server tier and the database tier. We assume that database writes are handled by a single master MySQL instance and that database reads can be handled by a cluster of MySQL slaves. Under this assumption, we have developed a component for load balancing and scaling the database tier that requires minimal modification of the application.

Our prototype is based on the RUBiS [25] 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 for our experimental evaluation.

To enable RUBiS to support load balancing over the 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 2 shows the deployment of our components along with the main interactions.

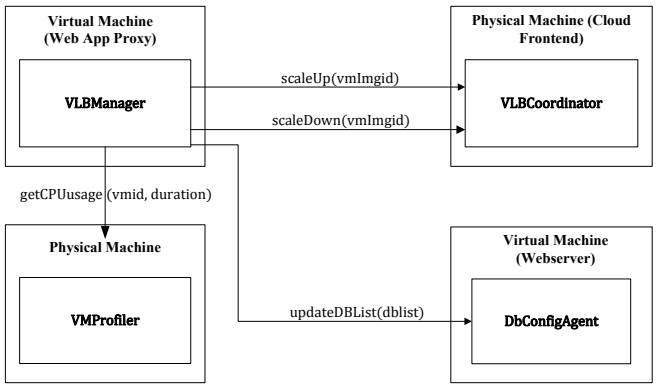


Figure 2: Component deployment diagram for system components including main interactions.

4. Experimental Setup

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.

4.1. Testbed cloud

We built a small private heterogeneous compute cloud on seven physical machines (Front-end, Node1, Node2, Node3, Node4, Node5, and Node6) using EUCALYPTUS. Figure 3 shows the design of our testbed cloud. Front-end and Node1 are Intel Pentium 4 machines with 2.84 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 a 2.6 GHz CPU. Node4, Node5, and Node6 are Intel Pentium Dual Core machines with 2.8 GHz CPU. Front-end, Node2, Node3, Node4, Node5, and Node6 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 six 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 six separate machines (Node1, Node2, Node3, Node4, Node5, and Node6) connected to the private network.

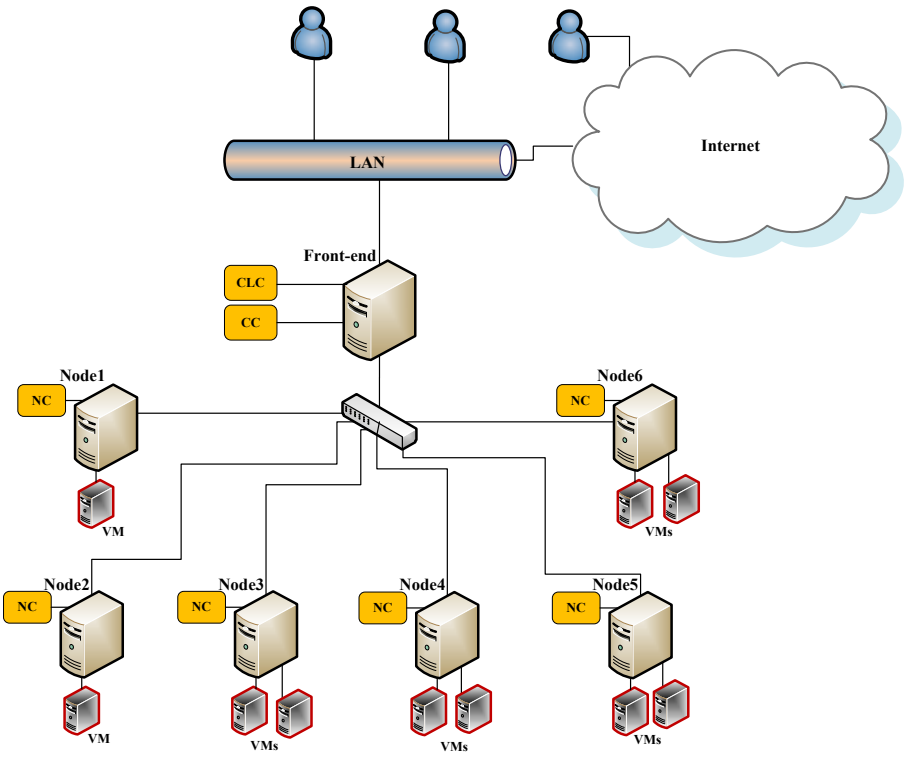


Figure 3: EUCALYPTUS-based testbed cloud using seven physical machines. We installed the CLC and CC on a front-end node attached to both our main LAN and the cloud's private network. We installed the NCs on six separate machines (Node1, Node2, Node3, Node4, Node5, and Node6) connected to the private network. Each physical machine has the capacity to spawn a maximum number of virtual machines as shown (highlighted in red) in the figure, based on the number of cores.

