MINIMALISTIC ADAPTIVE RESOURCE MANAGEMENT FOR MULTI-TIER APPLICATIONS HOSTED ON CLOUDS

by

Waheed Iqbal

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Examination Committee:	Dr. Matthew N. Dailey (Chairperson) Dr. David Carrera (External Expert) Dr. Raphaël Duboz (Member)
External Examiner:	Prof. Hai Jin Dean, School of Computer Science and Technology Huazhong University of Science and Technology Wuhan, China
Nationality:	Pakistani
Previous Degree:	Master of Engineering in Computer Science Asian Institute of Technology, Thailand
Scholarship Donor:	Higher Education Commission (HEC), Pakistan AIT Fellowship

Asian Institute of Technology
School of Engineering and Technology
Thailand
December 2012

Acknowledgments

I would like to thank many people and organizations that helped make this thesis possible.

First, I would like to record my gratitude and special thanks to Matthew N. Dailey and David Carrera for their supervision, support, and guidance throughout this research. Matthew always encouraged my ideas, provided necessary resources, and encouraged me to work on this dissertation. David Carrera's technical expertise and critical feedback enabled me to focus on the specific issues addressed in this dissertation.

Second, I gratefully acknowledge the Higher Education Commission (HEC) of Pakistan for providing financial support for my studies at AIT.

Third, I am feeling fortunate to be part of the AIT VGL lab where Matthew has made a good environment and provides necessary resources for his research students to stay focused and enjoy the work. I am grateful to all my lab fellows especially Faisal Bukhari, Irshad Ali, Waheed Noor, Kifayat Ullah, Jednipat Moonrinta, Zia Uddin, Sher Doudpota, Sergio Goncalves, Supawadee Chaivivatrakul, and Kan Ouivirach, for their support and help during this research work. I would also like to thank Paul Janecek and Olivier Nicole of the Computer Science and Information Management department at AIT for their support and help.

Fourth, I would like to thank the HEC, the IEEE Technical Committee on Scalable Computing, the SOSP 2011 organizers, and the AIT Computer Science and Information Management department for providing grants to present my research work in conferences.

Finally, I express my love and gratitude to my beloved parents, brother Naveed, sister Kiran, my wife Sana, and daughter Fatima for their endless love, prayers, and support throughout the duration of my studies. I am grateful for their understanding to allow me to spend most of my time on this dissertation for a long duration. I would like to happily admit that without their support and encouragement, I would not have been able to focus and finish this dissertation successfully.

Abstract

Cloud computing allows Web application owners to host their applications with low operational cost and enables them to scale applications to maintain performance based on traffic load and application resource requirements. Cloud providers intend to maximize revenue by serving large pools of users using minimal resource allocation while ensuring that specific service guarantees.

One of the typical architectures for cloud-hosted applications is the multi-tier Web application. From the user's point of view, response time is the most important attribute of a Web application. To offer specific response time guarantees for multi-tier Web applications using minimal resource allocation, cloud service providers need to automatically identify bottlenecks and scale only the specific resource tier where the bottleneck occurs. However, for multi-tier Web applications in particular, it is difficult to automatically identify bottlenecks and scale the appropriate resource tier. Application performance in the multi-tier architecture is complex in nature; depends critically on traffic usage patterns, which are highly dynamic and difficult to predict.

In this dissertation, we first propose and evaluate a system that reactively identifies bottlenecks in multi-tier Web applications and scales only the appropriate tier to ensure specific response time guarantees with minimal hardware profiling and application-centric knowledge. The proposed system is capable of minimizing resource allocation by retracting over-provisioned resources automatically using a predictive model based on the application's specific usage patterns. We demonstrate the feasibility of the approach in an experimental evaluation with a testbed EUCALYPTUS-based cloud and a synthetic workload. This kind of automatic bottleneck detection and resolution under dynamic resource management has the potential to enable cloud infrastructure providers to ensure response time guarantees to multi-tier Web applications while minimizing resource utilization.

Second, we improve the system by enhancing it to learn optimal resource allocation policies online. The proposed system first identifies workload patterns for a multi-tier Web application from access logs using unsupervised machine learning, reactively identifies bottlenecks for specific workload patterns, and learns optimal resource allocation policies by monitoring access logs in real time. The experimental results show that the proposed system can enable cloud providers to offer service time guarantees for multi-tier Web applications without any prior knowledge of the application's resource demands or workload patterns.

Third, we propose and evaluate a black-box method for capacity prediction that again identifies workload patterns for a multi-tier Web application and then, based on those patterns, builds a model capable of predicting the application's capacity for any specific workload pattern. The capacity of a multi-tier Web application varies substantially as the pattern of requests in the workload changes. In an experimental evaluation, we compare a baseline method that predicts capacity without a model of the application-specific workload patterns to several regression models using the proposed workload identification method. All of the models based on workload pattern identification outperform the baseline method. The best model, a Gaussian process regression model, gives only 6.42% error. The proposed method would be useful for proactively performing dynamic allocation of resources to multi-tier Web applications, meeting service level agreements at minimal cost.

Finally, we demonstrate the utility of adaptive resource allocation via an approach to scale gracefully in the presence of rapid increases in workload for multi-tier applications that have resource-intensive back ends responsible for continuous collection and analysis of real-time data from external services or applications. We demonstrate the feasibility of the approach in an experimental evaluation with a testbed cloud and a realistic simulation of a large scale external XMPP chat service. We conclude that cloud computing with adaptive resource allocation has the potential to increase the usability, stability, and performance of large-scale back-end mashup applications.

The techniques proposed in this dissertation would help cloud infrastructure providers offer response time guarantees to the owners of multi-tier applications using adaptive resource management while minimizing resource allocation. Application owners could also implement the proposed techniques on top of any cloud-based infrastructure to ensure necessary response time guarantees for their applications. Both cloud service providers and application owners benefit: the cloud service providers increase their revenue by accommodating more applications with minimal resources, and application owners obtain guarantees for the performance of their applications with low operational costs.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This dissertation explores the possibilities to satisfy response time guarantees for multi-tier applications hosted on a cloud using adaptive resource management with minimal hardware profiling and application-centric knowledge. In this chapter, we discuss the motivation of our work, explain the contributions, and provide an outline of the dissertation.

1.1 Motivation

The rapidly-increasing popularity of cloud computing is attracting many users to use cloud services. Cloud providers are offering cost savings and automation services to attract large pools of users. From the user's point of view, *response time* is the most important quality attribute of a Web service, and Web service providers aim to limit response time at minimal cost. Dynamic resource scaling in cloud computing could potentially enable specific response time guarantees, but current service-level agreements (SLAs) (Wustenhoff, 2002) offered by cloud infrastructure providers do not address response time.

One of the typical architectures for cloud-hosted applications is the multi-tier Web application consisting of at least a presentation tier, a business logic tier, and a data management tier running as separate processes. Multi-tier Web applications hosted on a specific fixed infrastructure can only service a limited number of requests concurrently before some bottleneck occurs. Once a bottleneck occurs, if the arrival rate does not decrease, the application will saturate, service time will grow dramatically, and eventually, requests will fail entirely. Hosted multi-tier Web applications, in which each tier is deployed to a separate set of fixed virtual machines, can only service a limited number of requests concurrently before some bottleneck occurs. Once a bottleneck occurs, the application will saturate, response time will grow dramatically, and the application will start failing to service requests.

For simple Web applications, it is possible to achieve low operational costs by first minimizing resource allocation then dynamically scaling allocated resources as needed to handle increased loads. It would also help to achieve better performance and low operational cost by retracting over-provisioned resources automatically depending on the usage patterns of the applications. However, for multi-tier applications, it is more difficult to automatically identify the exact location of a bottleneck and scale the appropriate resource tier accordingly (Urgaonkar, Pacifici, Shenoy, Spreitzer, & Tantawi, 2005). It is also difficult to identify the exact over-provisioned resources for multi-tier applications to minimize the operational cost with stable performance on different workload patterns. This is because multi-tier applications are complex and bottleneck patterns may be dependent on the specific pattern of workload at any given time.

It is possible to identify bottlenecks for a specific application by monitoring and profiling the low-level hardware resource utilization in each tier (Rao & Xu, 2010). However, these fine-grained techniques have to deal with extra monitoring overhead, virtualization complexity, and end-user security concerns involved in installing monitoring agents on rented virtual machines. Furthermore, low-level hardware profiling without monitoring response time metrics could not identify performance issues created by software misconfiguration. One example is whether the number of connections from each

server in the Web server tier to the database tier is appropriate.

A generic black box approach to identify bottleneck resources based on high-level information such as throughput and access logs without instrumenting any virtual machines would be much better for cloud providers to help them offer response time guarantees to the owners of applications using minimal resources. It would allow cloud infrastructure service providers to offer response time guarantees to a variety of different kinds of multi-tier applications without requiring application-centric knowledge.

In this dissertation, we advocate the use of coarse-grained monitoring techniques based on application access logs, throughput, and minimal hardware profiling techniques to help offering response time guarantees for multi-tier applications hosted on cloud. Our approach is to actively monitor response times for requests to a multi-tier Web application, gather CPU usage statistics, and identify the bottlenecks using heuristics. 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. We also improve system to learns optimal resource allocation policies by monitoring application's access logs in real time using unsupervised machine learning techniques.

1.2 Our Contributions

This dissertation makes the following contributions towards offering response time guarantees for multi-tier applications hosted on cloud.

1. We propose and assess a system for automatic detection and resolution of bottlenecks in a cloud-hosted multi-tier Web application in order to satisfy specific maximum response time requirements. The systems is able to identify and retract over-provisioned resources.
2. We introduce a new approach to identify workload patterns for multi-tier Web applications from access logs using unsupervised machine learning.
3. We devise a new methodology to reactively identifies bottlenecks for specific workload patterns.
4. We enable system to learns optimal resource allocation policies by monitoring application's access logs in real time.
5. We propose and evaluate a black-box method for capacity prediction that first identifies workload patterns for a multi-tier Web application and then builds a model capable of predicting the application's capacity for any specific workload pattern.
6. As one novel application of some of the techniques introduced in this dissertation, I present an approach to adaptively allocate cloud resources to scale gracefully in the presence of rapid increases in workload for applications that have resource-intensive back ends responsible for continuous collection and analysis of real-time data from external services or applications.

1.2.1 List of Publications

In this section, I provide a list of papers published as part of this dissertation and the list of papers currently under preparation for submission.

Published Work

- **Policy Learning for Adaptive Allocation of Cloud Resources to Multi-tier Web Applications**
  Waheed Iqbal, Matthew Dailey, David Carrera
  In Proceedings of 23rd ACM Symposium on Operating Systems Principles (SOSP), 2011
  Included in Chapter 3 and 4

- **Black-Box Approach to Capacity Identification for Multi-Tier Applications Hosted on Virtualized Platforms**
  Waheed Iqbal, Matthew Dailey, David Carrera
  In Proceedings of IEEE International Conference on Cloud and Service Computing (CSC), 2011
  Included in Chapter 5

- **Adaptive Resource Provisioning for Read Intensive Multi-tier Applications in the Cloud**
  Waheed Iqbal, Matthew Dailey, David Carrera, Paul Janecek
  In Future Generation Computer Systems (FGCS), 27(6): pp. 871–879, 2011
  Included in Chapter 2

- **SLA-Driven Dynamic Resource Management for Multi-tier Web Applications in a Cloud**
  Waheed Iqbal, Matthew N. Dailey, David Carrera
  In Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid), pp.832–837, 2010
  Included in Chapter 2

- **SLA-Driven Automatic Bottleneck Detection and Resolution for Read Intensive Multi-tier Applications Hosted on a Cloud**
  Waheed Iqbal, Matthew Dailey, David Carrera, Paul Janecek
  In Proceedings of the 5th International Conference on Advances in Grid and Pervasive Computing (GPC), pp.37–46, 2010
  Included in Chapter 2

- **Adaptive Resource Allocation for Back-end Mashup Applications on a Heterogeneous Private Cloud**
  Waheed Iqbal, Matthew Dailey, David Carrera, Imran Ali, Paul Janecek
  In Proceedings Proceedings of the 7th International Conference on Electrical Engineering/Electronics Computer Telecommunications and Information Technology (ECTI-CON), pp.317–321, 2010
  Included in Chapter 6

Manuscripts Currently in Preparation

– **Unsupervised Learning of Dynamic Resource Provisioning Policies for Multi-tier Web Applications in the Cloud**
Waheed Iqbal, Matthew Dailey, David Carrera

Manuscripts in Planning

– **Coarse-grained Versus Fine-grained Dynamic Resource Provisioning on Amazon EC2 Cloud.**
Waheed Iqbal, Matthew Dailey, David Carrera
To be submitted in IEEE Transactions on Parallel and Distributed Systems

1.3 Dissertation Outline

In this section, I provide an outline of the rest of the dissertation.

In Chapter 2, we propose a methodology and present 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. We also propose 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.

In Chapter 3, we describe a new model to identify workload patterns for a multi-tier Web application from access logs using unsupervised machine learning.

In Chapter 4 we present a methodology to reactively identifies bottlenecks for specific workload patterns, and learns optimal resource allocation policies by monitoring access logs in real time using machine learning techniques.

In Chapter 5, we propose and evaluate a black-box methodology for capacity prediction that first identifies workload patterns and then, based on those patterns, builds a model capable of predicting the application's capacity for any specific workload pattern.

In Chapter 6, we demonstrate an approach to allocate cloud resources dynamically to scale gracefully in the presence of rapid increase in workload for multi-tier applications that have resource-intensive back ends responsible for continuous collection and analysis of real-time data from external services or applications.

Finally in Chapter 7, we conclude this dissertation by summarizing our contribution and concluding remarks.

Chapter 2

Adaptive Resource Provisioning

2.1 Introduction

Cloud providers 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.

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 (Nurmi et al., 2009) 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 statistics 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 chapter, I 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 address some of these limitations in future chapters.

2.2 Related Work

There has been a great deal of research on dynamic resource allocation for physical and virtual machines and clusters of virtual machines (Anedda, Leo, Manca, Gaggero, & Zanetti, 2010). Zhu, Wang, and Singhal (2006) and Padala et al. (2007), proposed a two-level control loop to make resource allocation decisions within a single physical machine. This work does not address integrated management of a collection of physical machines. Wang, Zhu, Padala, and Singhal (2007) 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 (WMware, 2010) 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.

Khanna, Beaty, Kar, and Kochut (2006) proposed an approach similar to VMware DRS in which they 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. (2006) address the problem of deploying a cluster of virtual machines with given resource configurations across a set of physical machines. Czajkowski et al. (2005) 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 WMware (2010) and Khanna et al. (2006), 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 WMware (2010), Khanna et al. (2006), Foster et al. (2006), or Czajkowski et al. (2005).

Sundararaj, Sanghi, Lange, and Dinda (2006) studied network bandwidth allocation issues in the deployment of clusters of virtual machines. 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 (Amazon Inc, 2009) allows consumers to scale up or down according to criteria such as average CPU utilization across a group of compute instances. Azeez (2008) presents the design of an auto-scaling solution based on incoming traffic analysis for Axis2 Web services running on Amazon EC2. Bodik et al. (2009) present 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. Liu and Wee (2009) monitor 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. Urgaonkar et al. (2005) present 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. Urgaonkar, Shenoy, Chandra, Goyal, and Wood (2008) present 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.

Singh, Sharma, Cecchet, and Shenoy (2010) present 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 previous work (Iqbal, Dailey, & Carrera, 2009), we consider single-tier Web applications, use log-based monitoring to identify SLA violations, and use dynamic resource allocation to satisfy SLAs. Now here in this chapter, 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.

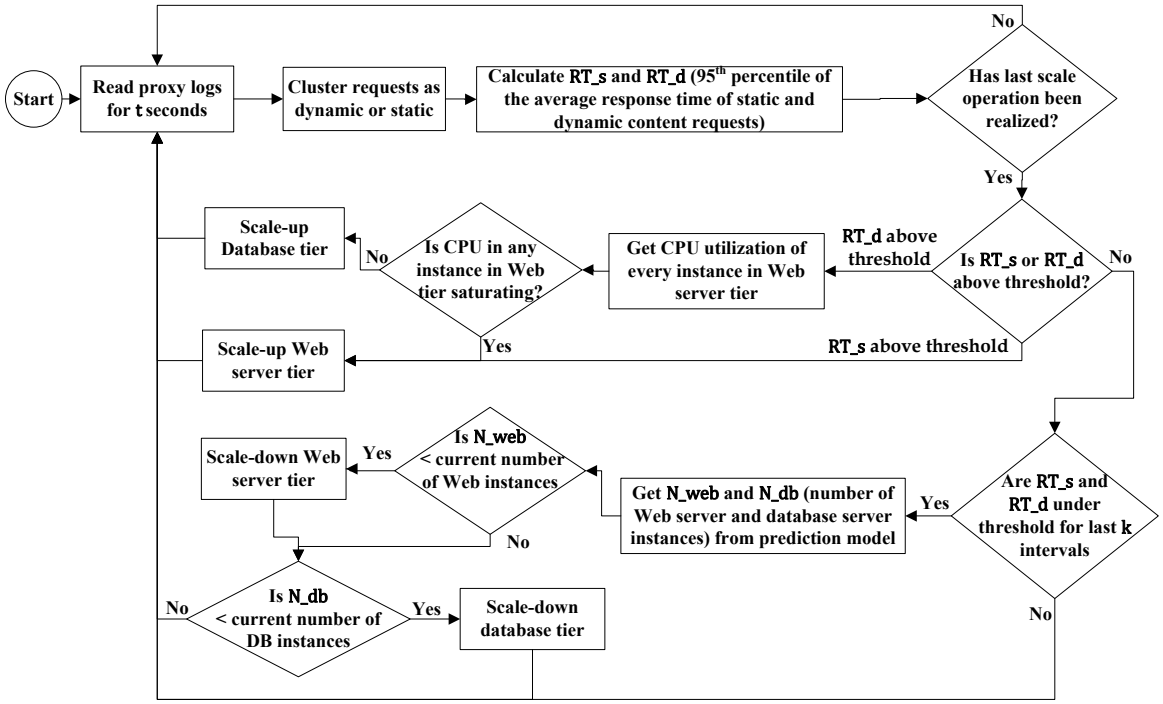


Figure 2.1: Flow diagram for prototype system that detects the bottleneck tier in a two-tier Web application hosted on a heterogeneous cloud and dynamically scales the tier to satisfy a SLA that defines response time requirements and ensures to release over-provisioned resources.

2.3 System Design and Implementation Details

2.3.1 Dynamic provisioning for multi-tier Web applications

Here I 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 2.1.

2.3.1.1 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 monitored separately for purposes of bottleneck de-

tection. 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 (xkoto, 2009) 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.

2.3.1.2 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:

$$n_t^{web} = a_0 + a_1(h_t^s + h_t^d) + a_2(h_t^s + h_t^d)^2 + \epsilon_t^{web} \quad \text{(Equation 2.1)}$$

$$n_t^{db} = b_0 + b_1 h_t^d + b_2(h_t^d)^2 + \epsilon_t^{db}, \quad \text{(Equation 2.2)}$$

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 Equation 2.1) 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 Equation 2.2) 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 2.1 to identify over-provisioned resources for the current workload and retract them from the current configuration.

2.3.2 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 (Sysoev,

2002) 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 (Rodero-Merino et al., 2010) components.

`VLBCoordinator` interacts with a EUCALYPTUS cloud using Typica (Google Code, 2008). Typica is a simple API written in Java to access a variety of Amazon Web services such as EC2, SimpleDB, and DevPay (Wikipedia, 2009). The core functions of `VLBCoordinator` are `instantiateVirtualMa` 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 (OW2 Consortium, 1999) 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 2.2 shows the deployment of our components along with the main interactions.

2.4 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.

2.4.1 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 2.3 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

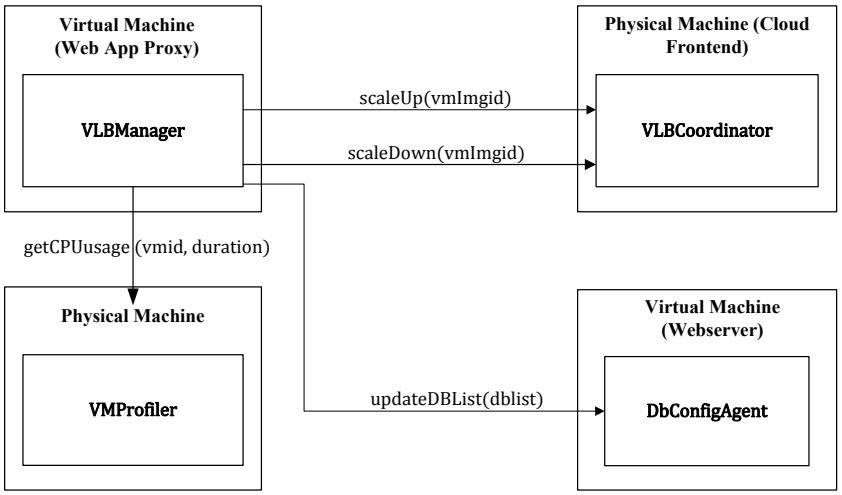


Figure 2.2: Component deployment diagram for system components including main interactions.

