Hasso-Plattner-Institut
University of Potsdam
Internet Technology and
Systems Group

Scalability and Performance Management of Internet Applications in the Cloud

A thesis submitted for the degree of
"Doktors der Ingenieurwissenschaften"
(Dr.-Ing.)
in IT Systems Engineering

Faculty of Mathematics and Natural Sciences
University of Potsdam

By: Wesam Dawoud
Supervisor: Prof. Dr. Christoph Meinel

Potsdam, Germany, March 2013

Published online at the
Institutional Repository of the University of Potsdam:
URL http://opus.kobv.de/ubp/volltexte/2013/6818/
URN urn:nbn:de:kobv:517-opus-68187
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-68187

To my lovely parents
To my lovely wife Safaa
To my lovely kids Shatha and Yazan

Acknowledgements

At Hasso Plattner Institute (HPI), I had the opportunity to meet many wonderful people. It is my pleasure to thank those who supported me to make this thesis possible.

First and foremost, I would like to thank my Ph.D. supervisor, Prof. Dr. Christoph Meinel, for his continues support. In spite of his tight schedule, he always found the time to discuss, guide, and motivate my research ideas. The thanks are extended to Dr. Karin-Irene Eiermann for assisting me even before moving to Germany. I am also grateful for Michaela Schmitz. She managed everything well to make everyones life easier. I owe a thanks to Dr. Nemeth Sharon for helping me to improve my English writing skills.

My friends and office mates Raja and Christian, it was a nice time that we spent together. I am thankful for every German word or piece of knowledge you had passed to me. Cycling to HPI was enjoyable and helped me to start nice and productive days.

Matthias, Franka, Birgit and Haojin; it was always enjoyable chatting with you. Lutz, thanks for the tiny breaks that helped refreshing mind and continuing work. Aaron, Amir, Xiaoyin and Feng; the after-work basketball and football games were a must to get rid of the daily stress.

Ahmed, Eyad, Ibrahim and Nuhad; I will miss our late nights in the coffee shop discussing everybodys research ideas, exchanging experience, and supporting each other.

I owe a lot to my family Mum, Dad, and my wife Safaa for her love, trusts, and support especially at hard moments.

Definitions

Scale out: To scale out (scale horizontally) is to add more nodes to the system. An example might be adding more web instances to web tier.

Scale up: To scale up (scale vertically) is to add resources to the same node in a system. An example might be to add more physical memory (i.e., RAM) to a database node.

Scale down: To scale down is to release some acquired resources, either by releasing some nodes or by removing some of the node's resources.

Scalable architecture: It is an architecture enables the Internet application to scale rapidly, automatically, and transparently.

Service Level Agreement (SLA): SLA is an agreement outlining a specific service commitment made between contract parties – a service provider and its customer. The agreement describes the overall service, support details, financial aspects of service delivery, penalties, terms and conditions, and performance metrics that govern service delivery.

Service Level Objective (SLO): SLO is specific measurable characteristic of the SLA such as availability, throughput, response time, or quality. An example of response time as an objective is: "95% of the requests to an Internet application should be answered in less than 100 milliseconds measured over 24 hours".

On premises infrastructure: It is an infrastructure hosted in the facility of an organization, such as a university datacenter hosted within the university buildings.

Off premises infrastructure: It is an infrastructure hosted in the facility of another organization, such as the public cloud provided by Amazon EC2.

Abstract

Cloud computing is a model for enabling on-demand access to a shared pool of computing resources. With virtually limitless on-demand resources, cloud environments enable the hosted Internet applications to quickly cope with the spikes in workload. However, the overhead caused by the dynamic resource provisioning exposes the Internet applications to periods of under-provisioning and performance degradation. Moreover, the performance interference, due to the consolidation in cloud environments, complicates the performance management of Internet applications.

In this dissertation, we propose two approaches to mitigate the impact of the resource provisioning overhead. The first approach employs control theory to scale resources vertically and cope fast with workload. This approach assumes that the provider has knowledge and control over the platform running in the virtual machines (VMs), which limits it to Platform as a Service (PaaS) and Software as a Service (SaaS) models. The second approach is a customer-oriented one that deals with the horizontal scalability in an Infrastructure as a Service (IaaS) model. It addresses the trade-off problem between cost and performance with a multi-goal optimization solution. This approach finds the scale thresholds that achieve the highest performance with the lowest increase in the cost. Moreover, the second approach employs a proposed time series forecasting algorithm to scale the application proactively and avoid under-utilization periods. Furthermore, to mitigate the interference impact on the Internet application performance, we developed a system which finds and eliminates the VMs suffering from performance interference. The developed system is a light-weight solution that does not imply provider involvement.

To evaluate our approaches and the designed algorithms at large-scale level, we developed a simulator called (ScaleSim). In the simulator, we implemented scalability components acting as the scalability components of Amazon EC2. The current scalability implementation in Amazon EC2 is used as a reference point for evaluating the improvement in the scalable application performance. ScaleSim is fed with realistic models of the RUBiS benchmark extracted from the real environment. The workload is generated from the access logs of the official website of 1998 world cup. The results show that optimizing the scalability thresholds and adopting proactive scalability can mitigate 88% of the resources provisioning overhead impact with only a 9% increase in the cost.

Contents

List of Figures

Chapter 1

Introduction

Internet applications' usage is crucial part of everybody's cyber life. Whether provided as profit (e.g., online retailer) or non-profit services (e.g., Wikipedia), Internet applications are likely to be delivered with a high quality of service (QoS). The workload of an Internet application varies according to the time of the day and rises sharply on occasions. For example, an online retailer faces a daily cyclical workload variation and high spikes at special occasions such as Christmas. On the other hand, a video games discussion forum endures a cyclical workload variation, but experiences spikes of workload with each release of a new game, which is hard to predict. In fact, these applications are exist even before the cloud computing emergence. However, the cloud infrastructure provides an elastic environment for these applications to scale up and down according to workload variation.