4.2. Workload generation

We use `httperf` [26] 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 items up for auction in specific categories and geographical regions and also bids on items up for auction. In a first cycle, every five minutes, we increment the load level by 6, from load level 6 up to load level 108, and then we decrement the load level by 6 from load level 108 down to load level 6. In a second cycle, we increment the load level by 6, from load level 6 up to load level 60, and then we decrement the load level by 6 from load level 60 down to load

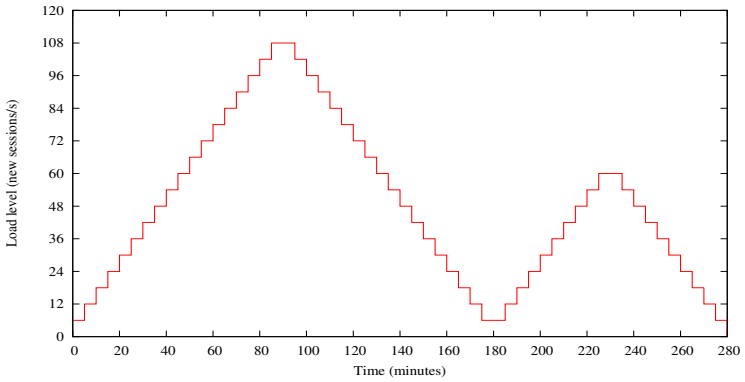


Figure 4: Workload generation profile for all experiments.

level 6. Each load level represents the number of user sessions per second; each user session makes six requests to static resources and five requests to dynamic resources including five pauses to simulate user think time. The dynamic resources consist of PHP pages that make read-only database queries. Note that while each session is closed loop (the workload generator waits for a response before submitting the next request), session creation is open loop: new sessions are created independently of the system's ability to handle them. This means that many requests may queue up, leading to exponential increases in response times. Figure 4 shows the workload levels we use for our experiments over time. We use three workload generators distributed over three separate machines during the experiments to ensure that workload generation machines never reach saturation.

We performed all of our experiments based on this workload generator and RUBiS benchmark Web application.

5. Experimental Design

To evaluate our proposed system, we performed three experiments. Experiments 1 and 2 profile the system's behavior using specific static allocations. Experiment 3 profiles the system's behavior under adaptive resource allocation using the proposed algorithm for bottleneck detection and resolution. Experiments 1 and 2 demonstrate system behavior using current industry practices, whereas Experiment 3 shows the strength of the proposed alternative methodology. Table 1 summarizes the experiments, and details follow.

5.1. Experiment 1: Simple static allocation

In Experiment 1, we statically allocate one virtual machine to the Web server tier and one virtual machine to the database tier, and then we profile system behavior over the synthetic workload described previously. The single Web server/single database server configuration is the most common initial allocation strategy used by most application deployment engineers.

5.2. Experiment 2: Static over-allocation

In Experiment 2, we over-allocate resources, using a maximal static configuration sufficient to process the workload. We statically allocate a cluster of four Web server instances and four database server instances, and then we then profile the system behavior over the synthetic workload described previously. Since it is quite difficult to determine an optimal allocation for a multi-tier application manually, we actually derived this configuration from the the behavior of the adaptive system profiled in Experiment 3.

5.3. Experiment 3: Adaptive allocation under proposed system

In Experiment 3, we use our proposed system to adapt to changing workloads. 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, and another single virtual machine was used to host the database tier. As discussed earlier, we modified RUBiS to perform load balancing across the instances in the database server cluster. The system's goal was 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 (refer to the flow diagram in Figure 1) was set to 85% utilization. This gives the system a chance to handle unexpected spikes in CPU activities, and it is a reasonable threshold for efficient use of the server [27].

To determine good values for the important parameters t (the time to read proxy traces) and k (the number of consecutive intervals required to satisfy response time constraints before a scale-down operation is attempted), we performed a grid search over a set of reasonable values for t and k.