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.

2.4.2 Workload generation

We use `httperf` (Mosberger & Jin, 1998) 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 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 2.4 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.

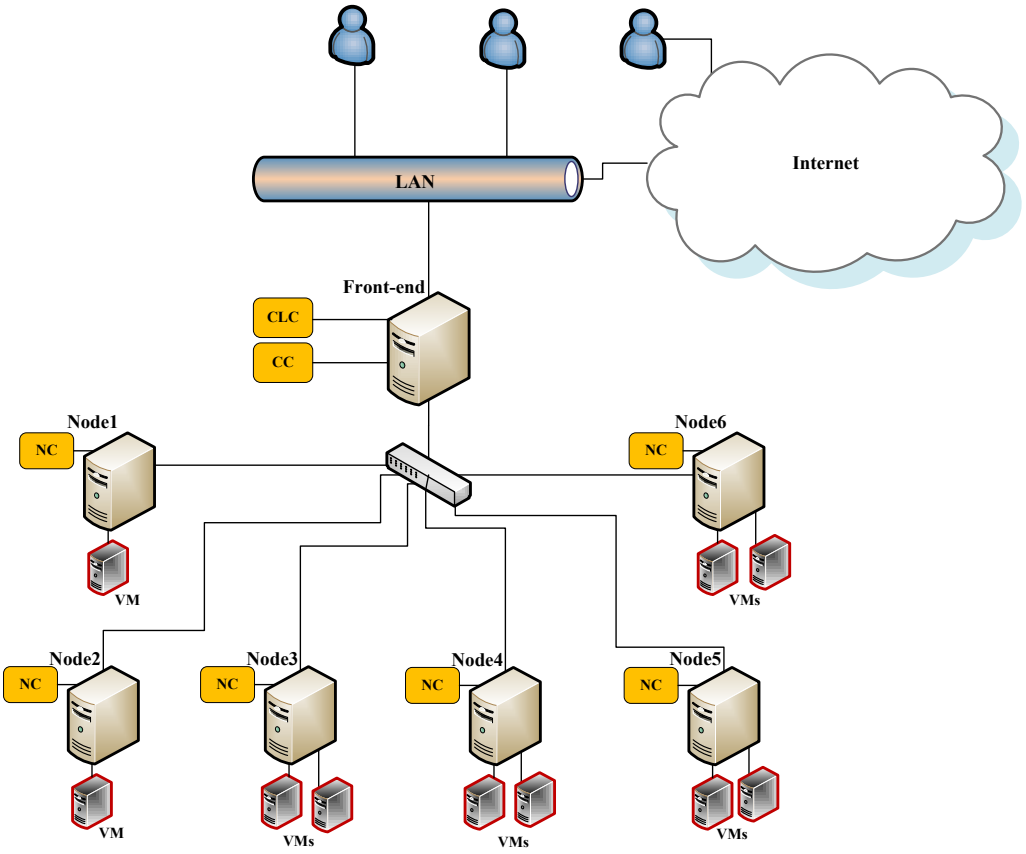


Figure 2.3: EUCALYPTUS-based testbed cloud using seven physical machines. We installed the CLC and CC on a front-end node attached to both our main LAN and the cloud's private network. We installed the NCs on six separate machines (Node1, Node2, Node3, Node4, Node5, and Node6) connected to the private 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.

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

2.5 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 2.1 summarizes the experiments, and details follow.

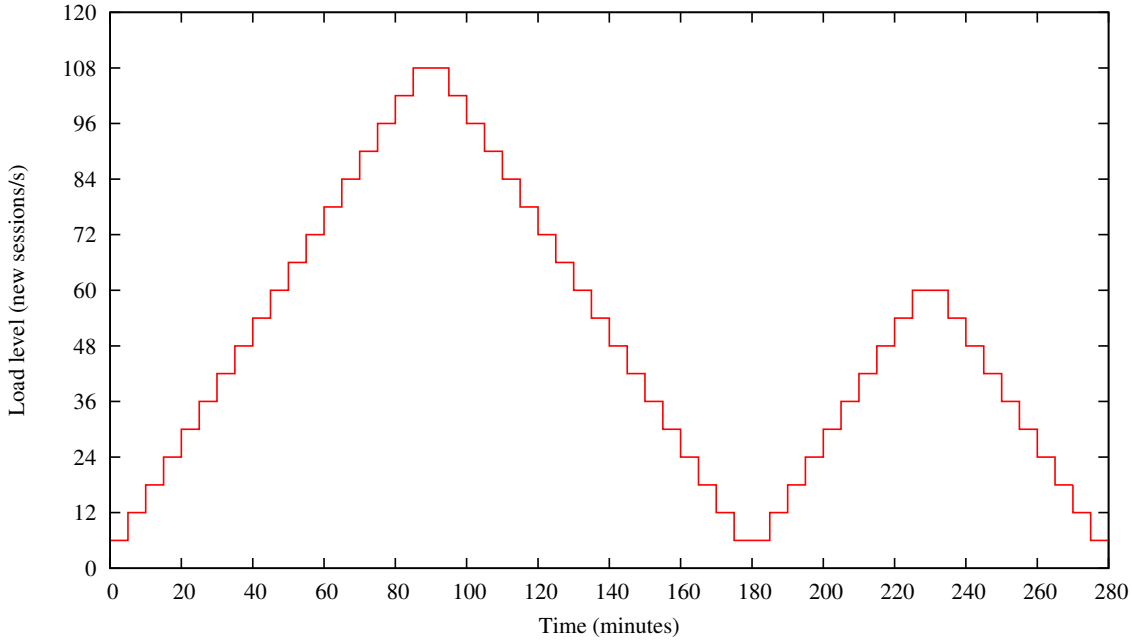


Figure 2.4: Workload generation profile for all experiments.

2.5.1 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.

2.5.2 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.

2.5.3 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-

Table 2.1: 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

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 2.1) 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 (Allspaw, 2008).

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.

2.6 Experimental Results

2.6.1 Experiment 1: Simple static allocation

This section describes the results we obtained in Experiment 1. Figure 2.5 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 2.6 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 2.7 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

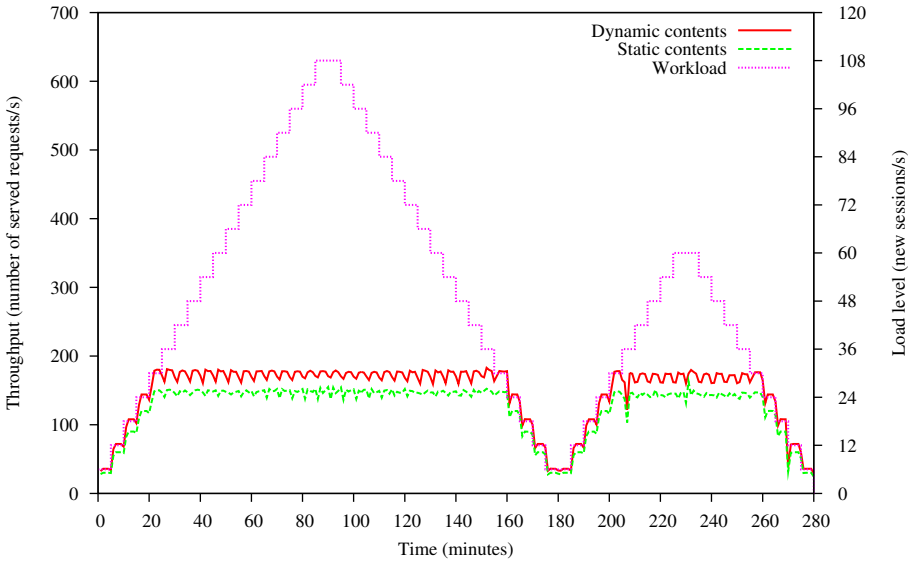


Figure 2.5: Throughput of the system during Experiment 1.

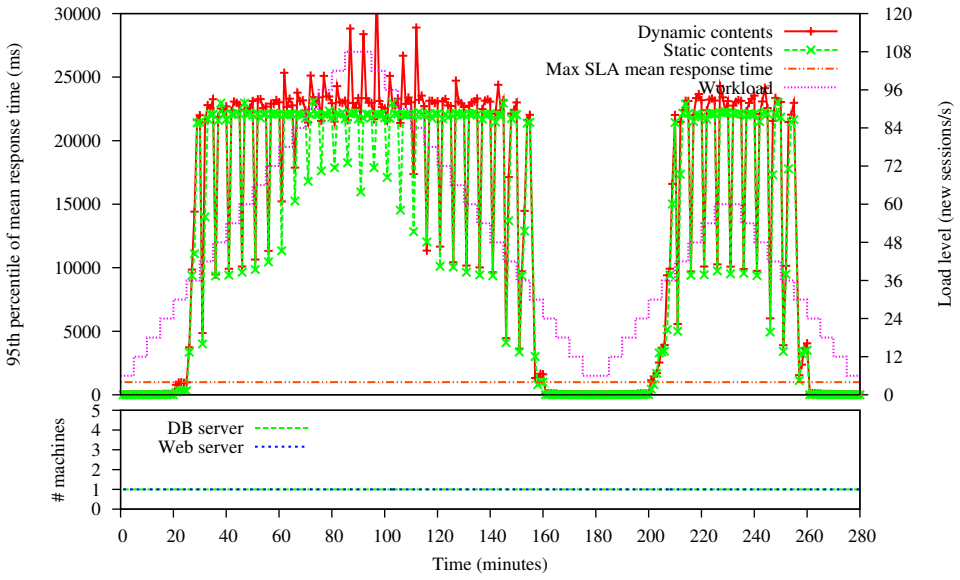


Figure 2.6: 95^{th} percentile of mean response time during Experiment 1.

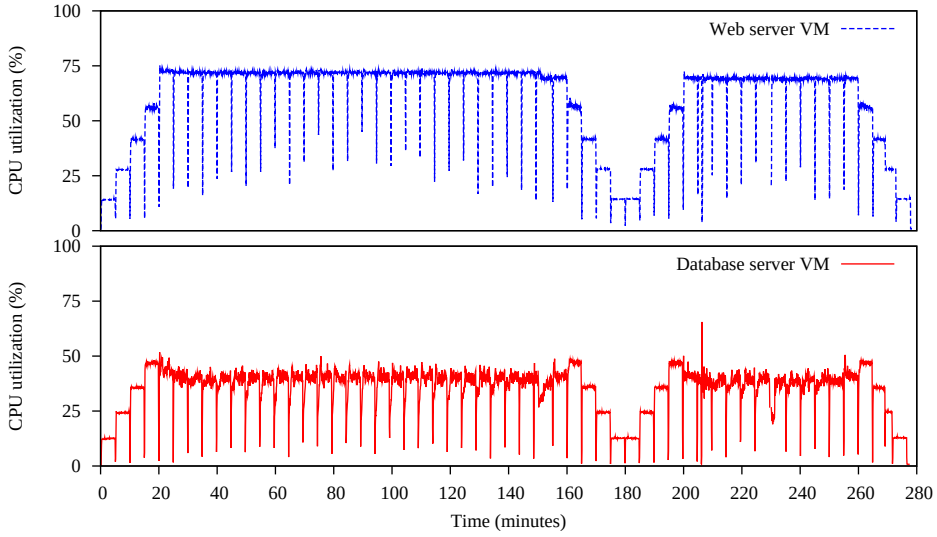


Figure 2.7: CPU utilization of virtual machines used during Experiment 1.

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

2.6.2 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 2.4.2. Figure 2.8 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 2.9 shows the 95^{th} percentile of average response times during Experiment 2. We do not observe any response time violations during the 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.

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

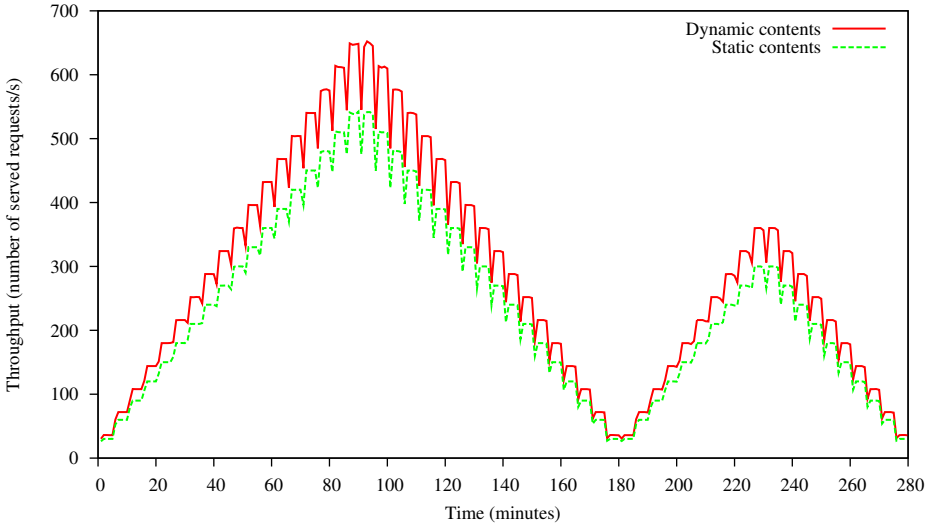


Figure 2.8: Throughput of the system during Experiment 2.

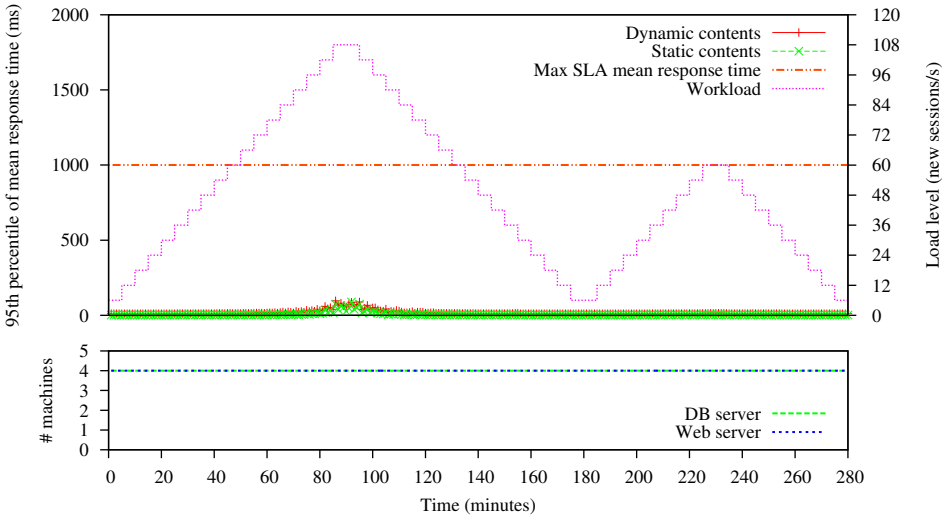
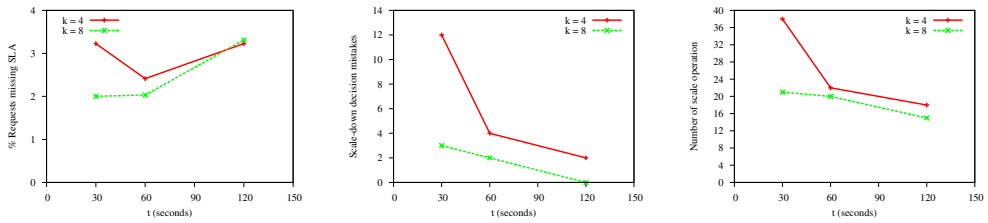


Figure 2.9: 95^{th} percentile of mean response time during Experiment 2.

Table 2.2: 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



(a) Percentage of requests missing SLA. (b) Scale-down mistakes. (c) Total number of scale operations.

Figure 2.10: Grid search comparison for determining appropriate values of t and k for the system.

2.6.3.1 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 2.2.

Figure 2.10 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 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

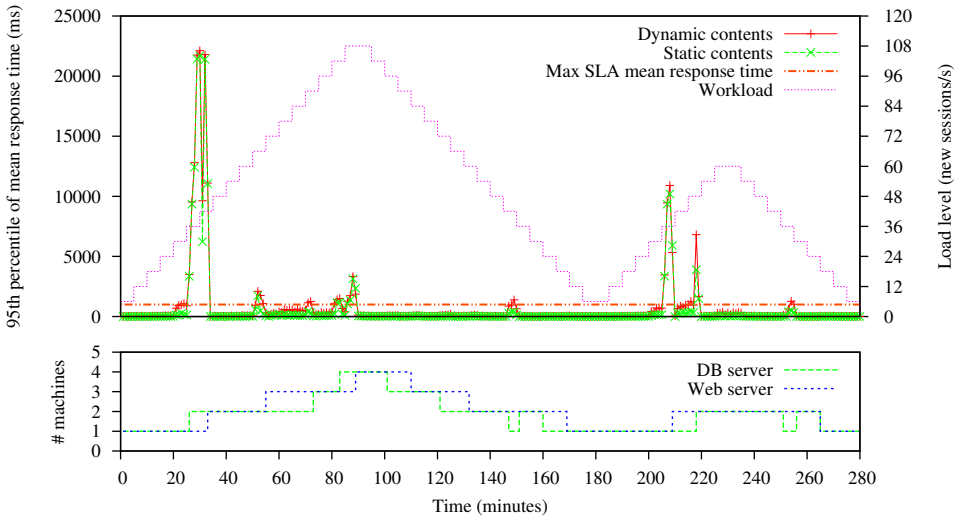


Figure 2.11: 95^{th} percentile of mean response time during Experiment 3 using $t = 60$ and $k = 8$ under proposed system.

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 2.11 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 periods 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 2.12 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 2.13 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

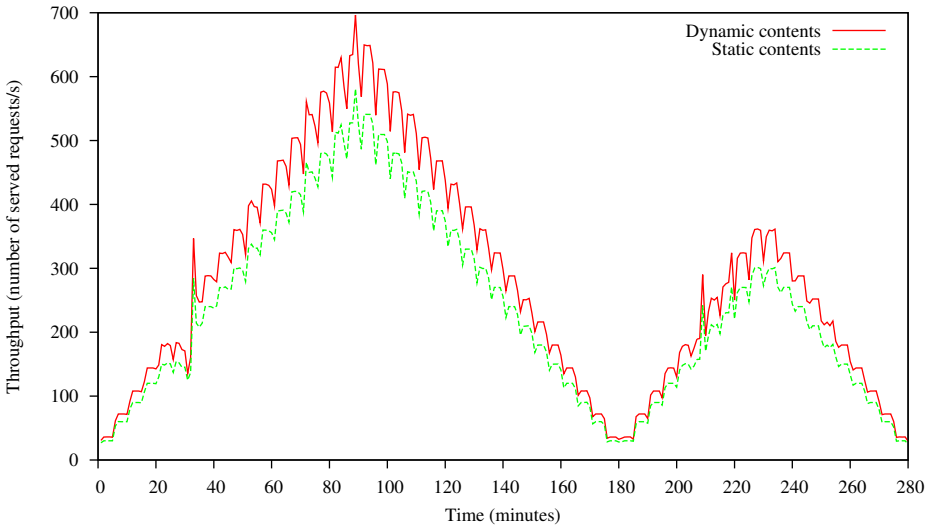


Figure 2.12: Throughput of the system during Experiment 3 using $t = 1$ and $k = 8$ under proposed system.

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.

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.

2.7 Discussion

In this chapter, I have presented our proposed methodology and described a prototype system for automatic identification and resolution of bottlenecks and automatic identification and resolution of overprovisioning in multi-tier applications 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.