With virtually limitless on-demand resources, cloud computing offers a scalable and fault tolerant environments that enable applications to handle workload variations. In current cloud computing environments, scalability and performance management of an Internet application running in Software as a Service (SaaS) model or in Platform as a Service (PaaS) model are the provider's responsibility. On the other hand, scalability and performance management of an Internet application running in the Infrastructure as a Service (IaaS) model is the customer's responsibility. Therefore, we concentrate our research on Internet applications hosted in IaaS environment.

1.1 Motivation

Cloud computing is a model enabling the Internet application to cope rapidly with workload variations. Nevertheless, provisioning resources in the cloud implies an overhead. The overhead can lead to periods of over-utilization that degrade the performance [76]. Moreover, due to workload consolidation in cloud infrastructures, the performance of an Internet application can be influenced by other co-located applications [74] [100] [108] [44].

1.2 Internet Application Performance Management Challenges

In this section, we explain several challenges confronting managing the performance of Internet applications. Dealing with these challenges properly is necessary for maintaining a high QoS.

1.2.1 Complex Resource to Performance Mapping

Modeling the relationship between computing system resources, such as the CPU utilization, and the system performance metrics, such as the response time, is important to manage a system performance dynamically. In mechanical systems, control theory is efficiently used for modeling systems, while the relation between a mechanical system input and output is governed by physical laws (e.g., Newton's law). In computing systems, the relationship between the system inputs and the system outputs is not clearly defined [47]. Moreover, some models, such as CPU utilization, show a bimodal behavior according to request rate and CPU entitlement [114]. Also, the rapid development and the diversity of Internet applications make it inappropriate to set a uniform threshold as an indicator for abnormal performance [90], as we explain empirically in Section 6.1.

1.2.2 Complexity of Multi-tier Systems

Typically, an Internet application is implemented as a multi-tier system. The multi-tier system provides a flexible, scalable, and modular approach for designing Internet applications. Each tier in the multi-tier system provides certain functionality. Nevertheless, the inter-tier's dependency propagates the performance degradation in one tier to the whole system. Moreover, some tiers employ replication and caching, which makes understanding the system behavior and managing the performance a non-trivial task [103].

1.2.3 Overhead of Resource Allocation

The overhead of resource allocation is a problem associated with Internet applications hosted in IaaS environment. Part of the overhead is attributed to the provider [76], where a complex management process is taken with each request for a new VM. According to Mao et al. [76], the initialization time of a VM in the cloud depends on many parameters including the provider, the location, the operating system, and the VM type. Another part of the overhead is attributed to the client's setup, which is controllable, as we explain in Section 5.2.1. Practically, the overhead of initializing computation power resources results in periods of resource over-utilization that hurt the application performance. A proactive scaling algorithm that predicts the coming workload and provision the resources in advance can help mitigating the QoS degradation, as we explain in Chapter 7.

1.2.4 Highly Variable Workloads

The workload of Internet applications varies dynamically over multiple periods of time. A period of a sudden increase in workload can lead to overloading resources when the volume of the workload exceeds the capacity of available resources. The length of overload periods increases proportionally with the overhead of resources provisioning. During overload periods, the response time may grow to very high levels where the service is degraded or totally denied. In production environments, such behavior is harmful for the reputation of Internet application providers.

1.2.5 Large-scale Management

Managing resources at large-scale in both public and private clouds is quite challenging. Many approaches developed towards managing Internet applications' performance are prototypes implemented at small-scale controlled environments. However, it is not necessarily that these approaches will work well at large-scale production environments. Moreover, the enormous number of the hosted VMs and the possible combinations of the hosted software within each VM make it hard or even impossible for the IaaS provider to maintain the performance of individual customers' applications.

To ease the resource management at a very large-scale infrastructure, some providers (e.g., Amazon EC2) apply a one-to-one mapping between virtual and physical CPU and memory resources [52]. In other words, the large-scale public cloud providers may sacrifice exploiting higher levels of utilization, looking for a better isolation and a feasible management tactic.

In the private cloud, resource management solutions like VMware DRS [5] and Microsoft PRO [2] lead to better performance isolation and higher utilization of hardware resources [52]. Nevertheless, according to Gulati et al. [52], during prototyping and evaluating a scaled version of VMware DRS, they realized the complexity of managing cloud-scale systems. As a team responsible for VMware DRS shipment, Gulati et al. [52] believe that increasing the consolidation ratio and improving the performance isolation at a large-scale (i.e., cloud-scale) still in need for more research.

1.2.6 Performance Interference

Performance interference in the cloud is a problem inherited from the virtualization technology [70] [78] [64] [53]. The contention for shared resources (e.g., memory bus and I/O devices) exposes the performance of competing VMs to degradation. Unfortunately, the contention for resources is typically hidden and cannot be measured by the conventional monitoring metrics provided by cloud providers. However, it can lead to unpredictable increase in the response time of Internet applications [89] [74]. Currently, the typical SLA of an IaaS provider describes only the annual up time of VM instances, but it does not discuss the

potential performance degradation caused by performance interference.

1.3 Research Questions

During our research, we investigate a set of questions that construct this dissertation storyline. The questions are:

1. What is a scalable application? How to build an Internet application scaling efficiently?

2. Virtualization technology provides dynamic management of resources. What is the possibility of exploiting that for maintaining Internet applications' performance?

3. What about the current implementation of the scalability in public IaaS environments? Is it efficient? Is there a way to improve it?

4. Performance interference is common in public IaaS environments. How to discover and avoid that?

1.4 Contributions, Assumptions, and Research Methodology

Towards an efficient scalability and performance management of an Internet application in cloud environments, we address the challenges mentioned in Section 1.2. At the beginning of our research, we exploit the control theory and the development of virtualization technologies to scale resources rapidly. Using control theory to scale resources vertically was a trend in the years between 2005 and 2009 [114] [123] [88] [113] [56]. The idea depends on redistributing the physical host resources dynamically (i.e., scale vertically) among the hosted VMs and scaling up overloaded VMs on account of the VMs with a low workload. The vertical scalability shows efficiency for maintaining Internet application performance. Nevertheless, it carries out complex management task to the provider, which makes it infeasible for large-scale environments.