Table 1: Summary of experiments.

Exp.	Description
1	Static allocation using one VM for Web server tier and one VM for database tier
2	Static over-allocation using a cluster of four VMs for the Web server tier and four VMs for database tier
3	Adaptive allocation using proposed methodology

6. Experimental Results

6.1. Experiment 1: Simple static allocation

This section describes the results we obtained in Experiment 1. Figure 5 shows the throughput of the system during the experiment. After load level 30, 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 6 shows the 95^{th} percentile of average response time during Experiment 1. From load level 6 to load level 24, we observe a nearly constant response time, but after load level 24, the arrival rate exceeds the limits of the system's processing capacity. 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 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 6 to load level 24.

Figure 7 shows the CPU utilization of the two 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 30, the CPU utilization remains nearly constant, indicating that the CPU was not a bottleneck for this application with the given workload.

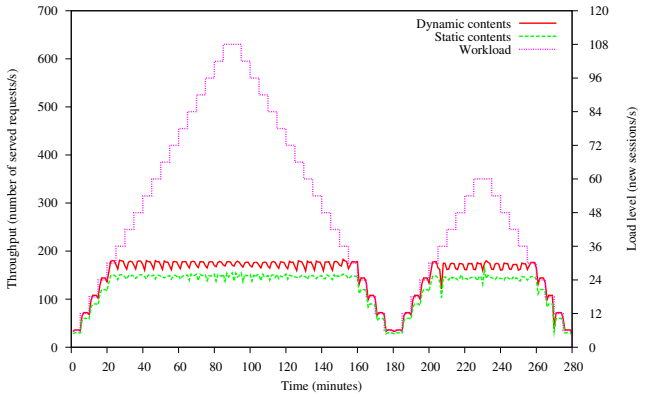


Figure 5: Throughput of the system during Experiment 1.

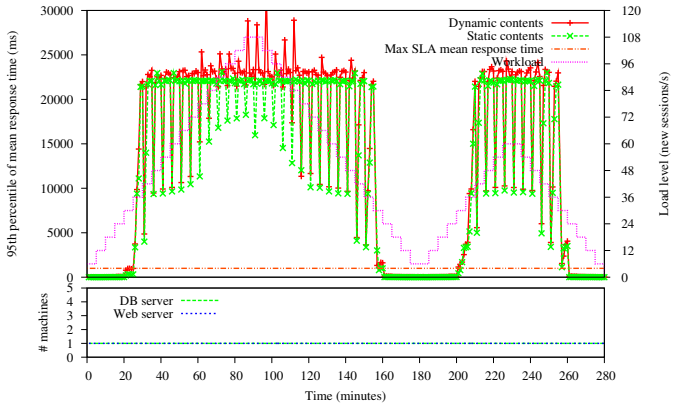


Figure 6: 95^{th} percentile of mean response time during Experiment 1.

6.2. Experiment 2: Static over-allocation

In Experiment 2, to observe the system's behavior under a static allocation policy using the maximal configuration observed during adaptive experiments, we allocated four virtual machines to the Web server tier and four virtual machines to the database tier, and generated the same workload described in Section 4.2. Figure 8 shows the throughput of the system during Experiment 2. We observe the expected linear relationship between load level and throughput; as load level increases, the system throughput increases, and as load level decreases, the system throughput decreases.

Figure 9 shows the 95^{th} percentile of average response times during Experiment 2. We do not observe any response time violations during the

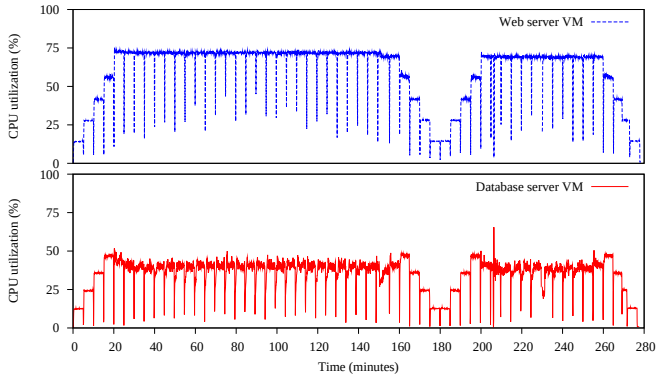


Figure 7: CPU utilization of virtual machines used during Experiment 1.