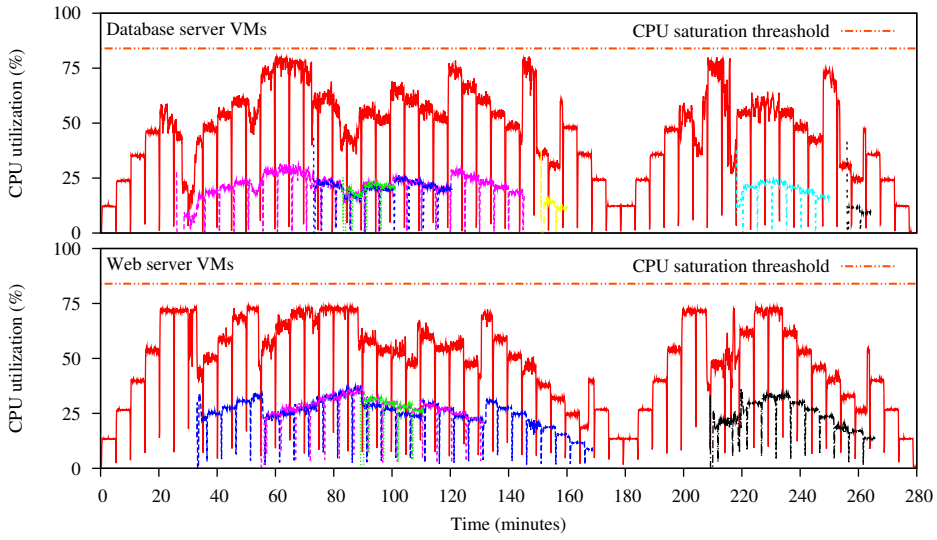


Figure 2.13: CPU utilization of all VMs during Experiment 3 using $t = 1$ and $k = 8$ under proposed system.

Chapter 3

Workload Pattern Modeling

In previous chapter, I presented a system that reactively identifies the bottlenecks in a multi-tier application and resolve them automatically without considering incoming traffic patterns. Now in this chapter, I provide a workload model and technique to identify the parameters of workload patterns for multi-tier Web applications using access logs and unsupervised machine learning.

3.1 Introduction

Web applications with large numbers of users have varying long term and short term workload patterns depending on special events and different time intervals of the day, week, or month. For example, the FIFA Web site (Football Association, 2011) observed sudden traffic spikes during the Soccer World Cup of 1998 (Martin & Tai, 2000). Search engines observed sudden spikes on the death of Farrah Fawcett and Michael Jackson (McGee, 2009). Online travel and booking sites exhibit different workload patterns at different times of the day and week (P. Nicolas, David, Ricard, Jordi, & Eduard, 2010).

The performance of an application could change due to a sudden increase in the number of requests or changes in the type of the requests required, possibily making large resource demands. Every Web application hosted on a specific infrastructure has limited capacity to server a maximum number of requests of a specific pattern. For example, at a certain time of day, a multi-tier Web application may receive only requests for static contents; then it may be capable of handling a huge number of requests for resources according to the same pattern without any bottlenecks. However, at another time of day, the application may receive requests primarily involving dynamic content generation, and the necessary database interaction would cause a bottleneck even with a small number of requests. Therefore, it is important to learn resource allocation policies that are dependent on the workload pattern in terms of volume and resource demands.

As a simple example of the effect of workload patterns on application bottlenecks, consider the following, in which we model five arbitrary but reasonable workloads for a specific application and profile the application's behavior. Each workload contains 10 different mixes of dynamic and static requests for the RUBiS benchmark auction application (OW2 Consortium, 1999). Figure 3.1 shows the service time saturation points for each of the five different workload patterns. We observe that the system performance varies from 1000 user sessions to 4500 user sessions depending on the specific workload pattern. These results indicate that the appropriate action to take to resolve a bottleneck might depend strongly on the current workload pattern.

Most of the work in dynamic resource provisioining (Villela, Pradhan, & Rubenstein, 2007; Bodik et al., 2009; Dejun, Pierre, & Chi, 2011) only identifies the change in the application workload volume to provision more resources to maintain application performance. However, only a few researchers have incorporated workload patterns in application's resource provisioning. Sharma et al. (2008) present a machine learning-based method to automatically characterize Web application resources by measuring CPU usage, number of requests, and network utilization. Bodik, Fox, Franklin, Jordan, and Patterson (2010) present a workload model for sudden increases in volume and demand for objects

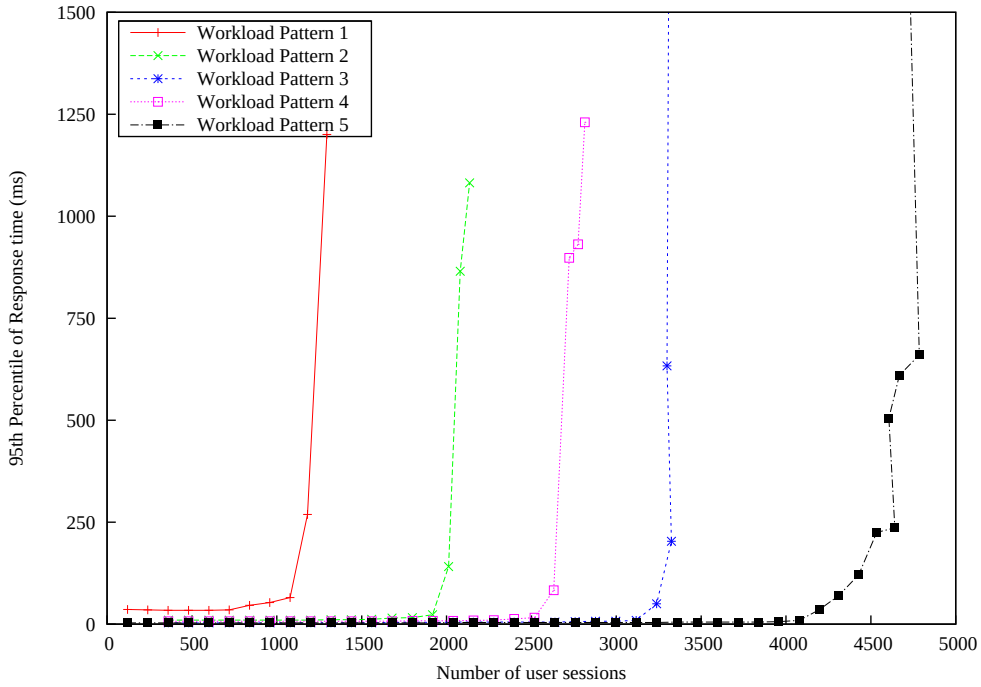


Figure 3.1: Response time saturation of a benchmark Web application under different workload patterns.

in stateful systems. Singh et al. (2010) present a technique to model dynamic workloads for multi-tier Web applications using k-means clustering (Hartigan & Wong, 1979) on a service time feature measured at each tier. 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. Our method identifies workload patterns using clustering, but we do not monitor each tier of the Web application. Instead, we treat the whole multi-tier application as a black box.

Our objective is to propose a simple workload model that identifies the behavior of the application resource demands and is easy to incorporate into resource provisioning algorithms. Our intension is to enable the system to incorporate the workload pattern into learning an algorithm for resource provisioning techniques, to resolve bottlenecks, and to scale the application gracefully.

In this chapter, I present our formal workload model and techniques to identify the parameters of workload patterns for multi-tier Web applications using access logs and unsupervised machine learning. The method automatically identifies groups of URIs with similar resource utilization characteristics from historical access log data. An experimental evaluation using real traffic from a Web application shows that the proposed method is able to characterize the Web resources appropriately based on the resource demands without profiling any hardware resources.

3.2 Workload Model

In this section, we describe our model for Web application workloads along with techniques for identifying the model's parameters via observations of the system's performance under different conditions over time. The model consists of two components:

– **URI space partitioning**: a partitioning of the application's URI space into requests with similar resource utilization characteristics.

– **Workload pattern**: a probabilistic model of request arrivals over the URI space.

We describe both of the components in the following subsections.

3.2.1 URI space partitioning

We assume that the URI space for a particular Web application can be partitioned into a set of k discrete *clusters* $\{c_1, \ldots, c_k\}$ with similar resource utilization characteristics. Within each cluster, we assume that the amount of any resource required (CPU time, network bandwidth, disk access, and so on) to service any particular request is random but follows an identical distribution for every distinct URI path in the URI space.

In the limit, in which k equals the number of distinct URI paths in the application's URI space, this is certainly a reasonable assumption. However, in practice, to make model identification tractable, we further assume that it is a reasonable approximation to fix k to a small number and thus map many URI paths to the same cluster.

To identify the URI space partitioning model from observations, we require that it is possible to collect a sufficiently large set of historical access logs. For most applications, these historical logs should be collected for a fairly long period of time such as days or weeks. We collect the logs and preprocess them to extract the URI path, document size, and service time for each request. We use a Gaussian mixture model clustering algorithm (Bishop, 2007) to group the requests into clusters based on document size and service time. We then construct a map from each URI path to the corresponding cluster ID. In cases where URI paths are mapped to multiple clusters, we use majority voting.

3.2.2 Workload patterns

Our workload pattern model is probabilistic. In each independent trial of the random experiment, we wait for the arrival of a single HTTP request from a remote client and observe the cluster $c \in \{c_1, \ldots, c_k\}$ that the request URI path falls into. Let C be the random variable describing which of the k clusters a request falls into. A probability distribution $P(C)$ over the clusters defines a *workload distribution* for the Web application.

In our model, a *workload pattern* is simply a specific workload distribution P over random variable C. We assume that over short periods of time, the workload distribution is stationary, so we write the workload pattern at a specific time as the vector

$$P(C) = (P(c_1), P(c_2), \ldots, P(c_k)) .$$

(For convenience, we abbreviate the event $C = c_i$ as simply c_i.)

Given the mapping from URI paths to cluster IDs, identifying the workload pattern for a specific interval of time is a simple matter of observing the frequency of arrival of requests for URI paths in each cluster.

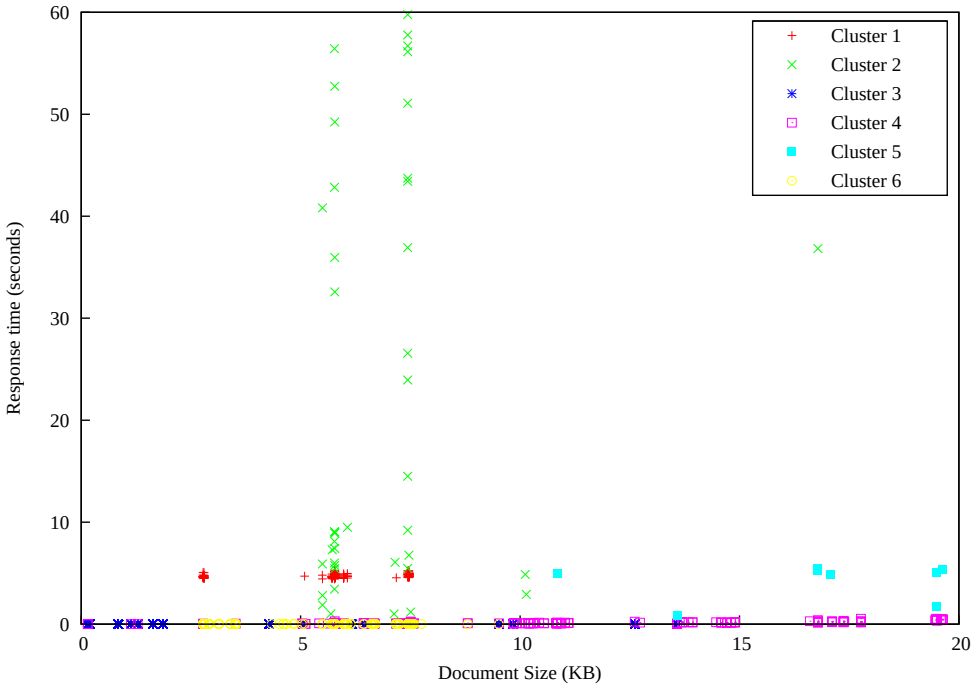


Figure 3.2: Clustering using document size and response time features on Lecture Buddy Web server traces.

3.3 URI space partitioning evaluation

To evaluate the effectiveness of the proposed URI space partitioning method we obtained one-month's worth of access logs from a real Web application named Lecture Buddy (Iqbal, 2011) hosted on Amazon EC2 using a "micro" instance. Figure 3.2 shows the resulting clusters of URIs based on document size and response time features. We used the Weka implementation of the Expectation Maximization (EM) algorithm for Gaussian mixture models (Bishop, 2007) to cluster the preprocessed log entries to identify the similar Web resources using clustering on document size and response time features. Weka's EM implementation automatically identifies the number of clusters (k) by maximizing the log-likelihood of future data.

After majority voting (to ensure that each URI map to only one cluster), we observed only three distinct clusters (Cluster 2, Cluster 3, and Cluster 5). Cluster 2 contains only static resources, Cluster 3 contains an equal number of large static resources (javascript libraries) and dynamic resources requiring heavy database interaction, and Cluster 5 contains all dynamic resources requiring minor database interaction. It shows that the proposed technique is able to cluster the Web application resources appropriately based on resource requirements. Table 3.1 shows the URI cluster mapping identified using the proposed technique along with the number of requests for each URI available in the dataset.

3.4 Discussion

In this chapter, I explained our simple and easy to use workload pattern model based on unsupervised learning techniques. Our proposed model only requires enough access log traces to group URIs based on similar resource requirements without any application-centric knowledge or profiling of any hardware resources. The proposed technique is applicable to any application capable of logging requested URI, response time, and response size features.

The proposed technique allows us to identify workload specific resource demands and use those in learning effective resource allocation policies, as well as to resolve bottlenecks and scale the application gracefully. We can also use this technique to model and predict the capacity of an application on varying workload patterns. In the future chapters, we use it to identify the current workload pattern during any given time interval and incorporate it into our resource allocation policy learning and Web application capacity identification techniques.

Table 3.1: URI cluster mapping using Lecture Buddy dataset.

URI	No. of Requests	Cluster ID
/images/bg.jpg	188	cluster2
/javascripts/jquery_ujs.js	249	cluster2
/stylesheets/blueprint/custom.css	272	cluster2
/stylesheets/blueprint/grid. css	271	cluster2
/javascripts/custom.js	258	cluster2
/stylesheets/blueprint/print.css	260	cluster2
/favicon.ico	212	cluster2
/stylesheets/blueprint/jquery.countdown.css	268	cluster2
/robots.txt	271	cluster2
/javascripts/jquery.countdown.min.js	251	cluster2
/images/lecturebuddy1.png	264	cluster2
/javascripts/application.js	253	cluster2
/stylesheets/blueprint/radio.png	129	cluster2
/stylesheets/scaffold.css	291	cluster2
/admin/	128	cluster3
/javascripts/jquery.min.js	193	cluster3
/sessions	26	cluster3
/javascripts/jquery-ui.min.js	150	cluster3
/pages/faq	22	cluster5
/student/	489	cluster5
/checkme/	70	cluster5
/	316	cluster5
/pages/about	17	cluster5
/summary/	6	cluster5

Chapter 4

Policy Learning for Adaptive Allocation of Cloud Resources

In previous chapters, I presented a system that idenfies and resolve bottlenecks in a multi- tier application automatically and provided a formal model to identify the workload patterns for multi-tier Web applications using access logs and unsupervised machine learning. Now in this chapter, I present a method for learning appropriate application- and workload-specific resource provisioning policies in real time using machine learning tehcniques.

4.1 Introduction

The rapidly-increasing popularity of cloud computing is attracting many users to use cloud services. Cloud providers are offering cost savings and automation services to attract large pools of users. Cloud providers maximize their profits by maintaining service levels using minimal infrastructure and maximal resource utilization.

One of the typical architectures for cloud-hosted applications is the multi-tier Web application consisting of at least a presentation tier, a business logic tier, and a data management tier running as separate processes. Multi-tier Web applications hosted on a specific fixed infrastructure can only service a limited number of requests concurrently before some bottleneck occurs. Once a bottleneck occurs, if the arrival rate does not decrease, the application will saturate, service time will grow dramatically, and eventually, requests will fail entirely.

It is important for Web applications to service all requests reliably and minimize the service time in order to be useful to their end users. Cloud providers can offer dynamic resource provisioning (Iqbal, Dailey, Carrera, & Janecek, 2011; Bodik et al., 2009) and auto scaling (Amazon Inc, 2009) to maintain maximum service time guarantees while minimizing resource utilization for a given workload. However, optimal proactive resource provisioning and scaling for a specific Web application require, at the least, a technique to automatically identify bottlenecks and scale the appropriate resource tier.

In principle, it is possible to identify bottlenecks for a specific application by monitoring and profiling the low-level hardware resource utilization in each tier under a variety of workloads. However, it would be impractical to apply these fine-grained techniques to all applications because of the extra monitoring overhead, virtualization complexity, and end-user security concerns involved in installing monitoring agents on rented virtual machines.

Since cloud providers do not and perhaps should not have insights into the applications they are hosting, but could readily access high-level information such as throughput and access logs without instrumenting any virtual machines, we advocate the use of coarse-grained monitoring techniques based on application access logs. This course-grained information coupled with methods for *learning* application- and workload-specific resource provisioning policies can enable cloud providers to automatically identify and resolve bottlenecks in the applications they are hosting.

Although learning bottleneck resolution policies from coarse-grained monitoring has potential benefits, one possible difficulty is that the optimal action to take when a bottleneck occurs might well

depend on the nature of the workload, which can change rapidly over time. As a simple example of the effect of workload patterns on application bottlenecks, we provided an example in Chapter 3, Figure 3.1. These results indicate that the appropriate action to take to resolve a bottleneck might depend strongly on the current workload pattern.

In this chapter, we present a method for learning appropriate application- and workload-specific resource provisioning policies. We develop a formal model for and techniques to identify the parameters of workload patterns for multi-tier Web applications using access logs and unsupervised machine learning. The method automatically identifies groups of URIs with similar resource utilization characteristics from historical access log data. We also design and empirically evaluate a method for satisfying a service-level agreement (SLA) that provides a maximum service time guarantee that works by reactively identifying bottlenecks for specific workload patterns and then learning resource allocation policies, all based on coarse-grained access log monitoring in real time. The policy learner initially uses a trial and error process to identify an appropriate bottleneck resolution policy in the context of a specific workload pattern then exploits that policy to reduce violations of the SLA while minimizing resource utilization. The approach does not require pre-deployment profiling or any insights about the application. We evaluate our proposed system on a EUCALYPTUS-based private cloud and the RUBiS benchmark Web application. We also compare our proposed approach with a baseline approach similar to Urgaonkar et al. (2008) that scales up all replicable tiers whenever bottleneck occurs.

There are a few limitations to this work. We only consider a two tier Web application installed on a private cloud. We assume that the workload pattern during each experiment remains constant. Additionally, we assume that sufficient bandwidth exists and enough historical access logs are available to identify URIs with similar resource utilization characteristics.

In the rest of this chapter, we present related work, our policy learning approach, and an experimental evaluation.

4.2 Related Work

There have been several efforts toward adaptive allocation of cloud resources to satisfy performance metrics. For example, Bodik et al. (2009) present a statistical machine learning approach to predict system performance for a single tier application and minimize the amount of resources required to maintain the performance of an application hosted on a cloud. Liu and Wee (2009) monitor the CPU and bandwidth usage of virtual machines hosted on an Amazon EC2 cloud, identify the resource requirements of applications, and dynamically switch 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.

There have been several efforts to use machine learning to manage application resources dynamically. For example, Bu, Rao, and Xu (2009) uses a reinforcement learning approach to identify the best server configurations settings (e.g., maximum number of clients and maximim number of threads), to maximize the performance of the system. Gerald, Nicholas K., Rajarshi, and Mohamed N. (2006) present a reinforcement learning approach to automatic resource allocation for single tier Web applications using offline initial policy learning.

Thus far, only a few researchers have addressed the problem of resource provisioning for multi-tier applications. Urgaonkar et al. (2005) present an analytical model using queuing networks to capture

the behavior of each tier. Rao and Xu (2010) present an on-line method for capacity identification (Allspaw, 2008) of multi-tier Web applications hosted on physical machines using hardware performance counters. Singh et al. (2010) present a technique to model dynamic workloads for multi-tier Web applications using k-means clustering on the service time feature that collects logs at each tier. 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. Our method identifies workload patterns using clustering, but we do not monitor each tier of the Web application. Instead, we treat the whole multi-tier application as a black box. Our proposed approach is able to learn resource allocation policies in real time.

Dejun et al. (2011) present a dynamic resource provisioning approach for multi-tier Web applications hosted on cloud and ensure to obtain homogeneous performance from every instance of tiers deployed on a heterogeneous environment. However authors do not consider different workload patterns.

The most recent work in this area (B. Nicolas, Thanasis G., & Karl, 2011) presents a cost-effective resource allocation approach to adaptively manage the cloud resources to satisfy response time and availability guarantees. The authors profile the response time of each tier of the application to identify bottlenecks but do not consider workload patterns. Our approach is more coarse-grained; it only profiles the application's load balancing proxy traces in the context of specific workload patterns to learn bottleneck resolution policies in real time.

To our knowledge, there is no system for dynamic resource allocation using coarse-grained monitoring able to learn optimal resource allocation policies for multi-tier Web applications in real time. We take the first step in this direction with a method to identify workload patterns and learn optimal resource allocation policies for a given workload.