Keeping in mind the fact that the cloud provider already deals with many complex management tasks, we start investigating customer-oriented solutions that do not imply the provider involvement in the performance management of Internet applications. We assume that the provider does its part by providing the possible, but not the best, performance isolation nowadays. We also assume that the customer is aware of his/her application performance thresholds and requirements. On the light of these assumptions, we direct our research towards understanding the available scalability components and finding the optimal configurations that maintain the performance of an Internet application. Moreover, we suggest proactive scalability to mitigate the impact of the overhead of resources provisioning on an Internet application performance. In our approach, we consider the trade-off between the performance and the cost as an optimization problem. Finally, to avoid the performance interference in public infrastructure we designed a system that detects and replaces the contended VMs automatically.

Our research depends on both physical and simulation environments for implementing and validating our contributions. We use the physical environment for modeling the performance of Internet applications, and then feeding these models to simulation environment. In our simulation environment (ScaleSim), we examine running Internet applications at large-scale, where we can examine the efficiency of our designed algorithm. The results show that optimizing scalability thresholds and adopting proactive scalability can mitigate 88% of the impact of resource provisioning overhead with only 9% increase in the cost. The results are promising, while we propose a customer-oriented solution applicable for current IaaS environments.

1.5 Dissertation Organization

- In Chapter 2, we present the development of Internet application hosting during the last decade.
- In Chapter 3, we summarize the research done towards maintaining the performance of Internet applications and express our research position among the related work.

- In Chapter 4, we investigate employing vertical scalability in the cloud to offer an agile scalability for Internet applications.
- In Chapter 5, we investigate the facts that can degrade the performance of an Internet application hosted in IaaS model.
- In Chapter 6, we model the performance of an Internet application at physical environment, and then run a large-scale simulation to evaluate current reactive scalability of Internet applications in the IaaS model.
- In Chapter 7, we study improving the current reactive scalability by tuning the scalability parameters. It is followed by the design and the evaluation of our proactive scalability.
- In Chapter 8, we study the possibility of degrading the performance of an Internet application due to contention on resources by other applications hosted in the public cloud. We build a system to detect and eliminate contentions in IaaS environments.

Chapter 2

Foundations

In this chapter, we review the concepts and technologies that contribute to the emergence of cloud computing as a paradigm enabling a rapid scalability for Internet applications. The chapter starts by defining the Internet application, then in Section 2.2, we explain the typical design of a scalable Internet application. In Section 2.3 we explain the service level agreement (SLA) in general, and then we explain what we mean by it in this dissertation. Section 2.4 presents the Internet application hosting development before the cloud computing emergence. Finally, in Section 2.5, we briefly introduce the cloud computing and explain what it offers for Internet applications.

2.1 Internet Applications

An Internet application, sometimes called an Internet service [102] [62], is an application accessed over the Internet. Internet applications can be separated into many types including: online media (e.g., news websites), online information search, online communities (e.g., social websites, blogs, and forums), online communications (e.g., emails and instance messaging), online education, online entertainment (e.g., online gaming), and e-business (e.g., online retailing and online auctions).

Typically, Internet applications are hosted on many servers which simultaneously provide services to a large number of users. The simplicity of updating

and managing Internet applications compared with desktop application is the key reason for Internet applications popularity. Web-based applications, which are accessed by web browsers, are popular type of Internet applications. However, not all Internet applications are web-based. For example, online games and media streaming can be accessed by software client that is downloaded and run at the client side. However, in this dissertation, we concentrate on the web-based Internet applications.

Last decade has witnessed moving many applications that are known to be desktop applications, such as document management and spreadsheets, into the Software as a Service (SaaS) model, which also increases the number of Internet applications. We are increasingly dependent on the Internet applications for both our personal and business affairs.

The parties that are involved in an Internet application hosting, management, and usage can be defined as follows:

- **Internet application's user:** An Internet application's user can interact with the Internet application through a web browser. The interaction may include browsing, submitting text, or uploading files.
- **Internet application's provider:** Typically, an Internet application's provider is a company or organization that runs an Internet application for profit purposes, such as on-line retails, or non-profit purposes, such as Wikipedia. In this dissertation, we assume that the Internet application is hosted in the cloud infrastructure, therefore, we refer to the owner as the cloud customer.
- **Cloud provider:** A cloud provider is the company that offers the Infrastructure and the tools for the cloud customers to host and maintain the performance of their applications in the cloud environment (e.g., Amazon EC2 [12]).

2.2 Multi-tier Systems

A multi-tier architecture provides a flexible, scalable, and modular approach for designing Internet applications. As seen in Figure 2.1, the system is divided

into multiple tiers. Each tier does a certain function. For instance, the web tier consists of web servers that respond to users requests and render the results into a format understandable by clients' browsers. Application tier consists of servers running the business logic of the Internet application. Finally, the database tier stores the incoming data persistently and also offers it to the other tiers on request.

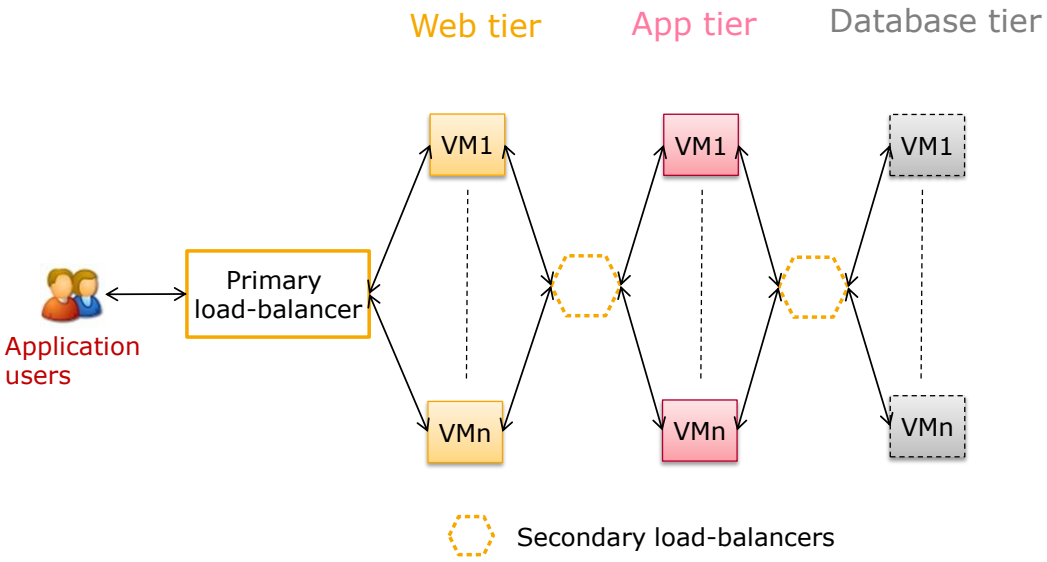


Figure 2.1: A typical multi-tier architecture

The type of the incoming request determines the participating tiers in the request processing. For example, a request for a static page can be handled only by the web tier. On the other hand, a search for items in an online retailer store results in interactions between all tiers.