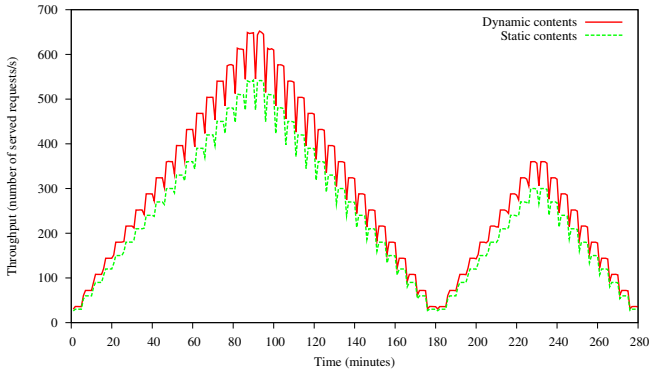


Figure 8: Throughput of the system during Experiment 2.

experiment. We observe a slight increase in response time during load levels 80 to 100 because, during this interval, the system is serving the peak workload and utilizing all of the allocated resources to satisfy the workload requirements. This experiment shows that the maximal configuration identified by our adaptive resource allocation system would never lead to violations of the response time requirements under the same load.

6.3. Experiment 3: Bottleneck detection and resolution under adaptive allocation

This section describes the results of Experiment 3 using our proposed algorithm for bottleneck detection and resolution. We first identified appropriate values and impact for important parameters (t and k) in our proposed

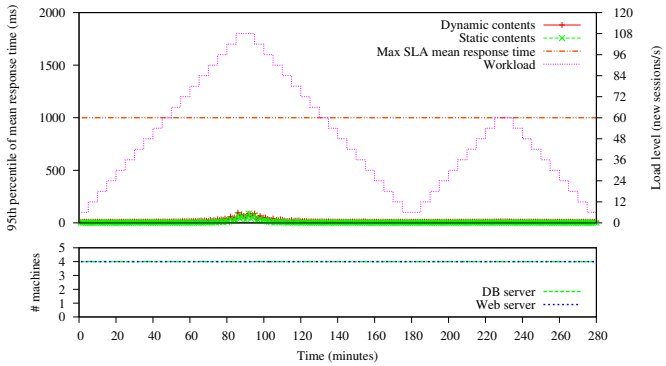


Figure 9: 95^{th} percentile of mean response time during Experiment 2.

Table 2: Summary of grid search to find good values for the important parameters of the proposed system.

t	k	**% requests missing SLA**	**Scale-down mistakes**	**Total operations**
30	4	3.228	12	38
30	8	2.002	3	22
60	4	2.413	4	22
60	8	2.034	2	20
120	4	3.227	2	18
120	8	3.312	0	15

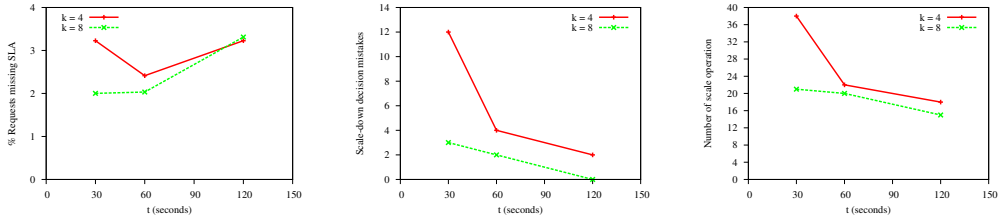
algorithm using a grid search. We then examined the results from the best configuration in more detail.

6.3.1. Parameter value identification

We used $t = 30, 60$, and 120 and $k = 4$ and 6 for the grid search. For each value of t and k, the percentage of requests missing SLA requirements, scale-down decision mistakes, and total number of scale operations (scale-up and scale-down) are shown in Table 2.

Figure 10 compares the percentage of requests missing SLA requirements, scale-down decision mistakes, and total number of scale (scale-up and scale-down) operations over different values of t and k.

We observe that a large number of requests exceed the required response time when we use small values ($t = 30$, $k = 4$) for both parameters or a large value ($t = 120$) for t. The parameter k is the number of consecutive



(a) Percentage of requests missing SLA.

(b) Scale-down mistakes.

(c) Total number of scale operations.