4.3 Policy Learning

In this section, we develop an online, unsupervised method for learning a policy for adaptive resource allocation to a multi-tier Web application based on the trial-and-error approach of reinforcement learning. Reinforcement learners are agents attempting to maximize their long term reward by taking appropriate actions in an unknown environment. Our learning agent uses a simplistic method to find the policy (a mapping from application and workload state to resource allocation action) maximizing an objective function that encourages satisfying a response time SLA with minimal resources. We first define our learning agent then give the detailed policy learning algorithm.

4.3.1 Model

The **system state** $s_t = (U, P(C), \lambda, p)$ at time t contains the configuration of the Web application's tiers U, the current workload pattern $P(C)$, the current arrival rate λ, and the current 95^{th} percentile of the service time p. For an n-tier application, we define a configuration by the vector $U = (u_1, \cdots, u_n)$, where element u_i indicates the number of machines allocated to tier i.

The **action** a the agent can select at any point in time is a particular scale-up strategy. For our benchmark two-tier Web application, the possible scale-up strategies are to scale up the Web tier (a^w), to scale up the database tier (a^d), to scale up both tiers (a^b), or do nothing ($a^\emptyset$). The set of

possible actions the agent can perform is thus $A = (a^w, a^d, a^b, a^\emptyset)$.

A **policy** π is a mapping from system states s_t to corresponding actions a. We use the *value function* approach, in which we assume knowledge of the value $Q(s_t, a)$ of each possible action a in state s_t and simply select the action with the most value: $\pi(s_t) = \text{argmax}_a Q(s_t, a)$.

The **reward function** r encourages the learning agent when it is successful and discourages it when it is unsuccessful at maintaining the SLA with minimal allocation of resources. We use the immediate reward function $r(s_t)$, where $s_t = (U, P(C), \lambda, p)$ is a system state:

$$r(s_t) = \frac{1}{p + \alpha \sum_{i=1}^{n} u_i}. \qquad \text{(Equation 4.1)}$$

α specifies the relative weight of the response time and resource minimization objectives. In our experiments, we use $\alpha = 250$.

We model the **environment** the agent interacts with as a stochastic function $E(s_t, a)$ mapping current state s_t and action (scaling strategy) a to a new state. The agent must wait for a user defined interval to give enough time for the system to realize the effects of the action. We also allow the agent to instantaneously retract a previously executed action in the current environment with a function $E'(s_t, a)$. E' simply scales down the tier configuration by the number of machines added by action a, without affecting the workload state. Access to function E' allows the agent to explore different actions with respect to a specific configuration under possibly fluctuating workloads.

4.3.2 Policy learning algorithm

We use a simplified greedy version of the Q-learning approach (Watkins & Dayan, 1992) to build, through online observation, an estimate of the value $Q(s_t, a)$ of each action in each state. The learning agent begins with no knowledge and monitors, over each interval of time, for SLA violations. Whenever the agent detects a violation, when the amount of information is sufficient, it performs the optimal action according to the current estimated Q value function. When knowledge of the value function is insufficient, the agent attempts all possible actions using a simple exhaustive exploration algorithm. Algorithm 1 gives pseudocode for our exploration and exploitation (on-line learning and decision making) approaches.

4.4 Experimental Environment

Here we describe an experimental environment consisting of a EUCALYPTUS-based testbed cloud, a benchmark Web application, and a synthetic workload generator aimed at evaluating our proposed approach.

Input: Environment functions E and E', state space S, initial state s_0, actions A

Output: Estimated state-action value function $Q : S \times A \mapsto \mathbb{R}$.

For all $s \in S$, $a \in A$, $Q(s, a) \leftarrow 0$

$t \leftarrow 1$

$s_1 \leftarrow E(s_0, a^{\emptyset})$

while *true* **do**

  Extract p (95^{th} percentile service time) from s_t

  if $p > \tau$ *(SLA violation detected)* **then**

    if *for any* a, $Q(s_t, a) = 0$ **then**

      for *each* $a \in A \backslash a^{\emptyset}$ **do**

        $t \leftarrow t + 1$

        $s_t \leftarrow E(s_{t-1}, a)$

        $Q(s_{t-1}, a) \leftarrow r(s_t)$

        $s_t \leftarrow E'(s_t, a)$

      end

    end

    $a_t \leftarrow \text{argmax}_a \, Q(s_t, a)$

  else

    $a_t \leftarrow a^{\emptyset}$

  end

  $t \leftarrow t + 1$

  $s_t \leftarrow E(s_{t-1}, a_t)$

end

Algorithm 1: Policy learning algorithm.

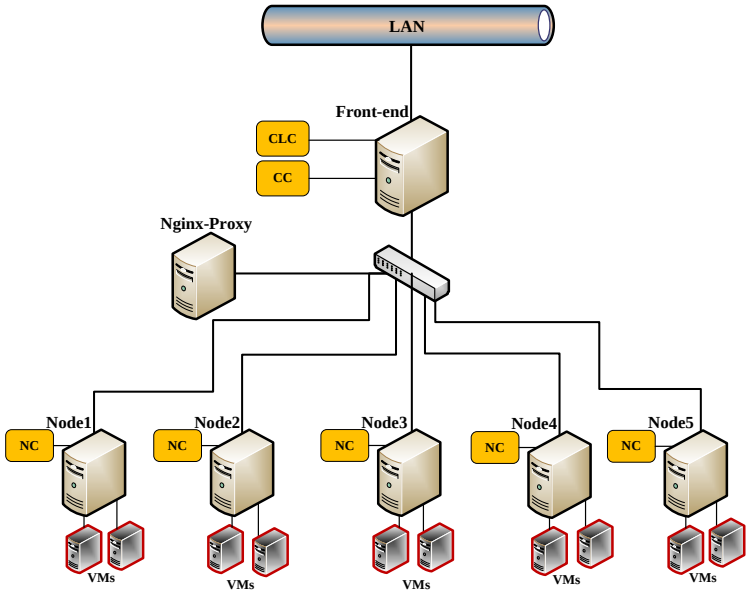


Figure 4.1: Experimental testbed for adaptive resource allocation policy learning.

4.4.1 Testbed Cloud

We built a small testbed cloud using seven physical machines (Front-end, Nginx-proxy, Node1, Node2, Node3, Node4, and Node5) and a gigabit Ethernet. Figure 4.1 shows the network design for our testbed cloud. Front-end is an Intel Pentium 4 machine with a 2.84 GHz CPU. Node3 is an Intel Core 2 Duo machine with a 2.6 GHz CPU. Node1, Node2, Node3, and Node4 are Intel Pentium Dual Core machines with 2.8 GHz CPUs. All machines have 2 GB RAM.

We used EUCALYPTUS to establish a cloud architecture comprised of one Cloud Controller (CLC), one Cluster Controller (CC), and five 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 separate machines (Node1, Node2, Node3, Node4, and Node5) connected to the private network. We dedicated one physical machine (Nginx-proxy) to act as a proxy server for the benchmark Web application.

4.4.2 Benchmark Web Application

RUBiS (OW2 Consortium, 1999) 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 for our experimental evaluation.

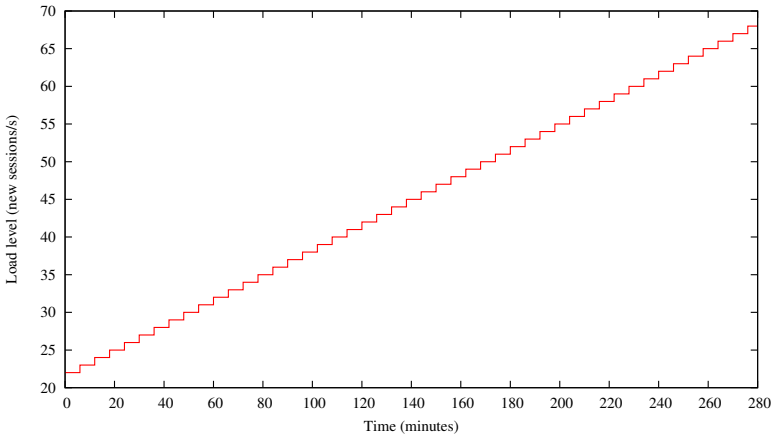


Figure 4.2: Step-up workload generation for Experiment 2.

4.4.3 Synthetic workload generation

We use `httperf` (Mosberger & Jin, 1998) to generate synthetic workload for our experiments. 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. We generate traffic in a step-up fashion, starting from a specific number of user sessions per second and increase the number of user sessions every 60 seconds. Figure 4.2 shows the number of user sessions per second as a function of time for all of our experiments.

In the experiments reported on in this paper, the workload pattern $P(C)$ is fixed during each experiment, but we use different workload patterns in experiment 1 (a relatively light workload pattern) and experiment 2 (a relatively heavy workload pattern). Details follow in the next section.

4.5 Experimental Evaluation

To evaluate the proposed method, we performed an experimental evaluation in which we first learned a URI partitioning model for the RUBiS benchmark Web application and then performed two experiments using different workload patterns. Both experiments executed in three phases in turn named **Exploration**, **Exploitation**, and **Baseline**, using the same workload pattern.

- In the **Exploration** phase, we initialize an empty policy and let the system learn in real time using the proposed policy learning algorithm.
- In the **Exploitation** phase, the agent uses the policy learned during exploration and resolves bottlenecks automatically using dynamic resource provisioning.
- In the **Baseline** phase, we use an alternative approach that reactively scales up both tiers whenever a SLA violation occurs.

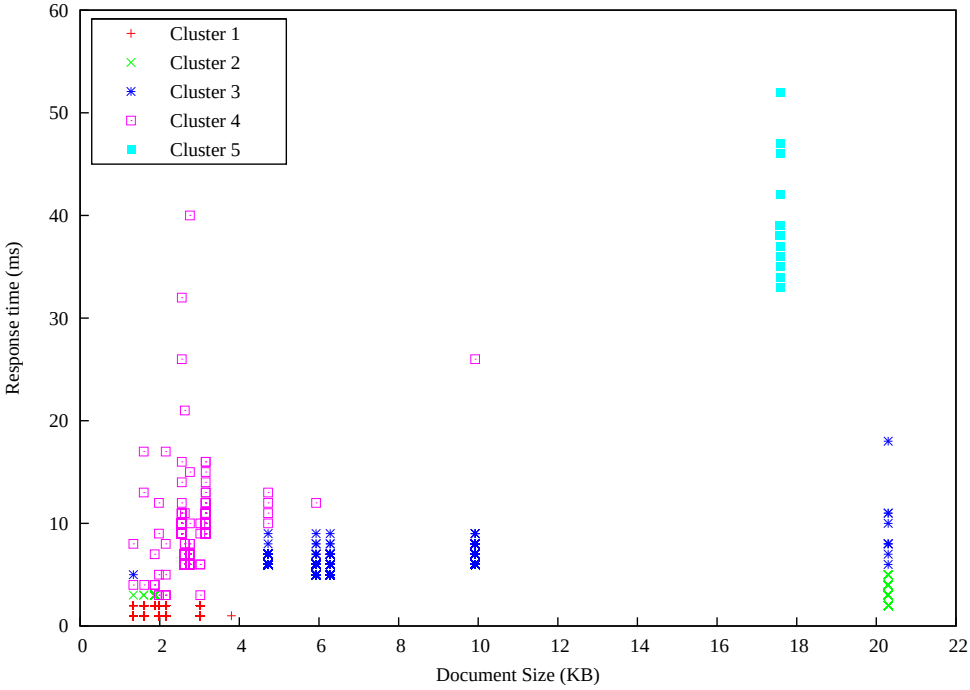


Figure 4.3: Clustering using document size and response time on RUBiS synthetic dataset.

4.5.1 URI partition model learning

We generated synthetic workload comprising different combinations of URIs corresponding to dynamic and static contents for RUBiS (sample benchmark Web application). Then we collected 19,200 log entries to identify the similar Web resources using the approach explained in Chapter 3, Section 3.2.1. The proposed approach identified five different clusters as shown in Figure 4.3. We obtained the majority cluster ID for each URI path in the log file and retained this mapping for the training stage.

4.5.2 Experiment 1 (light workload pattern)

In this experiment, we model each user session by six user requests following the workload distribution $P(C) = (0.334, 0.1667, 0.334, 0.1667, 0)$ according to the learned workload model. We call this workload a light workload pattern because it contains few requests and does not have any URIs belonging to cluster 5. The URIs belonging the cluster 5 are the most heavy users of resources and take a long time to process.

The synthetic workload for this experiment started from 90 user sessions per second and incremented every 60 seconds. For each of the three phases, we repeat the same workload generation process.

Figure 4.4 shows the 95^{th} percentile of average service time during the Exploration phase of Experiment 1. The bottom graph shows the exploration behavior and adaptive addition of machines to tiers. From load level 1 to load level 16, we observe a nearly constant service time, but after load level 16, the arrival rate exceeds the limits of the system's processing capability. The virtual machine hosting

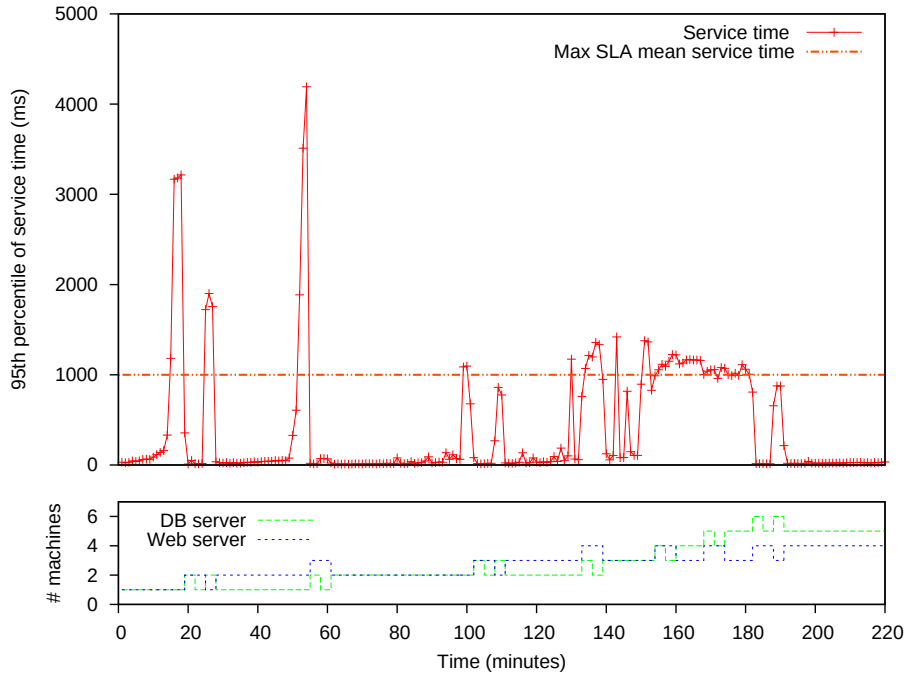


Figure 4.4: Experiment 1 (Exploration phase). The agent learns an initial policy by observing SLA violations then determining optimal bottleneck resolution actions. The top graph shows the 95^{th} percentile of the average service time, and the bottom graph shows the exploration behavior and adaptive addition of machines to the tiers.

the Web tier becomes a bottleneck, then requests begin to spend more time in the queue and request processing time increases. The learning agent starts exploring the possible set of actions. We see that both actions a^w and a^b suffice to satisfy the SLA whereas a^d fails to satisfy the SLA (causing the second spike in service time). Finally, the agent decides that a^w (only scaling the Web server tier) is the appropriate action to perform. The learner agent continues monitoring the SLA service time requirement, and when it detects new violations, it performs exploration to determine the best action and retains that action until the next violation occurs. At the end of the exploration phase of the experiment, the agent has learned a resource allocation policy to resolve bottlenecks automatically for this workload pattern.

Figure 4.5 shows the 95^{th} percentile of average service time during the Exploitation phase of Experiment 1. The bottom graph shows the exploitation behavior and adaptive addition of machines to the tiers. The agent simply exploits the policy learned in the Exploration phase. Whenever the agent detects service time requirement violations, it uses the policy to identify the action to resolve the bottleneck by adaptively provisioning resources to the selected tier(s). The number of requests in violation of the SLA is substantially decreased compared to the Exploration phase.

Figure 4.6 shows the 95^{th} percentile of average service time and adaptive scaling of both tiers on SLA violations during Baseline phase of Experiment 1. Although fewer SLA violations are observed than during the Exploitation phase, the application is overprovisioned for substantial periods of time.

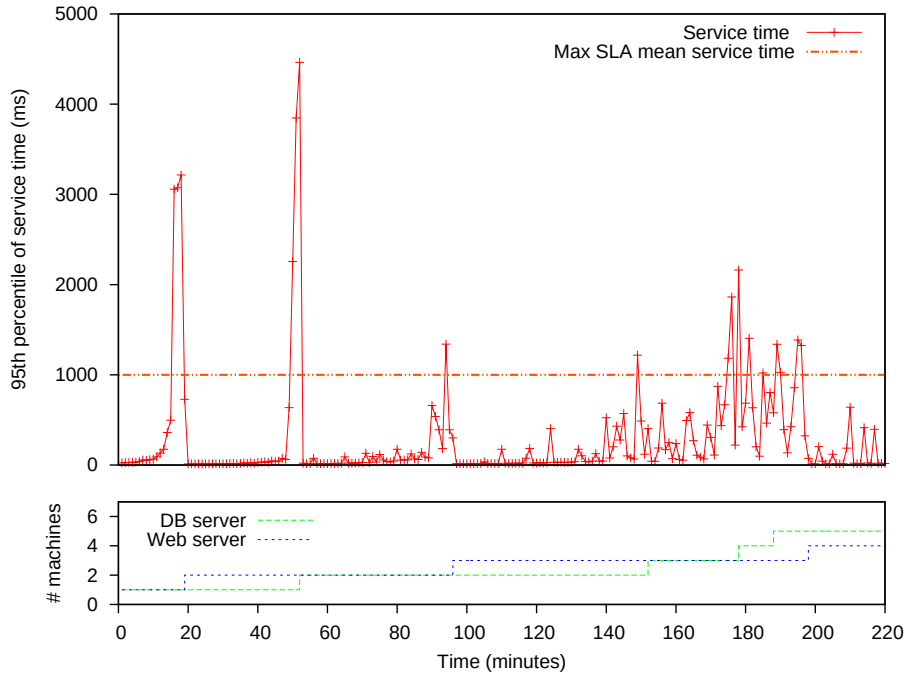


Figure 4.5: Experiment 1 (Exploitation phase). The agent exploits the policy learned during the Exploration phase. The top graph shows the 95^{th} percentile of the average service time, and the bottom graph shows the exploitation behavior and adaptive addition of machines to the tiers. The number of requests in violation of the SLA is substantially decreased by the policy learned in the Exploration phase.

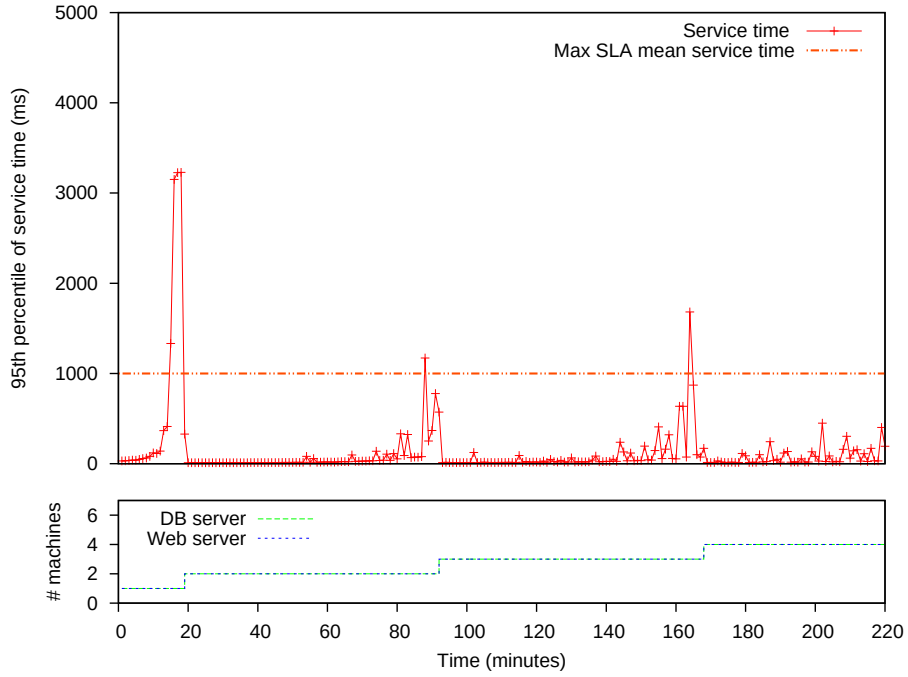


Figure 4.6: Experiment 1 (Baseline phase). The baseline autoscaling agent reactively scales both tiers on every service time violation. The top graph shows the 95^{th} percentile of the average service time, and the bottom graph shows the adaptive addition of machines to the tiers. Although fewer SLA violations are observed than during the Exploitation phase, the application is overprovisioned for substantial periods of time.