Typically, each tier is hosted on one or more physical machines (replicas) that are running the same copy of software. To balance the workload among these machines, each tier hosts a load-balancer. The load-balancer redirects the incoming traffic to the proper machine considering the current utilization of each machine and the active sessions. The database tier can also be scaled out into several nodes using Master/Slave architecture, where a Master node receives updating request and the other Slaves handling only the read requests. Splitting the requests according their type can be done by a load-balancer at database tier.

Before the emergence of cloud computing, scaling a multi-tier system is con-

ducted by adding more physical replicas to the overloaded tier, which is a time consuming process. The emergence of cloud computing models, specifically the IaaS model, enables provisioning more VM instances rapidly on-demand. Provisioned instances are charged based-on pay-as-you-go concept, which exploits the chance for reducing the cost.

The traffic of an Internet application is usually session-based, where a session consists of a sequence of requests issued by a client with thinking times in between. The session is often stateful, which makes it necessary to serve a session by the same server during all the session life-time. The load-balancer can guarantee that using "sticky session" (a.k.a. "session affinity"). Nevertheless, "sticky sessions" can result in unbalanced workload on replica servers. Moreover, terminating an instance dynamically will lead to losing the sessions directed to it. This may flush a user's shopping cart or require several logins, as examples. To solve this problem, the application should be modified to store the session context in a shared location accessible by all replicas (e.g., a database or a shared cache such as "memcached"[1]).

2.3 Service Level Agreement (SLA)

The SLA is a contract between a service provider and its customers. The SLA consists of one or more service-level objectives (SLOs). In some contracts, penalties may be agreed upon in case of non-compliance to the SLA. In this dissertation, we deal with two providers: First, the cloud provider. Second, the Internet application provider hosting the application in the cloud environment. The SLA between the cloud provider and the Internet application provider is out of the scope of this dissertation. We consider only the SLA of the Internet application provider, and specifically the response time as a quality of service (QoS) metric.

An example of a SLO in this dissertation is: "The response time of 95% of the requests to an Internet application should be no longer than 100 milliseconds". As seen, the SLO consists of three parts: QoS metric (e.g., response time), the bound (e.g., 100 milliseconds), and a relational operator (e.g., no longer than).

[1] http://www.memcached.org/

2.4 Internet Applications' Hosting

2.4.1 Models of Hosting

Before the cloud emergence, according to the intended level of QoS and security, there were two models of hosting:

1. **Dedicated Hosting:** In the dedicated hosting service, the customer leases one or more servers for hosting an Internet application. The servers are managed and maintained by administrators of the hosting service. However, the client has full control over the servers. The dedicated hosting is often used for Internet applications demanding a high security or QoS.

2. **Shared Hosting:** The shared hosting allows many customers to share same physical servers. However, each customer has a control panel that enables an access to the controlled resources. For example, through the control panel, the customer can upload the files to his own folder in the shared web server. Similarly, the customer can create one or more database instances in the shared database server. The shared hosting is a suitable model for small Internet applications that have a small budget and endure QoS degradation.

   Both models are hosted in data centers that are administrated by highly qualified engineers.

2.4.2 Data Centers

A data center is a facility housing computing, communication, and storage equipments. It is maintained by an organization to handle the core business and operational data of the organization. The size and the location of a data center depend mainly on its function. For example, a data center hosted within a university is relatively small compared with data centers of a big corporation spanning many continents. However, in all cases, a data center should have the following characteristics:

1. The hosted applications in data center should be 24/7 accessible.

2. The equipments of the data center should be reliable and fault-tolerant, which implies replicating the equipments and finding more reliable but probably more expensive hardware.

3. The data center should be physically secured, which requires a restricted access to the facility and equipments. This can be implemented using electronic access cards, video cameras, finger print recognition, etc.

4. The data center should be operated by experts who are equipped with the required tools and monitoring software.

5. The data center should be designed in a way enabling the scalability.

As a matter of fact, the cost for building and managing a data center with a high availability can exceed the budget of small to medium organizations. Therefore, many companies investing in building data centers that offer hosting platforms for customers. Hosting platforms enable emerging companies to focus on the core of their business rather than building and managing a hosting platform [102].

Recently, there is a trend towards moving data centers to geographical spots having plenty of renewable resources [8]. This move can contribute to CO_2 emission reduction. For quick deployment, modular data centers, typically shipped as containers, became available in the market by big providers such Cisco Systems, Sun Microsystems, IBM, HP, and Google [25]. These modular data centers provide efficient consumption of power and are a reliable solution against natural disasters.

2.4.3 Virtualization Technologies

Virtualization is one of the key enabling-technologies for cloud computing. The goal of virtualization is to improve utilization, enable fault-tolerance, and ease administration. The last few years have witnessed rapid migration to virtualized infrastructures. Statistics shows that by the end of 2011, 50% of server applications run on virtual rather than on physical infrastructure [1]. This number

[1] Announced by Paul Maritz, CEO of VMware, at the VMworld 2011 conference

increases to 80% in Australia and New Zealand[1], and we are moving towards infrastructures that are completely virtualized [2].

Definition and Categories: In computing terminology, virtualization means the creation of a virtual, rather than actual, version of resources. The virtualized resources can be accessed by operating systems, applications, devices, or even by human users. Resource virtualization can be categorized into storage, servers, and operating systems virtualization:

1. Storage virtualization: It maps several network storage devices into one single logical resource, which allows transparent provisioning of storage capacity and simplifies data mobility and management.