Figure 10: Grid search comparison for determining appropriate values of t and k for the system.

intervals of length t required to satisfy response time constraints before a scale-down operation is attempted. As k depends on t, using small values for t and k enables system to react quickly and make scale-down decisions that increase the number of scale-down mistakes. The system requires some time to recover from such mistakes, so we observe additional response time violations during the recovery. The large value of t increases the system's reaction time; this is why we observe a large number of requests exceeding the required response time with $t = 120$. We can also observe that as t increases, the number of scale down mistakes decreases, since scale down decisions are made less frequently. However, the slower response with high values of t also means that the system takes more time to respond to long traffic spikes and to release over-provisioned resources. Smaller values of t with larger values of k reduce the occurrence of scale down mistakes without negatively affecting the system's responsiveness to traffic spikes.

We selected the values $t = 60$ and $k = 8$ for further examination, as these values provide a good trade off between the percentage of requests missing the SLA, the number of scale-down decision mistakes, and the total number of operations. Figure 11 shows the 95^{th} percentile of the average response time during Experiment 3 using automatic bottleneck detection and adaptive resource allocation under this parameter regime. The bottom graph shows the adaptive addition and retraction of instances in each tier after a bottleneck or over-provisioning is detected during the experiment. Whenever the system detects a violation of the response time requirements, it uses the proposed reactive algorithm to identify the bottleneck tier then dynamically adds another virtual machine to the server farm for that bottleneck tier. We observe temporary violations of the required response time for short pe-

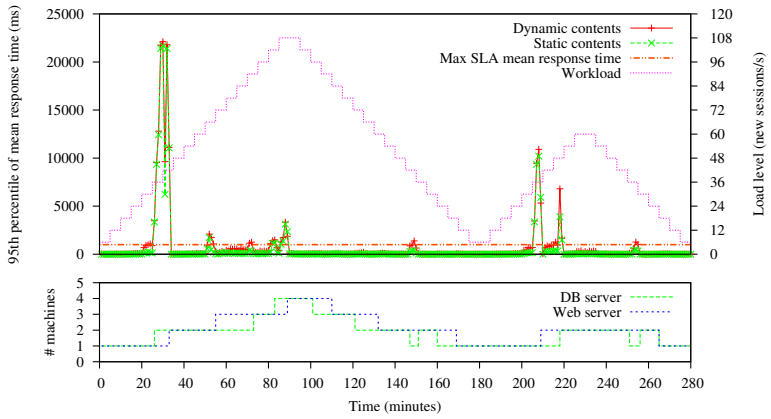


Figure 11: 95^{th} percentile of mean response time during Experiment 3 using $t = 1$ and $k = 8$ under proposed system.

riods of time due to the latency of virtual machine boot-up and the time required to observe the effects of previous scale operations. Whenever the system identifies over-provisioning of virtual machines for specific tiers using the predictive model, it scales down the specific tiers adaptively. In the beginning, the prediction model makes some mistakes; we can observe two incorrectly predicted scale-down decisions at load level 146 and load level 252. However, the reactive scale-up algorithm quickly brings the system back to a configuration that satisfies the response time requirements. Occasional mistakes such as these are expected due to noise, since the predictive approach is statistical. We could in principle reduce the occurrence of these mistakes by incorporating traffic pattern prediction as part of the decision model.

Figure 12 shows the system throughput during the experiment. We observe linear growth in the system throughput through the full range of load levels. The throughput increases and decreases as required with the load level.

Figure 13 shows the CPU utilization of all virtual machines during the experiment. Initially, the system is configured with one VM in each tier. The system adaptively adds and removes virtual machines to each tier over time. The differing steady-state levels of CPU utilization for the different VMs reflects the use of round-robin balancing across 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.

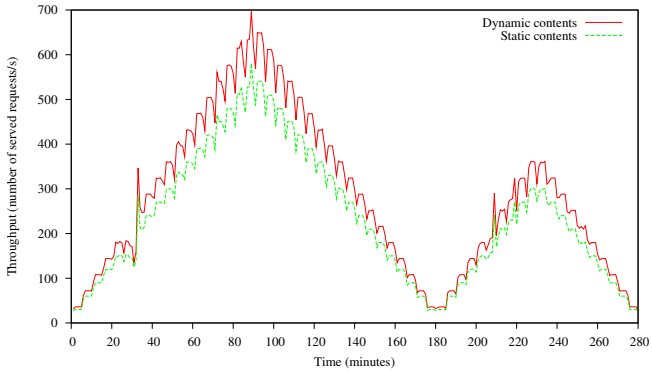


Figure 12: Throughput of the system during Experiment 3 using $t = 1$ and $k = 8$ under proposed system.