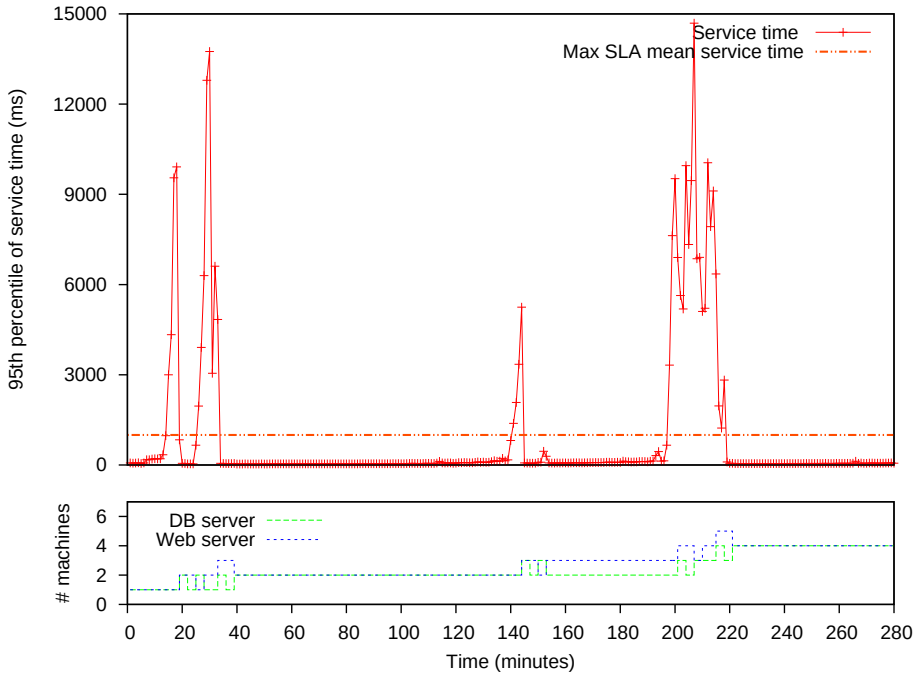


Figure 4.7: Experiment 2 (Exploration phase). The agent learns an initial policy by observing SLA violations then determining optimal bottleneck resolution actions. The top graph shows the 95^{th} percentile of the average service time, and the bottom graph shows the exploration behavior and adaptive addition of machines to the tiers.

4.5.3 Experiment 2 (heavy workload pattern)

In this experiment, we model each user session by 10 user requests following the workload distribution $P(C) = (0.3, 0.2, 0.2, 0, 0.3)$ according to the learned workload model. We call this workload a heavy workload pattern because it contains more requests, and 30% of the URIs belong to cluster 5, the most heavy users of resources that take a long time to process.

The synthetic workload for this experiment started from 22 user sessions per second and incremented every 60 seconds. For each of the three phases, we repeat the same workload generation process.

Figure 4.7 shows the 95^{th} percentile of the average service time during the Exploration phase of Experiment 2. The bottom graph shows the exploration behavior and adaptive addition of machines to tiers. From load level 1 to load level 20, we observe a nearly constant service time, but after load level 20, the arrival rate exceeds the limits of the system's processing capability. One or both 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. The learning agent starts exploring the set of possible actions. We see that the agent finally decides to perform action a^w (only scale the Web tier), as that action sufficed to control the response time during its selection. However, by the time the agent selects this action and starts monitoring again, the heavy workload has continued to increase, leading to another bottleneck, so the agent starts exploring again. The process continues, and at the end of the exploration phase of the experiment, the agent has learned a resource allocation policy to resolve bottlenecks automatically for the new workload pattern.

Figure 4.8 shows the 95^{th} percentile of the average service time during the Exploitation phase of

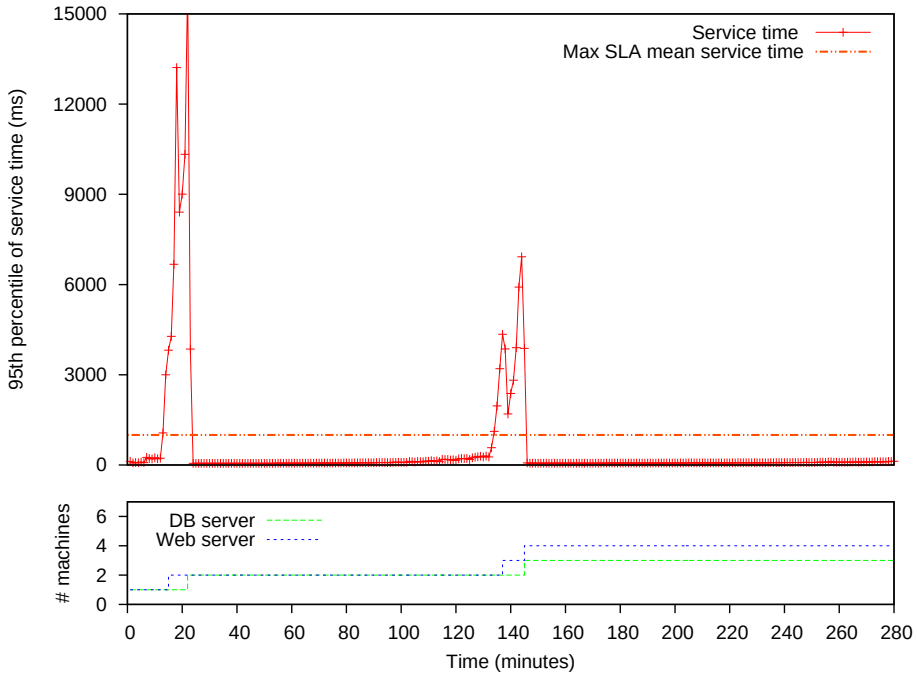


Figure 4.8: Experiment 2 (Exploitation phase). The agent exploits the policy learned during the Exploration phase. The top graph shows the 95^{th} percentile of the average service time, and the bottom graph shows the exploitation behavior and adaptive addition of machines to the tiers. As in Experiment 1, the number of requests in violation of the SLA is substantially decreased by the policy learned in the Exploration phase.

Experiment 2. The bottom graph shows the exploitation behavior and adaptive addition of machines to tiers. As in Experiment 1, here the agent simply exploits the policy learned in the Exploration phase. Whenever agent detects service time requirement violations, it uses the policy to identify the action to resolve the bottleneck by adaptively provisioning the resources to the selected tier(s). As in Experiment 1, compared to the Exploration phase, we observe that fewer requests violate the SLA.

Figure 4.9 shows the 95^{th} percentile of the average service time and adaptive scaling of both tiers on SLA violations during the Baseline phase of Experiment 2. As in Experiment 1, although fewer SLA violations are observed than during the Exploitation phase, the application is overprovisioned for substantial periods of time.

4.5.4 Summary of experimental results

To analyze the tradeoff between SLA violations and overprovisioning under our policy learning method and the baseline autoscaling method, we calculate two performance metrics: the total allocated CPU hours and the percentage of requests violating the SLA. Table 4.1 shows total allocated CPU hours and percentage of requests violating the SLA during the Exploitation and Baseline phases of both experiments. In both experiments, the system allocated fewer total CPU hours under the proposed policy learning approach. However, as is clear by comparing the detailed results already presented, the percentage of requests violating the SLA is higher for the proposed technique in both

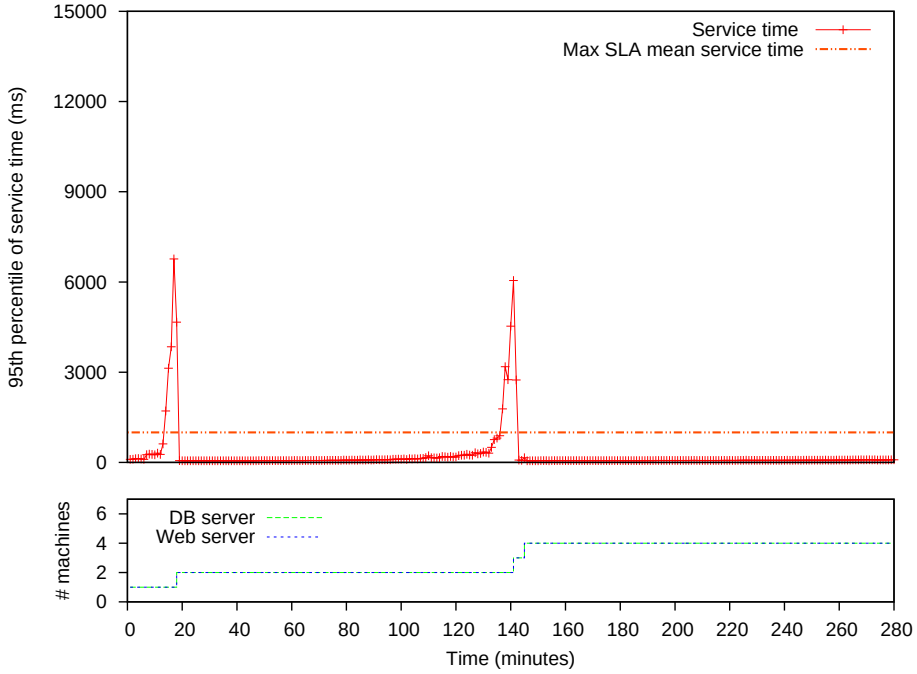


Figure 4.9: Experiment 2 (Baseline phase). The baseline autoscaling agent reactively scales both tiers on every service time violation. The top graph shows 95^{th} percentile of average service time, and the bottom graph shows the adaptive addition of machines to the tiers. As in Experiment 1, although fewer SLA violations are observed than during the Exploitation phase, the application is overprovisioned for substantial periods of time.

Table 4.1: Experimental results summary. Total allocated CPU hours and percentage of requests violating the SLA for proposed policy learning method and the baseline autoscaling method over Experiments 1 and 2.

		Total allocated CPU hours	**Percentage of requests violating SLA**
Experiment 1	Exploitation	17.7	1.03
	Baseline	19.8	0.35
Experiment 2	Exploitation	24.7	1.75
	Baseline	27.0	0.233

experiments.

Real world SLAs can incorporate an acceptable threshold of the maximum requests violating the SLA during a certain duration. However, the most important feature of the system is to bring the application performance back to the normal automatically and quickly. Therefore, we focused to enable the system to maintain the performance and bring the system to the normal state automatically using adaptive resource allocation.

The results show clearly that the Baseline approach moderately overprovisions resources but provides better performance in terms of service time. This is due to the fact that in our experiments, the workload is always increasing, so proactive overprovisioning allows the system to serve requests efficiently for a longer period of time before bottlenecks occur. It is of course always possible to overprovision resources to reduce and/or eliminate SLA violations, but in practice, since cloud providers must deal with multiple applications, overprovisioning would prevent the cloud provider from accommodating more consumers to maximize resource utilization.

Cloud providers need techniques that minimize SLA violations without overprovisioning resources. The evaluation shows that the proposed policy learning approach could help cloud providers host multi-tier Web applications with SLAs providing specific service time guarantees while minimizing resource utilization.

4.6 Discussion

In this chapter, we have presented a method for unsupervised, on-line autoscaling policy learning multi-tier Web applications under different workload patterns. The proposed approach does not require any prior knowledge of the application's resource utilization and minimizes the overhead needed to monitor, detect, and resolve bottlenecks. Our experimental evaluation shows the strength of the approach in resolving bottlenecks in multi-tier Web applications while only provisioning the necessary resources, meeting service level agreements at minimal cost.

The work described in this chapter is preliminary and has several limitations. One possible extension of this work is to incorporate long-term exploration as necessary to revise existing policies under

dynamically changing workload distributions and introduce policy learning for scale-down actions.

Chapter 5

Black-Box Approach to Capacity Identification

In previous chapters, I presented a system that idenfies and resolve bottlenecks in a multi- tier application automatically, provided a formal model to identify the workload patterns for multi-tier Web applications using access logs, and presented a method for learning appropriate application- and workload-specific resource provisioning policies in real time using machine learning tehcniques. Now in this chapter, I present a black-box approach to identify the system capacity of multi-tier Web applications hosted on specific infrastructure using machine learning practices.

5.1 Introduction

Multi-tier Web applications hosted on a specific infrastructure have limited capacity to service simultaneous requests. Once the arrival rate goes beyond a system's capacity, the application will saturate, and response time will grow dramatically.

A Web application's capacity varies according to different workload patterns (Banga & Druschel, 1997a, 1999). To understand the effect of workload pattern on application capacity, we provided and example in Chapter 3, Figure 3.1. In which we model five arbitrary but reasonable workloads for a specific application and profile the system's behavior. We observe that the system capacity varies from 1000 user sessions to 4500 user sessions depending on the specific workload pattern.

It is important for Web applications to minimize response time to provide better usability to end users. Cloud providers can offer dynamic resource provisioning (Iqbal, Dailey, Carrera, & Janecek, 2011) and auto scaling (Amazon Inc, 2009) services to maintain maximum response time guarantees while minimizing resource utilization for a given workload. However, optimal proactive resource provisioning and scaling for a specific Web application require, at the least, a profile of the application's current workload and a model of the application's capacity under various resource configurations.

In principle, it is possible for a cloud provider to build a capacity model for a specific application by monitoring and profiling the low-level hardware resource utilization in each tier under a variety of workloads. However, it would be impractical to apply these white-box techniques to all applications because of the extra monitoring overhead, virtualization complexity, and end-user security concerns involved in installing monitoring agents on the rented virtual machines. Instead, we advocate black-box techniques that only use application access logs to identify system capacity.

The capacity of a multi-tier Web application depends on many factors, including the details of the current workload pattern, the application's hardware resource utilization, and server configuration parameters such as the number of worker threads in the Web tier and the number of simultaneous connections allowed to the database tier. Therefore, cloud providers need to adopt techniques that contain minimum overhead yet are able to accurately predict capacity in order to offer valuable services such as admission control and dynamic scaling.

Cloud providers do not have insights into the applications they are hosting, but high-level information such as application throughput and access logs could be readily available without instrumenting vir-

tual machines. Cloud infrastructure service providers can use such information and adopt a black-box approach to identify the capacity of the applications they are hosting. This would allow cloud infrastructure providers to offer auto scaling services using dynamic provisioning for multi-tier applications without profiling hardware resource utilization and without scrutinizing the hosted applications.

One potential difficulty in black-box Web application capacity prediction is that capacity might vary depending on the workload pattern, which can change rapidly over time. To identify the capacity for Web applications, we propose to build a model that explicitly represents the relationship between the workload pattern and the number of requests that a system can service without saturating.

In this chapter, we present a black-box approach to identify the system capacity of multi-tier Web applications hosted on virtualized platforms using machine learning practices. Initially, we identify groups of URIs with similar resource utilization characteristics from historical access log data. Then we learn capacity predictors online by identifying the workload pattern and the maximum number of requests the system served without saturating. Our proposed approach does not require predeployment profiling or any insights about the application. We evaluate our proposed system on the RUBiS benchmark Web application.

There are a few limitations to this preliminary work. We only consider a two tier Web application installed on a homogeneous virtual infrastructure. Additionally, we assume that sufficient bandwidth exists and enough historical access logs are available to identify URIs with similar resource utilization characteristics.

In the rest of this chapter, we present related work, Web application capacity model, our approach, and an experimental evaluation.

5.2 Related Work

There have been several efforts in system capacity planning (Allspaw, 2008) and identification. For example, Cherkasova, Tang, and Singhal (2004) present a framework for identifying the required resources for streaming services to process a given workload. Banga and Druschel (1997b) provide a framework to generate synthetic workload for Web servers and performs stress testing to identify the Web server capacity on bursty traffic. Kant and Won (1999) derive mathematical formulae to identify the demand for hardware resources such as memory, processor data bus, and the network adapter on a given workloads for a Web server.

In the area of multi-tier Web application capacity identification, recent work (Rao & Xu, 2010) presents an on-line method for capacity identification of applications hosted on physical machines using hardware performance counters. The most recent work in this area (Singh, Sharma, Cecchet, & Shenoy, 2010) presents a technique to model dynamic workloads for multi-tier Web applications using k-means clustering on the service time feature by collecting logs at each tier. 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. Our method identifies workload patterns using clustering, but we do not monitor each tier of the Web application. Instead, we treat the whole multi-tier application as a black box. Our proposed approach is able to learn an online predictor to identify the system capacity on any given workload pattern.

There has been some effort made to learn application performance models online for resource allo-

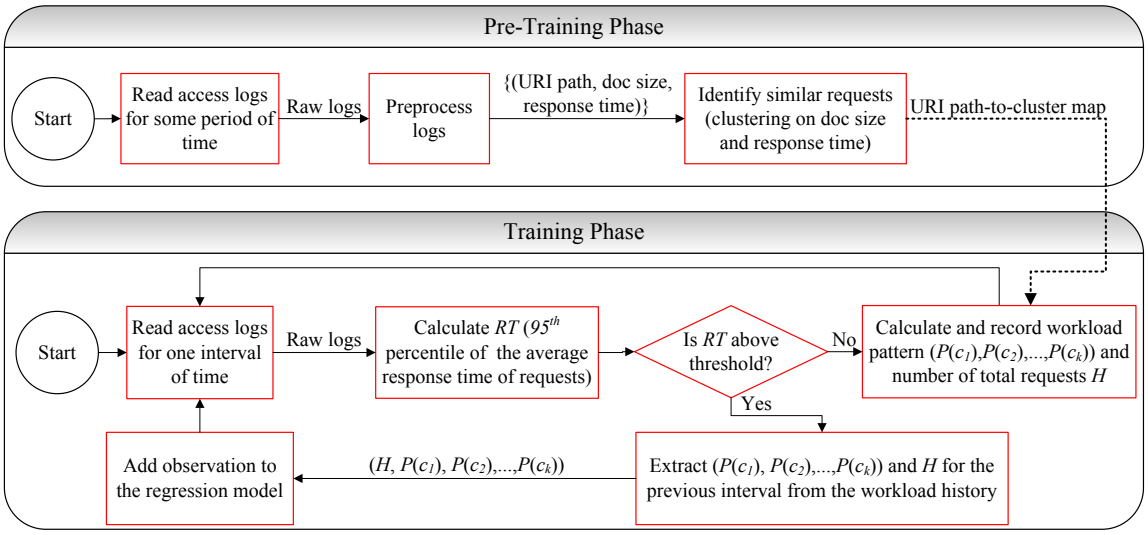


Figure 5.1: Flow diagram for capacity identification learning procedure. The method is composed of two main parts: the *pre-training* phase in which we identify URI clusters, and the *training* phase in which we perform online estimation of the statistical model.

cation. Bodik et al. (2009) present an approach to learn a performance model using local regression (a nonlinear regression technique) (Larry, 2006) for Web applications hosted on cloud. The model is used to provision resources needed to satisfy the SLA requirements. The focus of the proposed approach is to learn an accurate application performance model with minimal SLA violations. However, the proposed approach does not consider the effect of different workload patterns. In this preliminary work, we propose a method to identify workload patterns and learn an online predictor (performance model) to identify the system's capacity for the given workload.

5.3 Statistical Model for Web Application Capacity

Here we describe how we model Web application capacity. The components of the model are

- a partitioning of the application's URI space into requests with similar resource utilization characteristics (explained in Chapter 3, Section 3.2.1);
- a probabilistic model for workload patterns over the URI space (explained in Chapter 3, Section 3.2.2);
- a general statistical model for Web application capacity based on those workload patterns; and
- an online method for acquiring training data for the statistical model.

In this section, we describe the last two components in turn. The procedure's flow is shown schematically in Figure 5.1. In section 5.5, we will describe a series of experiments applying the method with various learning machines to a benchmark Web application.

5.3.1 Capacity model

We model the *capacity* of the Web application under a particular workload distribution as an unknown function $H(P;\Theta)$, where P is a particular workload pattern and Θ is a set of parameters.