2. Server virtualization: It partitions physical server (host) resources such as CPU, storage, memory, and I/O devices among many smaller virtual servers (guests). This can be done using a virtual machine monitor (VMM) layer (a.k.a, a hypervisor) running between the operating system of each guest and the hardware, as seen in Figure 2.2 and Figure 2.3.

3. Operating system virtualization: It offers an abstraction of operating system resources using a virtualization layer that does not run directly on the hardware, but on top of an operating system running on the hardware. Examples of such products are Oracle VM VirtualBox, VMware workstation, and Parallels workstation. In our research, we do not consider operating system virtualization while it implies a higher overhead compared with server virtualization. Due to the overhead, operating system virtualization can be more suitable for applications development and testing but not for production environments.

Virtualization techniques: Server virtualization has been developed rapidly during the last few years. The competition for improving virtualization performance is not limited to virtualization technology software developers (e.g., Xen

[1]Announced by Raghu Raman, senior VP & GM, cloud infrastructure & management at VMware

[2]Dr. Pradeep Padala, VMware, http://ga.soe.ucsc.edu/events/event/193

and VMware) but also extended to hardware companies (e.g., Intel and AMD), which implement virtualization's support to the hardware level.

Before discussing the virtualization types, we review the rings concept as a way to manage the access level of operating systems and applications to the computer hardware. In x86 architecture, there are four level of privileges known as Ring 0, 1, 2, and 3. The ring 0 has the highest privilege, where the operating system runs. On the other hand, applications run in the lowest privileges rings (typically, ring 3). Understanding the rings concept is necessary for realizing the difference in performance between the virtualization techniques.

1-Paravirtualization: The Paravirtualization (PV) is a virtualization technique developed by Xen [20], which is an open source project started at Cambridge University laboratories. Xen hypervisor runs a thin privileged domain (i.e., dom0) at Ring 0. It is the only domain that has access to hardware. Therefore, it is used to manage the hypervisor and the other domains. Beside dom0 runs user domains, where the user can host the applications. A user domain (i.e., domU) is a VM running at Ring 1, with a modified operating system (OS) and special drivers. DomU is modified to use special hypercalls. Hypercalls provide a communication channel between the hypervisor and the guest operating system through front-end and back-end drivers, as shown in Figure 2.2.

The hypercalls enable the guest OS to perform at near-native speed, which makes PV technique outperform the other virtualization techniques for some workloads [20]. Nevertheless, PV implies modifying the guest OS, which makes it impossible to host unmodified operating systems, such as Microsoft Windows. Currently, Paravirtualization is used by both Xen and VMware.

2-Full-virtualization without hardware assist: Full-virtualization is developed to allow unmodified OS to run in virtualized environment. In full-virtualization, the hosted OS is decoupled from the underlying hardware. To execute binary transactions by OS guest on hardware, the hypervisor intercepts instructions done in unprivileged level (Ring 1) by unmodified OS and run them in privileged level (i.e., Ring 0). The OS guest is not aware that it is being virtualized while it executes privileged operations as if it is running in Ring 0.

Practically, full-virtualization allows hosting unmodified OS such as Microsoft Windows, but at the same time the online interception and translation of operat-

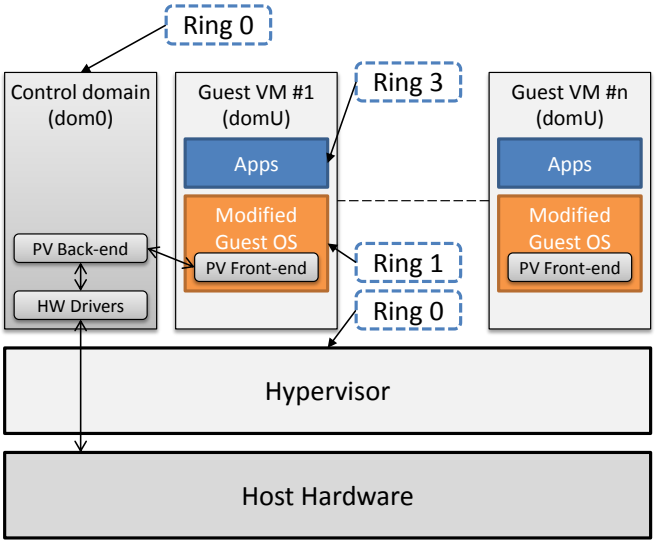


Figure 2.2: A machine running Xen hypervisor. It is an example for a paravirtualized architecture

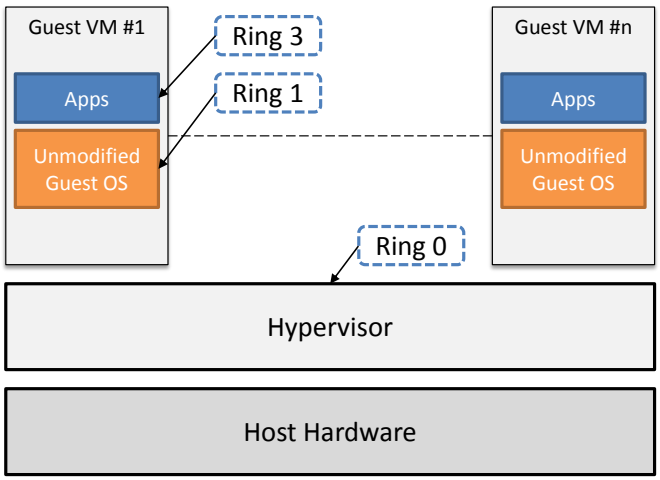


Figure 2.3: Full-virtualization

ing system instructions add more workload to the hypervisor. Consequently, the performance of some applications, especially input/output intensive applications, can be degraded using full-virtualization without hardware assistance. Currently, this type of full-virtualization is supported by VMware, Microsoft, and Parallels[1].

[1]VMware. Understanding full virtualization, paravirtualization, and hardware assist; http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf

3-Full-virtualization with hardware assist: In the mid 2000s, many hardware providers implemented virtualization support at the hardware level. Intel and AMD implemented extensions to their CPUs called Intel VT and AMD-V, respectively. Sun Microsystems (now Oracle) also added similar extensions to their UltraSPARC T-Series processors.