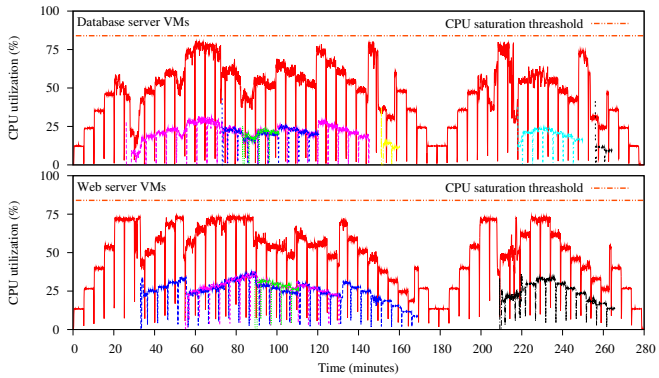


Figure 13: CPU utilization of all VMs during Experiment 3 using $t = 1$ and $k = 8$ under proposed system.

The experiments demonstrate first that insufficient static resource allocation policies lead to system failure, that maximal static resource allocation policies lead to overprovisioning of resources, and that our proposed adaptive resource allocation method is able to maintain a maximum response time SLA while utilizing minimal resources.

7. Conclusion

In this paper, we have proposed a methodology and described a prototype system for automatic identification and resolution of bottlenecks and automatic identification and resolution of overprovisioning in multi-tier ap-

plications hosted on a cloud. Our experimental results show that while we clearly cannot provide a SLA guaranteeing a specific response time with an undefined load level for a multi-tier Web application using static resource allocation, our adaptive resource provisioning method could enable us to offer such SLAs.

It is very difficult to identify a minimally resource intensive configuration of a multi-tier Web application that satisfies given response time requirements for a given workload, even using pre-deployment training and testing. However, our system is capable of identifying the minimum resources required using heuristics, a predictive model, and automatic adaptive resource provisioning. Cloud infrastructure providers can adopt our approach not only to offer their customers SLAs with response time guarantees but also to minimize the resources allocated to the customers' applications, reducing their costs.

We are currently extending our system to support n-tier clustered applications hosted on a cloud, and we are planning to extend our prediction model, which is currently only used to retract over-provisioned resources, to also perform bottleneck prediction in advance, in order to overcome the virtual machine boot-up latency problem. We are developing more sophisticated methods to classify URLs into static and dynamic content requests, rather than relying on filename extensions. Finally, we intend to incorporate the effects of heterogeneous physical machines on the prediction model and also address issues related to best utilization of physical machines for particular tiers.

Acknowledgments

This work was supported by graduate fellowships from the Higher Education Commission (HEC) of Pakistan and the Asian Institute of Technology to WI and by the Ministry of Science and Technology of Spain under contract TIN2007-60625. We thank Faisal Bukhari, Irshad Ali, and Kifayat Ullah for valuable discussions related to this work.

References

[1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility, Future Generation Computer Systems 25 (2009) 599 – 616.

[2] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, D. Zagorodnov, The EUCALYPTUS Open-source Cloud-computing System, in: CCA'08: Proceedings of the Cloud Computing and Its Applications Workshop, Chicago, IL, USA.

[3] P. Anedda, S. Leo, S. Manca, M. Gaggero, G. Zanetti, Suspending, migrating and resuming HPC virtual clusters, Future Generation Computer Systems 26 (2010) 1063 –72.

[4] X. Zhu, Z. Wang, S. Singhal, Utility-driven workload management using nested control design, in: ACC '06: American Control Conference, Minneapolis, Minnesota USA.

[5] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, K. Salem, Adaptive control of virtualized resources in utility computing environments, in: EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, ACM, New York, NY, USA, 2007, pp. 289–302.

[6] Z. Wang, X. Zhu, P. Padala, S. Singhal, Capacity and performance overhead in dynamic resource allocation to virtual containers, in: Integrated Network Management, 2007. IM '07. 10th IEEE International Symposium on Integrated Management, Dublin, Ireland, pp. 149–58.