As an (unrealistically simple) example of a possible form for the capacity function $H(P;\Theta)$, consider a simple sequential request processor that requires an average time of r_i to process a request in cluster c_i. In this case, we could write the capacity directly as

$$H(P,\vec{r}) = \frac{1}{\sum_{i=1}^{k} r_i \cdot P(c_i)}.$$

However, a real multi-tier Web application uses parallel processing at many levels to execute multiple responses concurrently. Without detailed knowledge of the application, the hardware it is running on, and the operating system resources allocated to it, we cannot in general hope to obtain a straightforward analytical expression for $H(P;\Theta)$. Instead, we estimate a statistical model for $H(P;\Theta)$ online from access log data. There are many possible parametric models for the function. For example, we might use the linear model

$$H(P;\vec{a}) = a_0 + a_1 P(c_1) + a_2 P(c_2) + \ldots + a_k P(c_k) + \epsilon, \qquad \text{(Equation 5.1)}$$

where $\vec{a} = (a_0, a_1, \ldots, a_k)$ are regression coefficients and ϵ is a Gaussian noise term with $\epsilon \sim N(0,\sigma^2)$. The regression coefficients can be recalculated after updating their sufficient statistics for all of the historical data every time a new observation is received.

Other forms and learning algorithms for $H(P;\Theta)$ are possible. For example, we could use support vector regression, Gaussian processes, or a multilayer perceptron. We experiment with several such learning machines in section 5.5.

5.3.2 Learning the capacity model online

To identify the relationship between workload pattern and system capacity online, we make a minimal assumption that we have a monitoring component that can log the response time for each request made against the system. From this component, towards estimating $H(P;\Theta)$, we extract training observations containing workload patterns and system capacity measurements. Here a capacity measurement is simply the number of requests per unit time that, under a specific workload pattern, the system was able to handle prior to saturation. Each training observation i for $H(P;\Theta)$ thus contains an observed number of requests $H^{(t)}$ and a workload pattern $P^{(t)}$. We write training vector $x^{(t)}$ as

$$\vec{x}^{(t)} = \left(H^{(t)}, P^{(t)}(c_1), P^{(t)}(c_2), \ldots, P^{(t)}(c_k)\right)$$

The period of time for estimating the workload pattern $P^{(t)}$ and capturing the number of requests $H^{(t)}$ should be fairly short, on the order of seconds or minutes. A valid training pattern is a $(H^{(t)}, P^{(t)})$ pair measured *just before* the system saturates due to overload. To measure saturation, we calculate the 95^{th} percentile of the response time during every interval t. If the response time remains under a predefined saturation level, we simply record the measurement and continue monitoring the logs. If the response time goes beyond the saturation level at time t, we construct a training pattern from

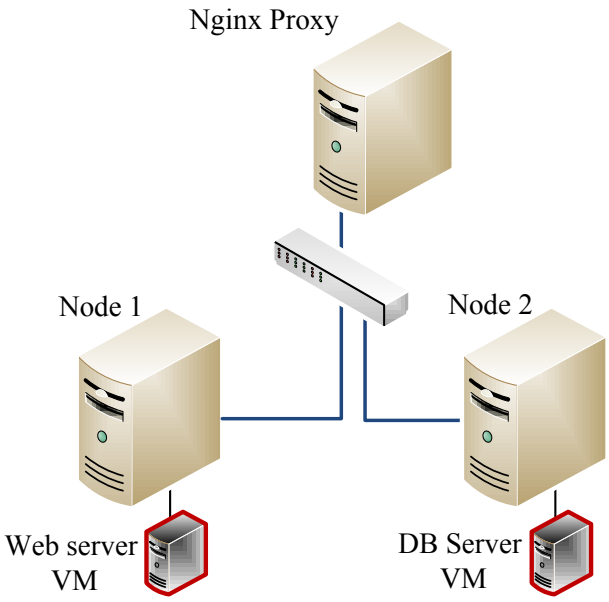


Figure 5.2: Experimental testbed.

the total number of requests ($H^{(t-1)}$) and workload pattern ($P^{(t-1)}(c_1), P^{(t-1)}(c_2), \ldots, P^{(t-1)}(c_k)$) using the logs for the previous interval, $t-1$. We then add the new observation to the regression model and continue monitoring the access logs.

Once the statistical model reaches the desired level of confidence in its estimates, external decision making components could begin to use the model to get a predicted system capacity for any workload pattern.

5.4 Experimental Environment

Here we describe an experimental environment consisting of a testbed virtual cluster, a benchmark Web application, and a synthetic workload generator aimed at evaluating our online capacity identification method.

5.4.1 Testbed System

We built a small testbed network using three physical machines (Front-end, Node1, and Node2) and a gigabit Ethernet. Figure 5.2 shows the network design for our testbed network. Front-end is an Intel Core 2 Duo machine with a 2.6 GHz CPU. Node1 and Node2 are 2.66 GHz CPUs. All machines have 2 GB RAM. We installed Xen hypervisor and deployed two virtual machines hosting the Apache Web server and the MySql database server on Node1 and Node2, respectively.

5.4.2 Benchmark Web Application

RUBiS (OW2 Consortium, 1999) 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 for our experimental evaluation.

5.4.3 Synthetic Workload Generation

We use `httperf` (Mosberger & Jin, 1998) to generate synthetic workload for our experiments. We generate a random workload for a specific duration with a required number of user sessions per second. To model the user sessions, in each workload generation experiment, we first prepare a list of random URIs representing the browsing behavior of one user. For each of the user browser requests in the list, we also extract the list of associated resources (images, CSS files, and so on) that the browser must also download in order to complete the page load. Once the complete list is generated, we then randomly subsample the URI set according to a desired mix of dynamic and static resources for the experiment. Once this random subset of the full session is identified, we keep the same URIs for all user sessions in that experiment.

5.5 Experimental Evaluation

Here we describe an experimental evaluation of our online capacity identification method using the previously-described experimental testbed with a variety of learning machines.

5.5.1 Pre-training

We generated synthetic workload contained different combinations of URIs consisting on dynamic and static contents for our sample benchmark Web application. We used the same URI-space partioning technique explained in Chapter 4, Section 4.5.1 to obtained the majority cluster ID for each URI path in the log file and retained this mapping for the training stage.

5.5.2 Dataset Generation

We identify identical user requests to the browsing-mix workload of the RUBiS Web application and identify the browser requests mapping for each user request. Then we automate the process to generate synthetic dataset. Algorithm 2 shows our approach to generate a synthetic dataset. We use $n = 375$ and $\tau = 1000$ ms. To generate random workload pattern, we generate 10 random user requests and identify browser requests against them and find out the cluster distribution. We keep increasing the number of user session for the workload until system 95^{th} percentile response time reaches beyond 1000 ms.

Input: size of data set (n), response time saturation level (τ), and user session step time interval (r).

Output: training set T containing workload patterns (P) paired with system capacity measurements (H).

$T \leftarrow \emptyset$

for $i = 1$ **to** n **do**

obtain random URIs to model one user session

calculate P from the selected URIs

$H = 0$

$u \leftarrow 1$ `/* # of user sessions to generate per second */`

repeat

generate workload containing u user sessions/second for time r

$h \leftarrow$ the number of requests processed

$t \leftarrow 95^{th}$ percentile of response time

if $t \leq \tau$ **then**

$H = h$

end

$u \leftarrow u + 1$

until $t > \tau$

$T \leftarrow T \cup \{(P, H)\}$

end

Algorithm 2: Dataset generation.

Table 5.1: Experimental results. Relative error and standard deviation for capacity prediction over a test set consisting of varying workload patterns.

Function	Average Relative Error	Standard Deviation
Gaussian process	6.42%	5.95%
Support vector regression	7.07%	6.57%
Multilayer perceptron	7.86%	7.815%
SMO regression	10.02%	12.31%
Multiple linear regression	11.24%	11.60%
Baseline	29.67%	29.19%

5.5.3 Model Learning using Regression

We split the dataset into 80% for training the different regression models and 20% for testing the models. Training and testing are disjoint datasets, training dataset use to estimate the parameters of the model and test dataset use to predict and calculate the error of the model.

Weka (Hall et al., 2009) is an open-source machine learning tool written in Java. It provides several algorithms to apply directly on datasets. We use the following regression algorithms available in Weka to learn and predict the capacity model for the sample benchmark multi-tier Web application:

Gaussian Processes

Weka provides Gaussian Process (Mackay, 1998) implementation for regression. We used the default parameter values in Weka to learn and test the regression model.

Support Vector Regression

LibSVM (Chang & Lin, 2001) is a library providing support vector machine and regression classifiers. Weka provides a wrapper class to use LibSVM. We used sigmoid kernel and performed grid search to identify the values of *cost*, *gamma*, and *epsilon* parameters. We found *cost* = 46340.95, *gamma* = 1.0, and *epsilon* = 0.5 from our training dataset.

Multilayer Perceptron

Weka provides Multilayer Perceptron implementation which uses back-propagation algorithm for classification and regression. We used it using the default parameter values given by the Weka.

SMO Regression

Sequential Minimal Optimization (SMO) regression implementation is available in Weka that uses support vector machine (Shevade, Keerthi, Bhattacharyya, & Murthy, 1999). We used SMO regression algorithm with default parameter values.

Linear Regression

Linear Regression provides a relationship between a scalar variable and a set of one or more variables. We used the linear regression implementation in Weka.

5.5.4 Results

We trained six system capacity models using the model and learning machines described in the previous section. We used 80% of the data set for training (estimating the parameters of the model) and reserved the remaining 20% for testing each model. For comparison, we also tested a baseline method that predicts capacity without exploiting the time-varying workload patterns; instead, the baseline method simply uses the average of the capacity measurements in the training set as the predicted capacity for any new workload. For each method, we calculated the percent relative error and the standard deviation of the percent relative error with respect to the corresponding actual capacity measurement. Table 5.1 shows the results of the comparison.

All of the learning machines using workload pattern identification outperform the baseline method. The best model, a Gaussian process regression model, gives only 6.42% relative error over the test set. The support vector regression and multilayer perceptron models give error rate close to that of the Gaussian process. The evaluation shows that the proposed approach would help to dynamically identify Web application capacity with very low error rates.

5.6 Discussion

In this paper, we have presented a black-box approach to identifying workload patterns and on-line learning of capacity models for multi-tier Web applications under varying workload characteristics. Our experimental evaluation using different learning machines shows that the approach could help cloud infrastructure service providers to perform proactive dynamic allocation of resources to multi-tier Web applications, meeting service level agreements at minimal cost. It is also possible that the

proposed approach could be used by the application itself to learn a capacity model and autonomously provision the required resources using the API provided by the cloud infrastructure provider.

The main limitation of the approach is that it requires sufficient time and access log data to learn the clustering and capacity models.

A possible extension of this work is to integrate our proposed approach with proactive resource provisioning and scaling towards offering service level agreements that provide maximum response time guarantees, and compare the solution with existing hardware profiling techniques used by Amazon EC2 Auto Scaling (Amazon Inc, 2009).

Chapter 6

Adaptive Resource Allocation for Back-end Mashup Applications

In previous chapters, I presented a system that idenfies and resolve bottlenecks in a multi- tier application automatically, provided a formal model to identify the workload patterns for multi-tier Web applications using access logs, and presented a method for learning appropriate application- and workload-specific resource provisioning policies in real time using machine learning tehcniques. I also presented an approach to identify the system capacity of multi-tier Web applications hosted on specific infrastructure. Now in this chapter, I demonstrate an approach to allocate cloud resources dynamically to scale gracefully in the presence of rapid increase in workload for multi-tier applications.

6.1 Introduction

Virtualization (VMWare, 2007; Paul et al., 2003) is one of the key technologies behind cloud computing infrastructures. It allows us to instantiate virtual machines dynamically on physical machines and allocate resources to them as needed. There are several benefits expected from virtualization, such as high availability, ease of deployment, ease of migration, ease of maintenance, and low power consumption, that help establish a robust infrastructure for cloud computing (Buyya, Yeo, & Venugopal, 2008; Armbrust et al., 2009). Many IT giants, such as IBM, Sun, Amazon, Google, and Microsoft, offer cloud-based computational and storage resource rental services to consumers. Consumers of these services host applications and store data for business or personal needs.

Cloud computing is rapidly growing as a research area spanning both academia and industry. The vast majority of the research has been oriented towards either batch-style applications for scientific computing or towards Web-centric applications, especially e-commerce applications. The typical scientific application uses single-instruction multiple data (SIMD) parallelization, without any dependencies on external services or between separate internal tiers. The typical Web application consists of a few tiers that are coupled, under the control of the application provider, and relatively easy to scale vertically or horizontally to meet performance goals. The needs of these types of applications are well understood, and cloud computing can fulfill them through on-demand resource provisioning, pay-per-use, and robust infrastructure.

Our research focuses on adaptive scaling of the virtual resources allocated to applications running on clouds. There is some existing work in this area. Amazon Auto Scaling (Amazon Inc, 2009) allows consumers to scale up or down according to criteria such as average CPU utilization across a group of compute instances. (Iqbal, Dailey, & Carrera, 2009) uses log-based monitoring and adaptive resource allocation to identify service-level agreement (SLA) violations in single-tier Web applications and scale the applications to satisfy the SLA. (Azeez, 2008) presents the design of an auto scaling solution based on incoming traffic analysis for Axis2 Web services running on Amazon EC2. (Bodik et al., 2009) 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. Dubeyb et al. (Dubey, Mehrotra, Abdelwahed, & Tantawi, 2009) present initial results from the use of dynamic regression and queuing modeling techniques to obtain an approximate system performance model for multi-tier Web application hosted in virtualized data centers. (Liu & Wee, 2009) monitors the CPU and bandwidth usage of virtual machines hosted on an Amazon EC2 cloud,

identifies the resource requirements for Web-based applications, and dynamically switches between different virtual machine configurations to satisfy the changing workloads.

Very little research has been done, however, on how to leverage the benefits of cloud computing for applications with alternatives to these well-understood architectures. In this chapter, we explore adaptive resource allocation for applications following an architectural pattern we call *Back-end Mashup*, in which the user-facing front end might be a relatively simple and lightweight Web interface, but the back end is a resource-intensive system that continuously collects and analyzes real-time data from external services or applications.

As an example back-end mashup, we have built a prototype application called `BuddyMonitor` that connects to any external chat server running XMPP (the eXtensible Messaging and Presence Protocol) and allows end users to monitor their chat "buddies" for specific durations of time and to examine their buddies' presence patterns over time. We find that as the number of concurrent monitors increases, `BuddyMonitor`'s resource demand on back-end virtual machines increases fairly rapidly, requiring allocation of additional virtual machines to accommodate additional requests without overloading the system.

To enable dynamic allocation of cloud resources for back-end mashup applications such as `BuddyMonitor`, we have built a prototype system that adaptively allocates resources to the application, ensuring that it can always accomodate new requests despite a-priori undefined resource utilization requirements. We evaluate the prototype resource allocator on a heterogeneous testbed cloud running `BuddyMonitor` and demonstrate that it prevents the application from reaching an unstable state despite growing resource utilization requirements.

In the rest of this chapter, I describe `BuddyMonitor`, our approach to adaptive resource allocation for back-end mashups, the prototype implementation, and an experimental evaluation of the prototype.

6.2 Buddy Monitor: An Example Back-end Mashup Application

Instant messaging (IM) provides real-time text communication between two or more people. Google, Microsoft, Yahoo, and Skype are some of the well-known IM providers. Most IM providers use XMPP to offer IM services. We developed a back-end mashup application named `BuddyMonitor` in Java using the Smack (Jive Software, n.d.) API that is able to connect to any XMPP server using credentials provided by an end user. Users can elect to monitor the presence or availability of a list of their IM buddies in real time for a particular duration of time, and the system records that presence information for later visualization and analysis.

In `BuddyMonitor`, a user's request to monitor a specific set of buddies for a certain duration of time is known as a *user monitor request*. Figure 6.1 shows the availability and presence patterns for five of one of the authors' real buddies on Google Chat on a specific day. The y axis shows the sampled presence code indicating whether the user is unavailable, away/busy, or available. User presence visualizations such as these are useful for various purposes, such as determining which of one's buddies are regularly online, what a buddy's work patterns are, and when the best time to catch a buddy online might be.

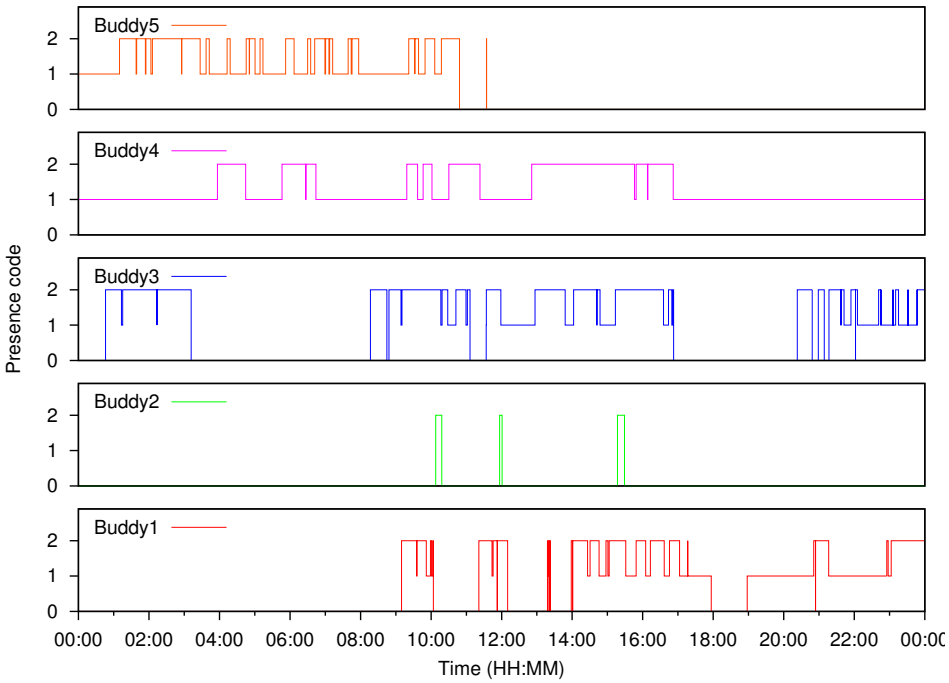


Figure 6.1: Availability and presence patterns for five buddies of a real Google Chat user for a specific day. The y axis indicates sampled presence codes indicating whether the user is *Unavailable* (code = 0), *Away/Busy* (code = 1), or *Available* (code = 2).

6.3 System Design and Implementation

To dynamically manage cloud resources utilized by virtual machines in private clouds, we developed two generic components: `VLBCoordinator` (Virtual Load Balancing Coordinator) and `VMProfiler` (Virtual Machine Profiler). `VLBCoordinator` interacts with a EUCALYPTUS cloud using Typica (Google Code, 2008). 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. `VMProfiler` is used to log the CPU utilization of each virtual machine. It exposes XML-RPC functions to obtain the CPU utilization of a specific virtual machine for the last n seconds. These components can be used with any cloud-based application running on EUCALYPTUS.

The `BuddyMonitor` back end is responsible for servicing user monitor requests and coordinates with the user-facing front end using a shared database. It obtains CPU usage statistics from `VMProfiler` and uses `VLBCoordinator` to spawn new virtual machines when necessary. Figure 6.2 shows deployment and the interation between the components in our system.

Figure 6.3 shows a flow diagram for the algorithm our `BuddyMonitor` back end prototype uses 1) to identify the maximum number of concurrent user monitor requests a virtual machine can process without bringing it to an unstable state and 2) to trigger the cloud scaling events required to adapt to increasing user request loads.

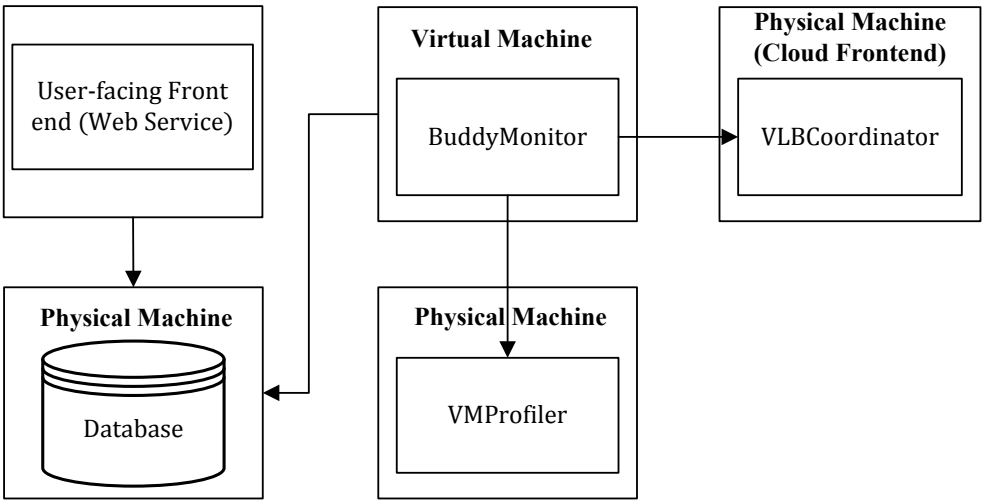


Figure 6.2: Component deployment and interaction in our prototype.