The new processors supporting virtualization introduce an additional privilege mode (Ring 1) below Ring 0, which allows the unmodified guest OS to run in Ring 0. A direct run of guest OS in Ring 0 mitigates the underlying overhead in full-virtualization technique. Full-virtualization with hardware assist is supported by VMware, Xen, Microsoft (Microsoft Hyper-V), and Parallels.

2.4.4 Virtualized Data Centers

A virtualized data center is a data center adopting virtualization technology. Statistics by International Data Corporation (IDC) showed that the number of servers in USA alone jumped from 2.6 million in 1997 to 11.8 million in 2007. Most of those servers run at 15% or less of the total capacity of the physical machine [50]. Virtualized data centers emergence is a legitimate outcome of the increase in the number of servers and the need for more efficient resource management solutions. However, the number of the VM instances increases proportionally to the number of the physical servers. As the number of VM instances increases, efficient management solutions become a necessity for virtualized infrastructure providers' success.

As a matter of fact, virtualization technologies add many advantages to data centers. We summarize them as follows:

1. Compatibility: in virtualized data centers, different applications with different platforms and libraries can be hosted on one physical host, each application runs in isolated VMs.

2. Easy management: virtualization technology come with a number of tools that simplify monitoring and management of virtualized resources. Moreover, many of the virtualization technology providers have developed APIs that allow virtualization technology users to build their controllers and automate resource management.

3. Increase revenue: the possibility of consolidating several servers (i.e., VMs) into one physical host can also raise server utilization rates to as high as 80 percent[1]. Consolidation enables data center providers (i.e., hosting platforms) to pack the virtual servers with low workload into a lower number of physical servers. In consequence, the power consumption by both the physical servers and the cooling systems can be reduced.

4. Fault tolerance: A failed machine can be recovered quickly form a VM image or from a snapshot.

5. Adaptability: The allocated resources to a VM can be dynamically changed to cope with the demand. For example, if the system administrator recognizes that a database VM requires more memory, he/she can reconfigure the VM's memory settings without replacing the physical server. Moreover, if a physical server is unable to satisfy a VM's demand, the VM can be migrated to another server with adequate resources.

While virtualization offers many benefits for data centers, security and performance are the main challenges in virtualized environments. In a former study, we investigated the potential challenges of using virtualization [43]. Moreover, we examined the security of Xen 3.x in [58] and offered suggestions for avoiding possible threats. Nevertheless, in this dissertation, we focus on application performance.

2.5 Cloud Computing

Cloud computing is a natural outcome of the rapid development of many foundation technologies especially virtualization, networking, storage, multi-tenant architectures, and distributed systems management [28]. The quick adoption of the cloud is driven by the invented service consumption models and delivery models that support a rapid business development [50]. However, cloud computing lasted as a buzzword for the last few years. The National Institute of Standards and Technology's (NIST), after 15 drafts, published their final definition

[1] http://www-03.ibm.com/systems/virtualization/news/view/vdc.html

of cloud computing as follows: "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models" [79].

The five essential characteristics of the cloud, according to NIST, are as follows:

1. **On-demand self-service:** On-demand self-service is an essential characteristic of cloud computing. It allows cloud computing users to provision computing power, storage, and networks in a simple and flexible way. For this purpose, cloud providers offer dashboards that authorize the users to control their resource provisioning. Also, resources can be managed through Applications Interfaces (APIs) that allows automating resources management without human interaction. Utility computing and pay-per-use subscription models are associated with on-demand characteristic. Rather than paying for owning an infrastructure, users are billed only for the period of time they consume resources in an infrastructure owned and maintained by a provider.

2. **Broad network access:** Provisioning resources in the cloud to host applications implies accessing these applications through the Internet by a vast number of clients. To offer a high QoS for all clients, cloud providers should have standardized network connections that have large enough capacity coping with the increase in the number of hosted applications and clients.

3. **Resource pooling:** A resource pool is a collection of unassigned resources (e.g., CPU, Memory and Network) that are available to different users and applications. The resources are assigned to the cloud user upon request. The exact location of the resource is not revealed to the customer. However, the customer can specify the location at a higher level (i.e., country and data center).

4. **Rapid elasticity:** Elasticity is the ability of scaling up and down resources to cope with the demand. Elasticity is very important for nowadays Internet applications, where predicting the coming workload became more difficult due to the social networks emergence that increases the probability of flash crowds. The size of the resource pools in public clouds is unlimited theoretically. However, cloud customers should build their applications in a proper way to benefit from the cloud elasticity.

5. **Measured service:** From the provider side, a detailed monitoring for cloud resources is crucial for capacity planning, resource optimization, and billing. From customers side, monitoring provisioned resources is crucial for performance management and resource provisioning optimization. In all cases, the monitoring should be light-weight to avoid influencing the hosted applications performance.

2.5.1 Cloud Computing Delivery Models

Cloud computing services are classified into three abstracted delivery models (layers) [79] [28]: Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). These abstracted delivery models can be composed from services of the underlying layers [28]. For example, PaaS layer can be a platform hosted on several VMs running in IaaS layer. However, as seen in Figure 2.4, Lamia et al. [118] shows that the two highest levels in the cloud stack (i.e., SaaS and PaaS layers) can bypass the IaaS for better performance. Nevertheless, this comes at the cost of simplicity and development efforts.