[7] WMware, VMware Distributed Resource Scheduler (DRS), 2010. Available at `http://www.vmware.com/products/drs/`.

[8] G. Khanna, K. Beaty, G. Kar, A. Kochut, Application performance management in virtualized server environments, in: NOMS '06: Network Operations and Management Symposium, Vancouver, BC, pp. 373–81.

[9] I. Foster, T. Freeman, K. Keahy, D. Scheftner, B. Sotomayer, X. Zhang, Virtual clusters for grid communities, in: CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06), IEEE Computer Society, Washington, DC, USA, 2006, pp. 513–20.

[10] G. Czajkowski, M. Wegiel, L. Daynes, K. Palacz, M. Jordan, G. Skinner, C. Bryce, Resource management for clusters of virtual machines, in: CCGRID '05: Proceedings of the Fifth IEEE International Symposium

on Cluster Computing and the Grid (CCGrid'05) - Volume 1, IEEE Computer Society, Washington, DC, USA, 2005, pp. 382–9.

[11] A. Sundararaj, M. Sanghi, J. Lange, P. Dinda, Hardness of approximation and greedy algorithms for the adaptation problem in virtual environments, in: ICAC '06: 7th IEEE International Conference on Autonomic Computing and Communicatioins, Washington, DC, USA, pp. 291–2.

[12] Amazon Inc, Amazon Web Services auto scaling, 2009. Available at `http://aws.amazon.com/autoscaling/`.

[13] A. Azeez, Auto-scaling Web services on Amazon EC2, 2008. Available at `http://people.apache.org/~azeez/autoscaling-web-services-azeez.pdf`.

[14] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. Jordan, D. Patterson, Statistical machine learning makes automatic control practical for internet datacenters, in: HotCloud'09: Proceedings of the Workshop on Hot Topics in Cloud Computing.

[15] H. Liu, S. Wee, Web server farm in the cloud: Performance evaluation and dynamic architecture, in: CloudCom '09: Proceedings of the 1st International Conference on Cloud Computing, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 369–80.

[16] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, A. Tantawi, An analytical model for multi-tier internet services and its applications, in: SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, volume 33, ACM, 2005, pp. 291–302.

[17] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, T. Wood, Agile dynamic provisioning of multi-tier internet applications, ACM Transactions on Autonomous and Adaptive Systems 3 (2008) 1–39.

[18] R. Singh, U. Sharma, E. Cecchet, P. Shenoy, Autonomic mix-aware provisioning for non-stationary data center workloads, in: ICAC '10: Proceedings of the 7th IEEE International Conference on Autonomic Computing and Communication, IEEE Computer Society, Washington, DC, USA, 2010.

[19] W. Iqbal, M. Dailey, D. Carrera, SLA-driven adaptive resource management for web applications on a heterogeneous compute cloud, in: CloudCom '09: Proceedings of the 1st International Conference on Cloud Computing, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 243–53.

[20] W. Iqbal, M. N. Dailey, D. Carrera, P. Janecek, SLA-driven automatic bottleneck detection and resolution for read intensive multi-tier applications hosted on a cloud, in: GPC '10: Proceedings of the 5th International Conference on Advances in Grid and Pervasive Computing, pp. 37–46.

[21] xkoto, Gridscale, 2009. `http://www.xkoto.com/products/`.

[22] I. Sysoev, Nginx, 2002. Available at `http://nginx.net/`.

[23] L. Rodero-Merino, L. M. Vaquero, V. Gil, F. Galn, J. Fontn, R. S. Montero, I. M. Llorente, From infrastructure delivery to service management in clouds, Future Generation Computer Systems 26 (2010) 1226 –40.

[24] Google Code, Typica: A Java client library for a variety of Amazon Web Services, 2008. Available at `http://code.google.com/p/typica/`.

[25] OW2 Consortium, RUBiS: An auction site prototype, 1999. `http://rubis.ow2.org/`.

[26] D. Mosberger, T. Jin, httperf - a tool for measuring web server performance, in: In First Workshop on Internet Server Performance, ACM, 1998, pp. 59–67.

[27] J. Allspaw, The Art of Capacity Planning, O'Reilly Media, Inc., Sebastopol CA, USA, 2008.