Table 6.1: Hardware configuration of physical machines used for the experimental testbed cloud.

Node	Type	CPU	RAM
Front-end	Intel Pentium IV	2.80 GHz	2 GB
DBServer	Intel Core 2 Duo	2.6 GHz	2 GB
Node1	Intel Pentium IV	2.66 GHz	1.5 GB
Node2	Intel Celeron	2.4 GHz	2 GB
Node3	Intel Pentium IV	2.66 GHz	0.5 GB

6.4 Experiments

In this section we describe the setup for an experimental evaluation of our prototype back-end mashup application running on a testbed cloud and interacting with a simulated large-scale XMPP service.

6.4.1 Testbed cloud

We built a small private heterogeneous compute cloud using five physical machines and EUCALYPTUS (Nurmi et al., 2008). The cloud consists one Cloud Controller (CLC), one Cluster Controller (CC), three Node Controllers (NCs), and one database server. 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. We installed a MySQL server on a physical machine named DBServer. Table 6.1 shows the hardware configuration of the machines, and Figure 6.5 shows the deployment of software components and network connections to the machines.

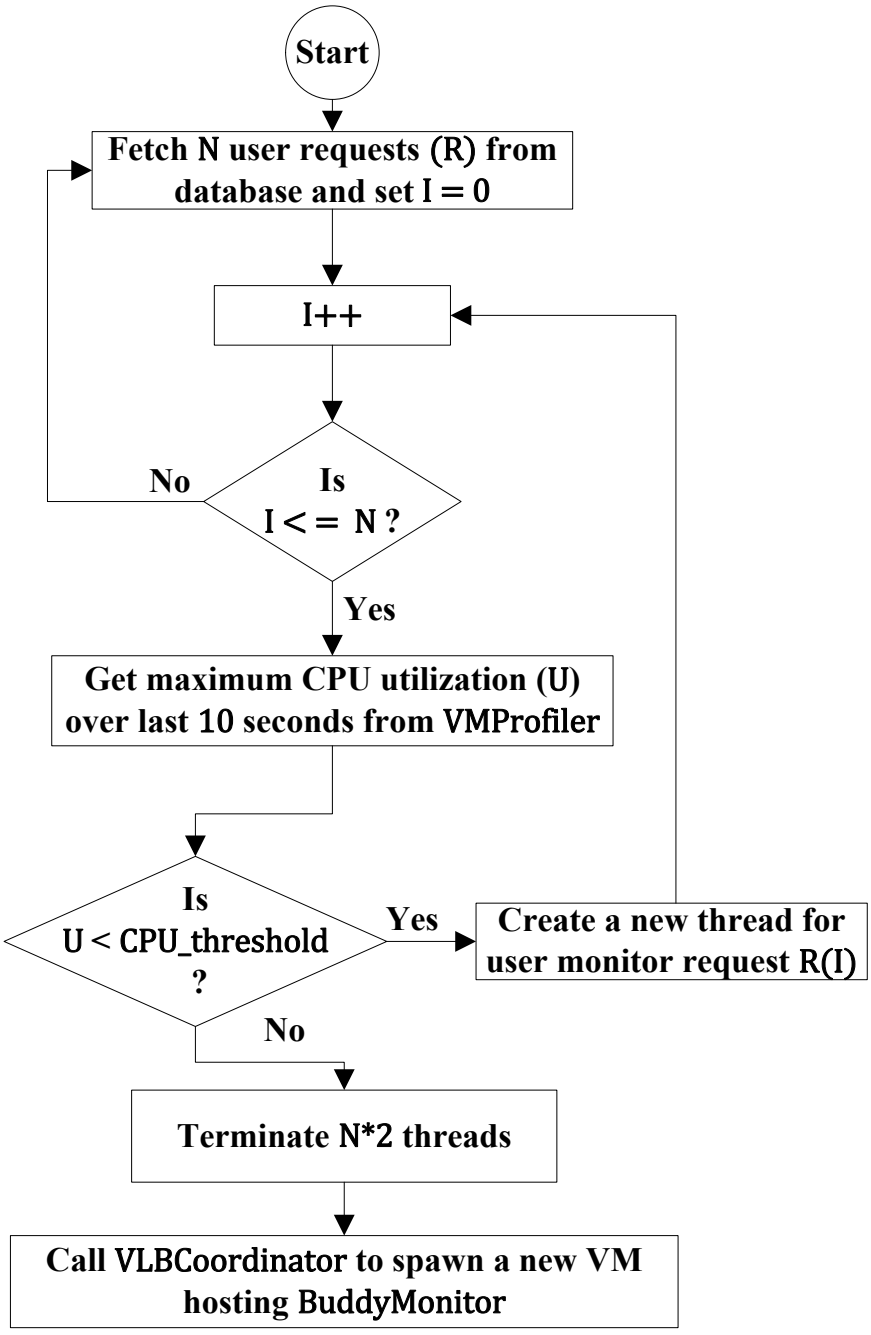


Figure 6.3: Flow diagram for the algorithm our `BuddyMonitor` back end prototype uses to identify the maximum number of concurrent user monitor requests a virtual machine can process and to trigger cloud scaling events.

6.4.2 `BuddyEmulator` and workload generation

The `BuddyMonitor` back-end gets user monitor requests from a database shared with the user-facing front end and responds to those requests by opening monitoring connections to an XMPP server. To simulate the effects of user base growth on `BuddyMonitor`, we needed access to an XMPP server with a large number of potential buddies whose status changes over time, and we needed to simulate different user monitor request workloads over time.

For the XMPP server, we installed ejabberd (*Ejabberd: Distributed fault-tolerant Jabber/XMPP server in Erlang*, n.d.), an open-source XMPP server providing basic IM services. We added 2,000 users and associated 50 roster items (buddies) with each user. To keep the simulation simple, we used the same 50 roster items for each user.

We then developed `BuddyEmulator`, a standalone Java program using the Smack API (Jive Software, n.d.) that connects to the ejabberd server and simulates changes to the presence and availability status of the 50 buddies over time. It does so by connecting each of the 50 shared buddies (associated with all 2,000 users) to ejabbered and continuously changing the presence and availability status of each buddy randomly for a random duration between 0 to 60 seconds.

Finally, we simulated a large number of users making user monitor requests using a SQL script that executes directly on the `BuddyMonitor` database to insert the requests. We fixed the duration of each user monitor request to 30 minutes. The green line in Figure 6.8 shows the number of active user monitor requests at each point of time in the simulation.

We performed two experiments based on this simulation of a real-world IM service, as described below.

6.4.3 Experiment 1: Static allocation

In this experiment, we established the experimental setup shown in Figure 6.4 and profiled the system's behavior against `BuddyEmulator`. We statically allocated only one virtual machine (VM1) to the `BuddyMonitor` back end. Here `VMProfiler` is only used to obtain the CPU usage of VM1 during the experiment.

6.4.4 Experiment 2: Adaptive allocation

In this experiment, we ported `BuddyMonitor` to our private testbed cloud and implemented the algorithm described in Figure 6.3 to leverage the benefits of cloud computing and adaptive resource provisioning for this back-end mashup application. Figure 6.5 shows the experimental setup we established. Initially, only one virtual machine (VM1) is alive and processing user monitor requests, while VM2 and VM3 are cached by EUCALYPTUS. As previously described, we used `VMProfiler` to monitor the CPU usage of each virtual machine, and we used `VLBCoordinator` to adaptively invoke additional virtual machines as required by the system. We used N = 10 and CPU_threshold = 65.0 in the algorithm of Figure 6.3.

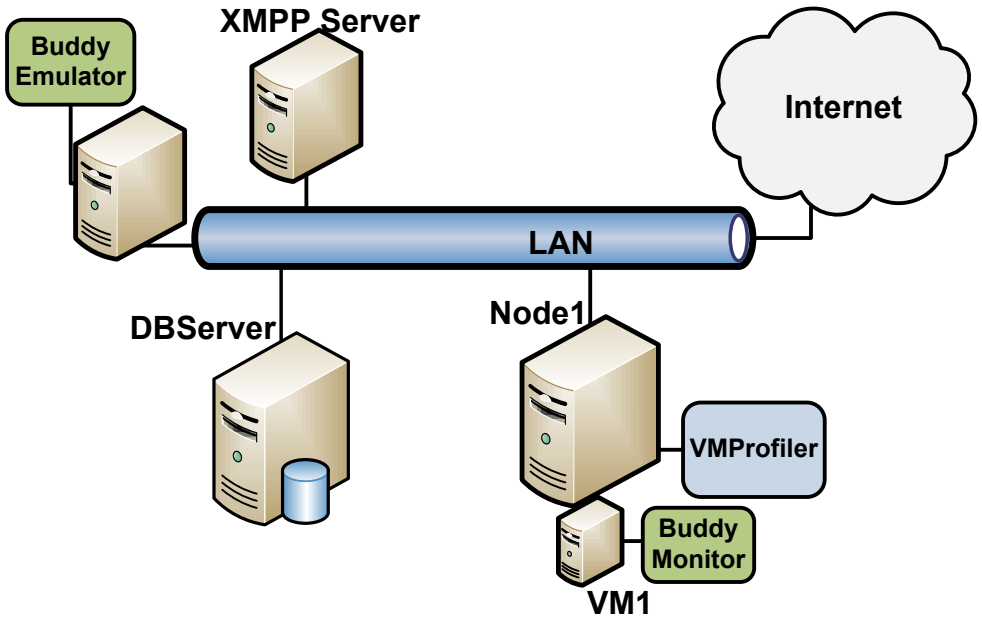


Figure 6.4: Setup for Experiment 1 (static resource allocation).

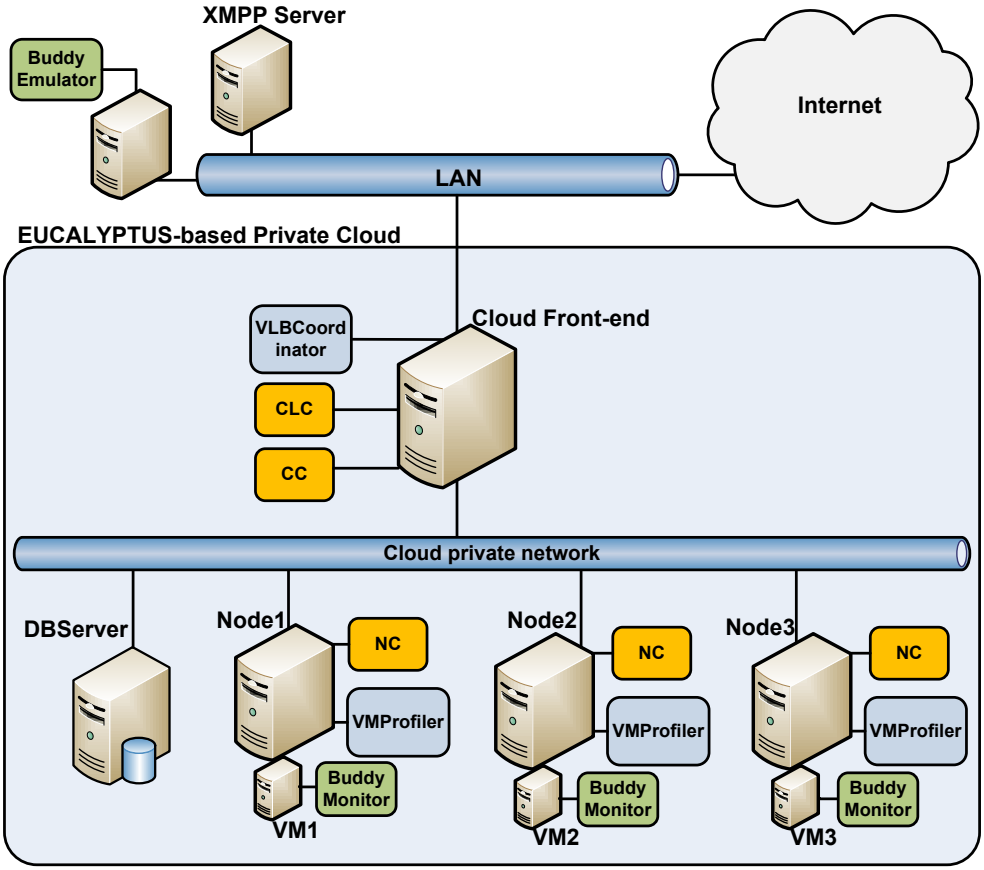


Figure 6.5: Setup for Experiment 2 (adaptive resource allocation).

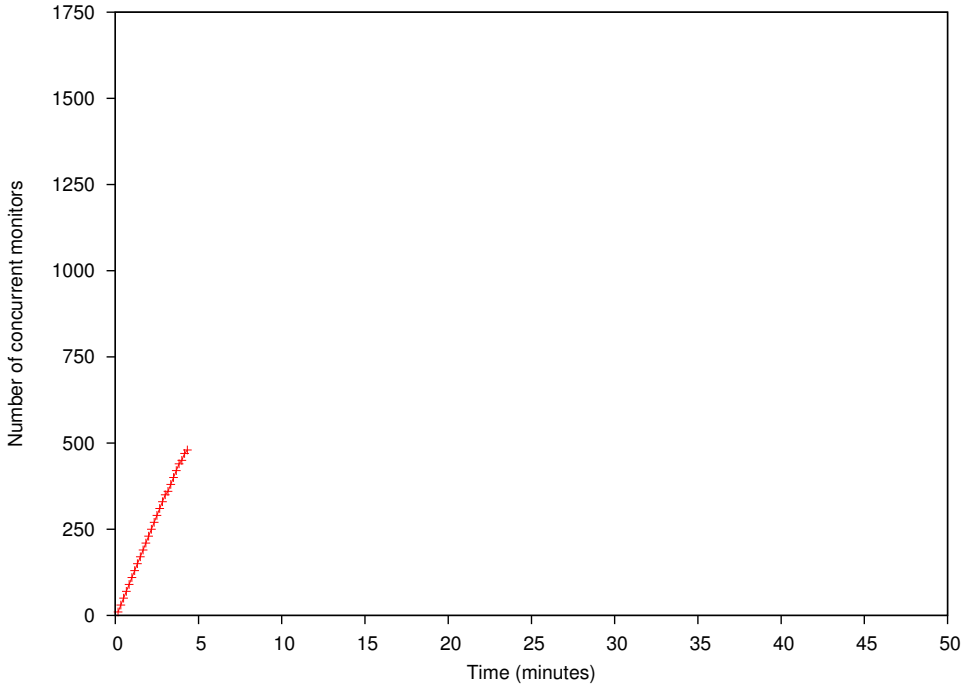


Figure 6.6: Number of concurrent services processed during Experiment 1.

6.5 Results

6.5.1 Experiment 1: Static allocation

This section describes the results we obtained in Experiment 1. Figure 6.6 shows the number of concurrent user monitors requests being serviced at each point in time. After successfully reaching 490 concurrent monitors, the system stopped responding and stopped servicing user monitor requests completely.

Figure 6.7 shows the CPU utilization of VM1 during the experiment. When the resource utilization of the system reaches some critical point, the system starts thrashing and the measured CPU utilization rapidly increases to over 90%. We halted the simulation after 20 minutes after finding that no user monitor requests were being handled at all.

With static resource allocation, even with a relatively small user base, to prevent catastrophic failure of the system, we would have to either limit the number of user requests allowed (not satisfy the needs of our users) or provision additional resources a priori (underutilizing those resources).

Since neither of these static solutions are suitable, in Experiment 2, we explore the use of adaptive allocation of cloud resources.

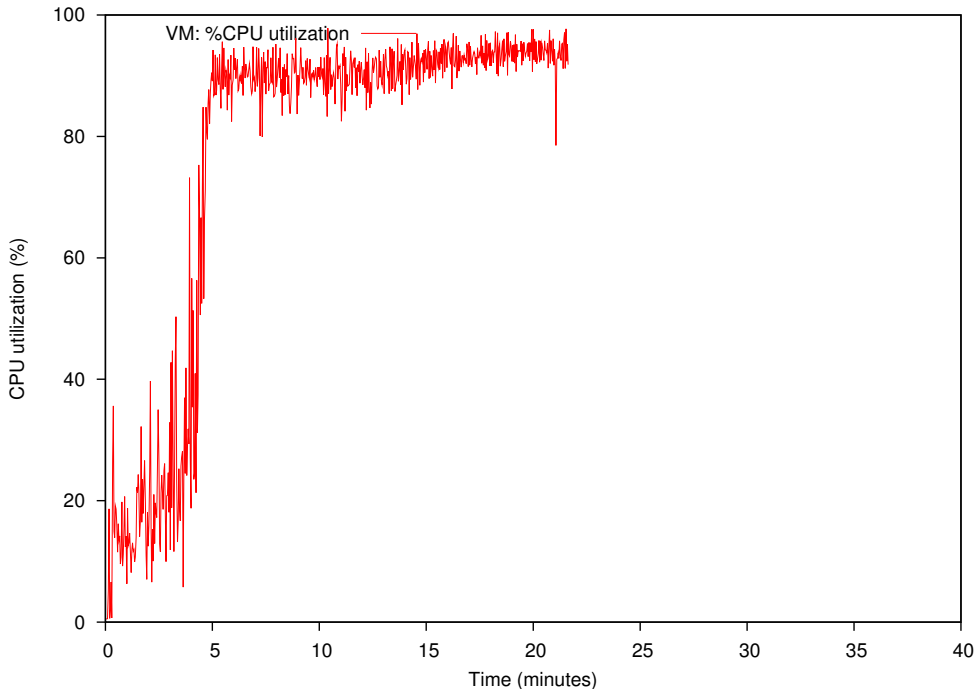


Figure 6.7: CPU utilization of virtual machine VM1 during Experiment 1.

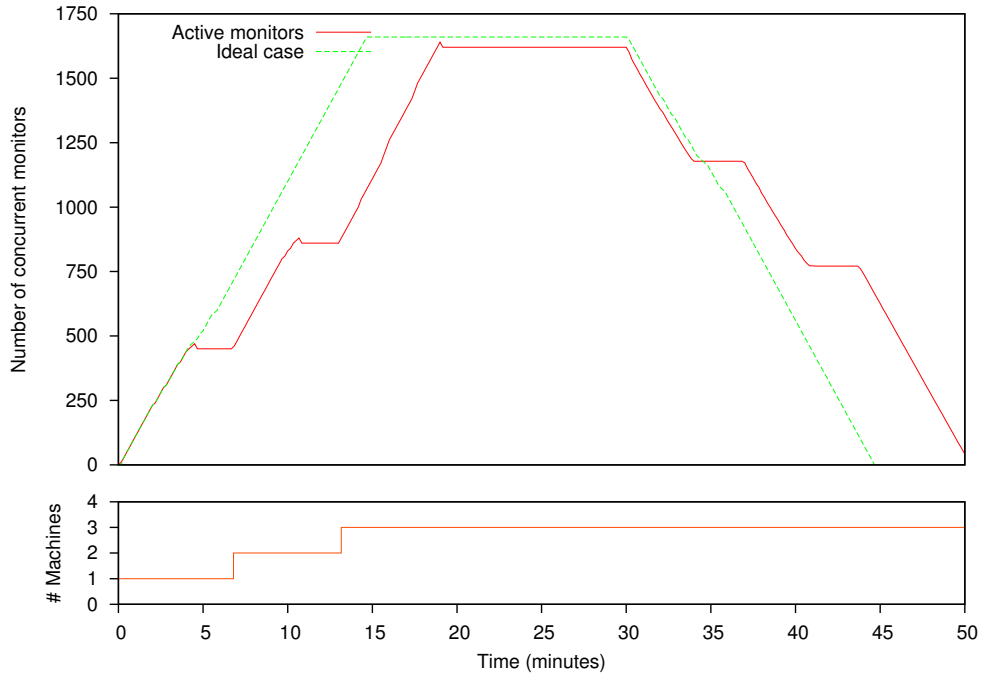


Figure 6.8: Number of concurrent user monitor requests processed during Experiment 2. The bottom graph shows the adaptive addition of virtual machines over time.