The details of the basic three service models are as the following:

- **Software as a Service (SaaS):** The SaaS is a delivery model for many applications ranging from very specialist business applications, such as invoicing, customer relationship management (CRM), and human resource management (HRM) to web-based applications that can be accessible by any Internet' users, such as web-based email and web-based office suites. The principal point for a company to move to SaaS applications is to concentrate more on the business design than IT support by outsourcing hardware

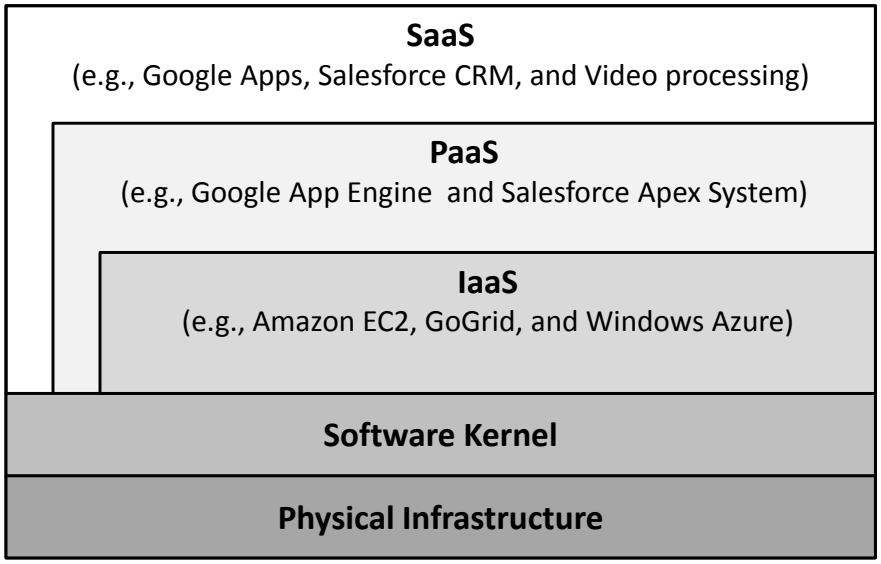


Figure 2.4: Cloud computing delivery models' stack [118]

and software maintenance and support to the SaaS provider. Nevertheless, SaaS providers offer their customers dashboards that allow managing and customizing the SaaS applications to suit their needs.

- **Platform as a Service (PaaS)** The PaaS is a delivery model for many platforms such as programming languages execution environments, databases as a service, and web hosting platforms. PaaS customers need not to worry about the complexity of managing and operating the underlying hardware and software layers. The scalability and performance management is the provider's job. However, some PaaS providers ask the customers to configure some parameters that help maintaining the performance. For example, Google app engine ask the customer to select the front-end instance class from three different classes of instances, each class has different CPU frequency and Memory size. Moreover, it enables the customer to predict the number of required instances. This number helps the provider to prepare VM images in advance and avoid the delay of copying the code when a new instance is needed.

- **Infrastructure as a Service (IaaS)** The computation power in IaaS model is often delivered as virtual machines running in the provider cloud

infrastructure. According to the hosted application and the expected workload, the customer can decide about the operating system and the virtual machine capacity specifications, respectively. Additional resources are offered to IaaS customers such as virtual machine images library, monitoring services, block and file-based storage, IP addresses, virtual network, load-balancers, and firewalls [17]. In IaaS models, the customers are free to install the platform and the software, but on the other hand they are responsible for patching, securing, updating, and maintaining the performance of the operating system, the platform, and the application.

As seen in Figure 2.4, the three models are built on top of software layer called "Software Kernel" [118]. This software layer is necessary to manage the physical servers and components at physical infrastructure layer; it can be for example a hypervisor, clustering middleware, or scalability controllers that scale applications at PaaS and SaaS layers.

2.5.2 Cloud Computing Deployment Models

According to the organization size, location, and the purpose of operating computational resources, cloud services can be deployed in different ways, namely private, community, public, or hybrid.

- **Private cloud:** A private cloud typically deployed to be used by a single organization. However, it can be hosted on-premises (i.e., within the organization itself) or by a third party. The capacity of the private cloud is known contrary to the public cloud capacity. Nevertheless, private cloud offers more privacy, control on resources, and guaranteed performance.

- **Community cloud:** A community cloud typically serves organizations that have the same interest and concerns. For example, different research or educational organizations can federate their private clouds to have one community cloud that is available to every organization in the community. The cloud can be managed and operated by one or more of the organizations in the community.

- **Public cloud:** A public cloud is deployed to be used by public users. It is typically owned and operated by business, academic, or governmental organization. It is hosted on the premises of the provider but can span many geographical locations. Public clouds owned by business organizations offer an attractive environment for the emerging business.
- **Hybrid cloud:** The hybrid model of cloud deployment helps the organizations that already have their own infrastructure (e.g., private cloud) but approach occasional spikes in demand. Such organizations can keep using their secure and full controlled infrastructure, and in the same time benefit from pay-per-use concept of the public cloud.

2.5.3 Cloud Computing Advantages

In this section we list some of the advantages of hosting an Internet application in the cloud.

- **Reduction of the cost:** Customers can avoid spending large amounts of their budget on purchasing and installing an IT infrastructure by moving to the cloud. Administration cost also can be reduced while the cloud provider is responsible for managing, patching, and upgrading both the software and hardware systems. Cloud computing paradigm allows customers to focus on business management and development instead of human resource management and training. From the provider side, the huge number of customers increases the possibility of consolidating much workload to less physical hardware. The consolidation reduces the expenses of the provider and allows it to offer lower price services.
- **Scalability on-demand:** Scalability, a highly valuable characteristic in cloud environments, enables customers to react quickly to IT needs. Moreover, the on-demand model, in which the costs are based on the actual consumption, allows the customers to benefit from the scalability while optimizing the cost.
- **Disaster recovery/backup:** Public clouds are developed to serve global business. Therefore, data centers of a public cloud provider are typically

distributed to several geographical locations, regions, and sometimes zones [12] to isolate the failure propagation. The virtualization of the data centers allows recovering the failure of a VM instance by initiating a new instance from the machine image. In case of the physical host failure, the VMs can be migrated to another physical host. Additionally, the cloud storage services (e.g. Amazon S3) are replicated automatically for more durability and reliability, with the option of reducing the number of replicas and reducing the cost.

2.5.4 Cloud Computing Issues

In spite of the fact that many advantages can be gained by moving to cloud environments, many concerns can influence the business owners' decision about moving to the cloud. The main issues are summarized as the following:

- **Privacy:** Privacy is a big concern in cloud environment while cloud systems process and store valuable data from different individuals on same physical servers. Much research [106] [59] [9] is conducted to assure users that their data will not be released or accessed even by the service provider under any circumstances. Encryption can protect the data but it is always a trade-off between security and performance [27].