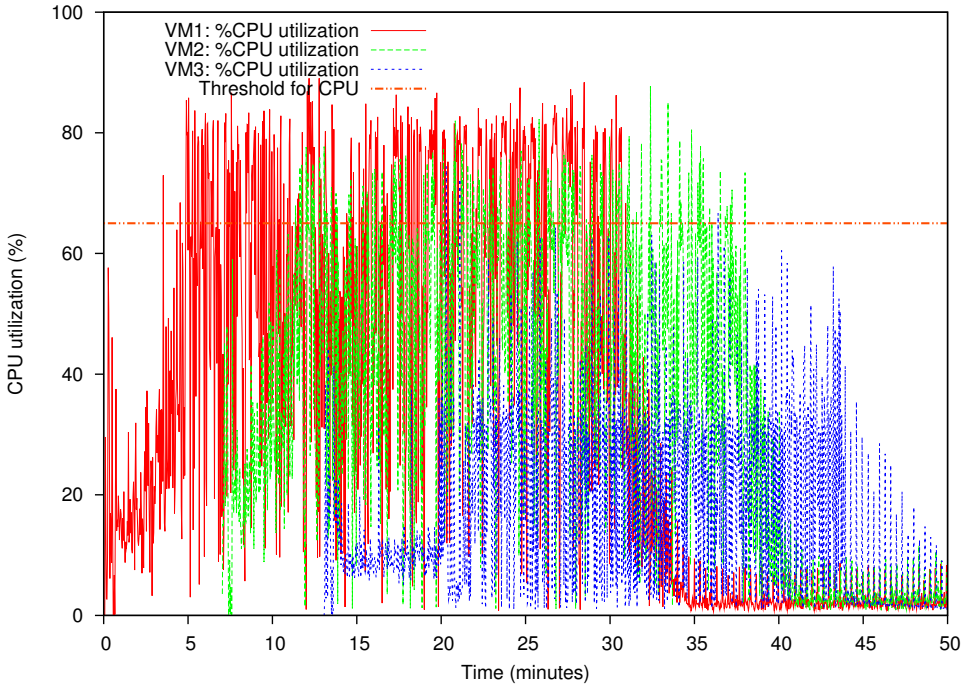


Figure 6.9: CPU utilization of virtual machines during Experiment 2.

6.5.2 Experiment 2: Adaptive allocation

This section describes the results of Experiment 2. Figure 6.8 shows the number of active user monitors during the experiment and compares it with the number of outstanding user monitor requests at each point in time. Initially, the system was configured with VM1 only. As the number of user monitors running on VM1 grows, the `BuddyMonitor` component on VM1 dynamically identifies the maximum number of concurrent monitor requests VM1 can handle and adaptively spawns VM2 to serve more user monitor requests. Whenever VM2 dynamically identifies the maximum number of concurrent user monitors it can handle, it then spawns VM3 to accommodate more requests. We observe linear growth in the number of active user monitors except during the time required to boot new VMs.

Figure 6.9 shows the CPU utilization of the virtual machines during the experiment, and Figure 6.10 shows the cumulative number of user presences logged by `BuddyMonitor` during the experiment. In contrast to the static allocation policy used in Experiment 1, the system is able to handle the rapid growth in user monitor requests without bringing any of the VMs to an unstable state.

One limitation of our current prototype is that it never services all of the active user monitor requests because we terminate 2N monitors each time a VM reaches the saturation point. We found that this was necessary to ensure stable behavior of our `BuddyMonitor` VMs. We plan to fix this issue in the future, but since our current goal is merely to explore adaptive VM allocation, the limitation is not important.

Another limitation of our current prototype, evident from the bottom graph of Figure 6.8, is that our current prototype is not capable of scaling down as the number of active user monitors decreases. Although this limitation is also unimportant for the current experiments, it is an interesting issue, since scaling down in this case would require migrating the monitors running on one underutilized

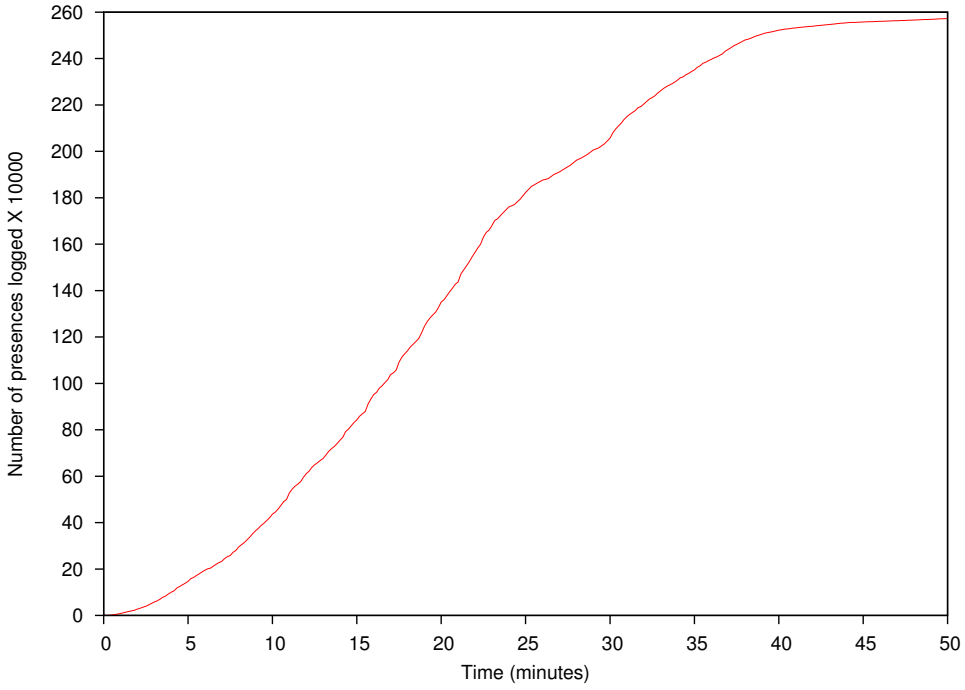


Figure 6.10: Cumulative number of presences logged by `BuddyMonitor` during Experiment 2.

VM to another underutilized VM.

6.6 Discussion

In this chapter, I have described a working prototype back-end mashup application system that achieves scalability, stability, and good performance through adaptive dynamic resource allocation on a heterogeneous private cloud. Our experimental results, based on a realistic simulation of a large scale external service, demonstrate the feasibility of our approach.

Besides addressing the already-discussed limitations of the `BuddyMonitor` application, in future work we plan to explore techniques to reduce the effect of virtual machine boot up time by predicting future workload and allocating resources more proactively based on those predictions, and to further understand the performance requirements of back-end mashups through further case studies.

Chapter 7

Conclusion

Multi-tier applications impose special challenges in the task of leveraging the benefits of cloud infrastructure. In this dissertation, I have addressed the problem of multi-tier applications hosted on clouds. We propose and evaluate methodologies for resolving multi-tier application bottlenecks under minimal hardware profiling and application-centric knowledge. In this concluding chapter, I summarize the contributions and provide concluding remarks.

7.1 Summary of Contribution

We began this dissertation by exploring a possibility to dynamically scale multi-tier Web applications hosted on clouds in order to satisfy specific maximum response time requirements using minimal resource allocation. In the quest, we proposed a system (explained in Chapter 2) for automatic detection and bottleneck resolution in a multi-tier Web application using combination of heuristic and predictive approach capable of satisfying specific response time requirements. The proposed solution minimizes the resources required to satisfy response time requirements using minimal hardware profiling and without requiring application-centric knowledge.

During the research, we observed and learned that application resource requirements depend on the application workload. We thus devised and proposed a simple model to identify workload patterns (explained in Chapter 3) from access logs using unsupervised machine learning. The proposed method is able to characterize Web resources appropriately based on resource demands without profiling any hardware resources.

We then extended the research by including the proposed workload pattern model to reactively identify bottlenecks for specific workload patterns and learning optimal resource allocation policies by monitoring access logs in real time (explained in Chapter 4). The proposed system initially uses a trial and error process to identify an appropriate bottleneck resolution policy in the context of a specific workload pattern, then it exploits that policy to reduce violations of the SLA while minimizing resource utilization. The approach does not require pre-deployment profiling or any insights about the application.

We then proposed and evaluated a black-box methodology for capacity prediction of a Web application hosted on a specific infrastructure (explained in Chapter 5). The proposed method learns a model and predicts the capacity of the application under varying workload patterns. The proposed method could be integrated with any predictive dynamic provisioning system for Web applications to satisfy response time requirements and minimize the SLA violation periods.

Finally, we demonstrated an approach (explained in Chapter 6) for allocating cloud resources dynamically to scale gracefully in the presence of rapid increases in workload for multi-tier applications that have resource-intensive back ends responsible for continuous collection and analysis of real-time data from external services or applications.

7.2 Concluding Remarks

Most cloud infrastructure and cloud hosting providers do not offer performance guarantees to owners of hosted applications. However, application owners need some specific guarantees to satisfy their business requirements. Applications based on the multi-tier architecture are good candidates to deploy on clouds. But since multi-tier architectures are complex, it is difficult to scale them automatically to offer specific performance guarantees.

To maximize revenue, we believe that cloud providers should use minimal hardware profiling and minimal application-centric knowledge to develop generic cloud-based services for hosting the majority of typical applications. In this dissertation, we explore the possibility of offering response time guarantees to multi-tier applications hosted on clouds using minimal hardware profiling and application-centric knowledge. We hope that our research could serve to guide cloud providers, entrepreneurs, and individual users in their quest to perform automatic resource management for multi-tier applications.

References

Allspaw, J.(2008). *The art of capacity planning*. Sebastopol CA, USA: O'Reilly Media, Inc.

Amazon Inc. (2009). *Amazon Web Services auto scaling.* (Available at `http://aws.amazon.com/autoscaling/`)

Anedda, P., Leo, S., Manca, S., Gaggero, M., & Zanetti, G. (2010). Suspending, migrating and resuming HPC virtual clusters. *Future Generation Computer Systems*, *26*(8), 1063 – 1072.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., et al. (2009, Feb). *Above the clouds: A berkeley view of cloud computing* (Tech. Rep. No. UCB/EECS-2009-28). Berkeley, CA, USA: EECS Department, University of California, Berkeley.

Azeez, A. (2008). *Auto-scaling Web Services on Amazon EC2.* (Available at `http://cloudcomputing.sys-con.com/node/612375/`)

Banga, G., & Druschel, P.(1997a). Measuring the capacity of a web server. In *Proceedings of the USENIX symposium on internet technologies and systems* (pp. 6–6). Berkeley, CA, USA: USENIX Association.

Banga, G., & Druschel, P.(1997b). Measuring the capacity of a Web server. In *Proceedings of the USENIX symposium on internet technologies and systems* (pp. 6–6). Berkeley, CA, USA: USENIX Association.

Banga, G., & Druschel, P. (1999). Measuring the capacity of a web server under realistic loads. *World Wide Web Journal (Special Issue on World Wide Web Characterization and Performance Evaluation*, *2*.

Bishop, C. M. (2007). *Pattern recognition and machine learning*. Secaucus, NJ, USA: Springer.

Bodik, P., Fox, O., Franklin, M. J., Jordan, M. I., & Patterson, D. A.(2010). Characterizing, modeling, and generating workload spikes for stateful services. In *SoCC '10: Proceedings of the 1st ACM symposium on cloud computing* (pp. 241–252). New York, NY, USA: ACM.

Bodik, P., Griffith, R., Sutton, C., Fox, A., Jordan, M., & Patterson, D. (2009). Automatic exploration of datacenter performance regimes. In *ACDC '09: Proceedings of the 1st workshop on automated control for datacenters and clouds* (pp. 1–6). New York, NY, USA: ACM.

Bodik, P., Griffith, R., Sutton, C., Fox, A., Jordan, M., & Patterson, D. (2009). Statistical machine learning makes automatic control practical for internet datacenters. In *HotCloud '09: Proceedings of the workshop on hot topics in cloud computing.* San Diego, CA, USA: USENIX Association.

Bu, X., Rao, J., & Xu, C.-Z. (2009). A reinforcement learning approach to online web systems auto-configuration. In *ICDCS '09: Proceedings of the 2009 29th IEEE international conference on distributed computing systems* (pp. 2–11). Washington, DC, USA: IEEE Computer Society.

Buyya, R., Yeo, C. S., & Venugopal, S. (2008). Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications* (pp. 5–13). Washington, DC, USA: IEEE Computer Society.

Chang, C.-C., & Lin, C.-J. (2001). *LIBSVM: A library for support vector machines.* (Available at `http://www.csie.ntu.edu.tw/\~cjlin/libsvm/`)

Cherkasova, L., Tang, W., & Singhal, S. (2004). An SLA-oriented capacity planning tool for streaming media services. In *DSN '04: Proceedings of the 2004 international conference on dependable systems and networks* (pp. 743–752). Washington, DC, USA: IEEE Computer Society.

Czajkowski, G., Wegiel, M., Daynes, L., Palacz, K., Jordan, M., Skinner, G., et al. (2005). Resource management for clusters of virtual machines. In *CCGRID '05: Proceedings of the fifth IEEE international symposium on cluster computing and the grid* (pp. 382–389). Washington, DC, USA: IEEE Computer Society.

Dejun, J., Pierre, G., & Chi, C.-H. (2011). Resource provisioning of Web applications in heterogeneous clouds. In *WebApps '11: Proceedings of the 2nd USENIX conference on*

Web application development (pp. 5–5). Berkeley, CA, USA: USENIX Association.

Dubey, A., Mehrotra, R., Abdelwahed, S., & Tantawi, A.(2009, October). Performance modeling of distributed multi-tier enterprise systems. *SIGMETRICS Performance Evaluation Review*, *37*, 9–11.

Ejabberd: Distributed fault-tolerant Jabber/XMPP server in Erlang. (n.d.). (Available at `http://www.igniterealtime.org/projects/smack/`)

Football Association, F. I. de. (2011). *The official FIFA web site.* Zurich, Switzerland. (Available at `http://www.fifa.com/`)

Foster, I., Freeman, T., Keahy, K., Scheftner, D., Sotomayer, B., & Zhang, X.(2006). Virtual clusters for grid communities. In *CCGRID '06: Proceedings of the sixth IEEE international symposium on cluster computing and the grid* (pp. 513–520). Washington, DC, USA: IEEE Computer Society.

Gerald, T., Nicholas K., J., Rajarshi, D., & Mohamed N., B. (2006). A hybrid reinforcement learning approach to autonomic resource allocation. In *Proceedings of the 2006 IEEE international conference on autonomic computing* (pp. 65–73). Washington, DC, USA: IEEE Computer Society.

Google Code. (2008). *Typica: A Java client library for a variety of Amazon Web Services.* (Available at `http://code.google.com/p/typica` [Online; accessed 16-Nov-2009])

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009, November). The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, *11*, 10–18.

Hartigan, J. A., & Wong, M. A.(1979). A K-means clustering algorithm. *Applied Statistics*, *28*, 100–108.

Iqbal, W.(2011). *Lecture Buddy: Real time teaching evaluation and feedback system.* (Available at `http://www.lecturebuddy.com`)

Iqbal, W., Dailey, M., & Carrera, D. (2009). 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* (pp. 243–253). Berlin,

Heidelberg: Springer-Verlag.

Iqbal, W., Dailey, M. N., Carrera, D., & Janecek, P. (2011, June). Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Gener. Comput. Syst.*, *27*, 871–879.

Jive Software. (n.d.). *Smack API.* (Available at `http://www.igniterealtime.org/projects/smack/`)

Kant, K., & Won, Y. (1999, September). Server capacity planning for Web traffic workload. *IEEE Transactions on Knowledge and Data Engineering*, *11*, 731–747.

Khanna, G., Beaty, K., Kar, G., & Kochut, A. (2006). Application performance management in virtualized server environments. In *NOMS '06: Network operations and management symposium* (p. 373-381). Vancouver, BC: IEEE Computer Society.

Larry, W. (2006). *All of nonparametric statistics (Springer texts in statistics).* Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Liu, H., & Wee, S. (2009). Web server farm in the cloud: Performance evaluation and dynamic architecture. In *CloudCom '09: Proceedings of the 1st international conference on cloud computing* (pp. 369–380). Berlin, Heidelberg: Springer-Verlag.

Mackay, D. J. (1998). *Introduction to Gaussian processes.* Dept. of Physics, Cambridge University, UK. (Available at `http://wol.ra.phy.cam.ac.uk/mackay/gpB.ps.gz`)

Martin, A., & Tai, J. (2000). A workload characterization of the 1998 World Cup web site. *IEEE Network*, *14*.

McGee, M. (2009). *Michael Jackson's death: An inside look at how Google, Yahoo, and Bing handled an extraordinary day in search.* (Available at `http://searchengineland.com/michael-jackson-extraordinary-day-in-search-21641`)

Mosberger, D., & Jin, T. (1998, December). httperf: A tool for measuring web server performance. *SIGMETRICS Performance Evaluation Review*, *26*, 31–37.

Nicolas, B., Thanasis G., P., & Karl, A. (2011). Automatic SLA-driven provisioning for cloud applications. In *CCGrid '11: Proceedings of the 2011 international sympo-*

sium cluster, cloud and grid computing. Newport Beach, CA, USA: IEEE Computer Society.

Nicolas, P., David, C., Ricard, G., Jordi, T., & Eduard, A. (2010). Characterization of workload and resource consumption for an online travel and booking site. In *Proceedings of 2011 IEEE international symposium on workload characterization* (pp. 1–10). Atlanta, GA, USA: IEEE Computer Society.

Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., et al. (2008). *EUCALYPTUS: A technical report on an elastic utility computing archietcture linking your programs to useful systems* (Tech. Rep. No. 2008-10). Santa Barbara, CA, USA: UCSB Computer Science.

Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., et al. (2009). The Eucalyptus open-source cloud-computing system. In *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM international symposium on cluster computing and the grid* (pp. 124–131). Washington, DC, USA: IEEE Computer Society.

OW2 Consortium. (1999). *RUBiS: An auction site prototype.* (`http://rubis.ow2.org/`)

Padala, P., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., et al. (2007). Adaptive control of virtualized resources in utility computing environments. In *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys european conference on computer systems 2007* (pp. 289–302). New York, NY, USA: ACM.

Paul, B., Boris, D., Keir, F., Steven, H., Tim, H., Alex, H., et al. (2003). Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth acm symposium on operating systems principles* (pp. 164–177). New York, NY, USA: ACM.

Rao, J., & Xu, C.-Z. (2010). Online capacity identification of multitier websites using hardware performance counters. *IEEE Transactions on Parallel and Distributed Systems*, *99*.

Rodero-Merino, L., Vaquero, L. M., Gil, V., Galn, F., Fontn, J., Montero, R. S., et al. (2010). From infrastructure delivery to service management in clouds. *Future Generation Computer Systems*, *26*(8), 1226 – 1240.

Sharma, A., Bhagwan, R., Choudhury, M., Golubchik, L., Govindan, R., & Voelker, G. M. (2008, August). Automatic request categorization in internet services. *SIGMETRICS Performance Evaluation Review*, *36*, 16–25.

Shevade, S., Keerthi, S., Bhattacharyya, C., & Murthy, K.(1999). Improvements to the SMO algorithm for SVM regression. *IEEE Transactions on Neural Networks*.

Singh, R., Sharma, U., Cecchet, E., & Shenoy, P.(2010). 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.* Washington, DC, USA: IEEE Computer Society.

Sundararaj, A. I., Sanghi, M., Lange, J. R., & Dinda, P. A.(2006). Hardness of approximation and greedy algorithms for the adaptation problem in virtual environments. In *ICAC '06: Proceedings of the 2006 IEEE international conference on autonomic computing* (pp. 291–292). Washington, DC, USA: IEEE Computer Society.

Sysoev, I.(2002). *Nginx.* (Available at `http://nginx.net/`)

Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., & Tantawi, A. (2005, June). An analytical model for multi-tier internet services and its applications. *SIGMETRICS Performance Evaluation Review*, *33*, 291–302.

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

Villela, D., Pradhan, P., & Rubenstein, D. (2007, February). Provisioning servers in the application tier for e-commerce systems. *ACM Transaction on Internet Technology*, *7*.

VMWare.(2007). *Understanding full virtualization, paravirtualization, and hardware assist.* Whitepaper.

Wang, Z., Zhu, X., Padala, P., & Singhal, S. (2007). Capacity and performance overhead in dynamic resource allocation to virtual containers. In *IM '07. 10th IEEE international symposium on integrated management* (pp. 149–158). Dublin, Ireland: IEEE Computer Society.

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279-292.

(10.1007/BF00992698)

Wikipedia. (2009). *Amazon Elastic Compute Cloud.* (Available at http://en.wikipedia.org/w/index.php?title=Amazon_Elastic_Compute_Cloud&oldid=272762076 [Online; accessed 1-March-2009])

WMware. (2010). *VMware Distributed Resource Scheduler (DRS).* (Available at http://www.vmware.com/products/drs/)

Wustenhoff, E. (2002). Service Level Agreement in the Data Center. *Sun BluePrint Online*.

xkoto. (2009). *GRIDSCALE database virtualization software.* (http://www.xkoto.com/products/)

Zhu, X., Wang, Z., & Singhal, S. (2006). Utility-driven workload management using nested control design. In *ACC '06: American control conference.* Minneapolis, Minnesota USA: IEEE Computer Society.