- **Security:** Security of cloud computing services is an issue that may be delaying its adoption. In a former study [43], we showed that security, as an issue, is inherited from the foundation technologies of the cloud such as networking, virtualization, and web services. However, the efficiency of traditional protection mechanisms should be reconsidered when applied to cloud environments. Improving virtualization isolation, auditing, implementing public key infrastructure (PKI), and maintaining cloud software security in general [19] are necessary for a secure cloud environment.

- **Data Lock-In:** Hosting as system in the cloud makes it vulnerable to lock-in problem, which means that the system cannot be easily moved to another provider. This makes customers exposed to many problems including the following: (i)Price increase. (ii)Reliability problems. (iii) Or even

the probability of the provider's absence by going out of the business [19]. The lock-in problem is very severe in SaaS and PaaS model. In IaaS model it has less severity. However, customers who plan to provision resources from different providers should implement application that is able deal with the APIs of the different providers. Standardizing the APIs can be the solution for the lock-in problem. However, the challenge is to convince the big cloud providers to standardize their interfaces.

- **Performance Isolation:** Virtualization, a foundation technology for the cloud computing, enables consolidating many VMs machine into same physical host. Each VM can be owned by a different organization and run different workload. Each VM has its share of the physical host resources (e.g., CPU, Memory, I/O). The hypervisor multiplexes physical resources between VMs to isolate their performance. However, shared resources isolation (e.g., I/O) is a challenging problem demanding much research [64] [78] [70]. For example, Armbrust et al. [19] measured the average disk write rate of 75 instances in Amazon EC2 as 52 Mbytes/s with a standard deviation of about 16%, which means that the I/O performance is exposed to influence by other customers' workload in the cloud.

Chapter 3

Related Work

Typically, managing Internet applications' performance focuses on the overloading times of Internet applications. Provisioning plenty of resources is straightforward way to guarantee applications performance at overloading times. Nevertheless, business is always adhered to a limited budget. A successful online business sticks to its budget without losing the clients satisfaction or/and paying penalties. Toward this goal, much research is conducted during the last years. This chapter presents the most related work and concentrates on the techniques that benefit from the emergence of cloud computing models (specifically the IaaS model). For example, the service degradation, as a performance management technique, does not get a big attention nowadays due to the possibility of provisioning resources rapidly, with relatively a low price, during overloading time.

We classify the related work according to a taxonomy proposed by [51]. The first classification of the related work depends on the techniques used for managing resource dynamically. The second classification depends on the mechanisms used to drive each resource management technique. The related work is illustrated in Table 3.1 considering both classifications.

Figure 3.1 shows performance management techniques, at the left side, in addition to the mechanisms driving performance management, on the right side. The figure also highlights the techniques and mechanisms that we depend on in this dissertation.

Before studying the dynamic resource management techniques, we start in Section 3.1 with an overview of the traditional performance management tech-

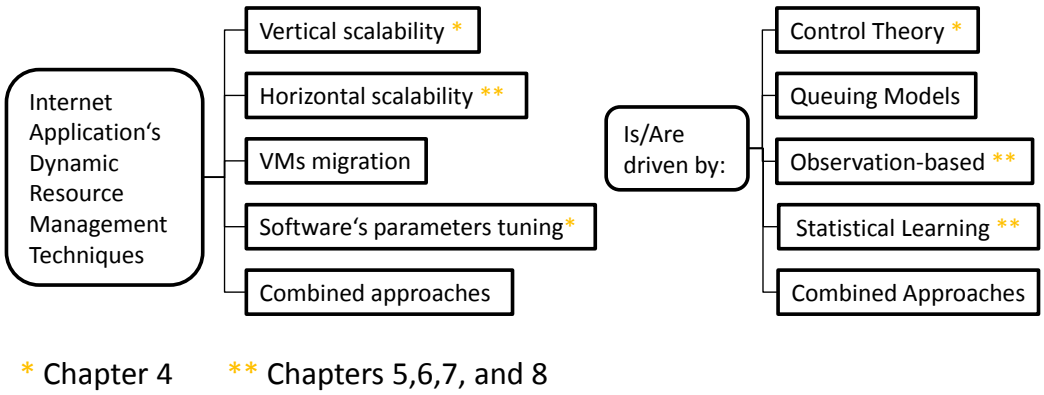


Figure 3.1: Performance management techniques for Internet applications and the driving mechanisms

niques used for many years before the emergence of the on-demand resource provisioning. In Section 3.2, we study the development of dynamic resource management techniques with the emergence of virtualization technologies and cloud computing. In Section 3.3, we review the approaches developed to mitigate the impact of the interference in public IaaS environments. In Section 3.4, we overview some of the commercial performance management tools that are used nowadays. Finally, in Section 3.5, we summarize the chapter.

3.1 Traditional Performance Management Techniques

For many years, and before the emergence of virtualization technologies and cloud computing, the admission control, the service differentiation, the service degradation, and sometimes combinations of them have been practical techniques for maintaining Internet applications performance during overloading periods of time. Admission control technique maintains the availability and performance of a server by controlling the number of admitted requests to the server. At specific threshold of utilization, any additional requests are dropped to keep the server utilization within the threshold. Typically, computing systems have different performance thresholds that are importantly measured by the system administrator for maintaining the system performance, as we explain in Section 6.1. In

contrary to admission control, service differentiation technique differentiates customers into classes and provides different QoS for each class. For example, at overloading time, an online retailer can give more priority (i.e., dedicate more resources or decline other classes) for buying request over browsing requests. Actually, admission control is a special case of the service differentiation technique [51]. Therefore, it is usually combined with service differentiation techniques [110] [61] [121] [109] to prevent overloading servers and offer different QoS for clients [51].

In spite of the emergence of the cloud computing paradigm, which promises a virtually limitless scalability, the traditional techniques are still needed for the following reasons: First, resources in the cloud are limited per an account (e.g., 20 EC2 instances limit associated with each Amazon AWS account). However, the user can ask for raising this limit. Second, if the system scalability is not limited by the resources, it will be limited by a specific budget [116]. Finally, raising the resources' limit does not protect the Internet application from the denial of service attack (DoS). It also exposes the company budget to an unexpected increase. To protect the Internet application from attacks that target the service availability, performance, or budget, GoGrid load-balancer [1] for example limits the number of accepted connections per a client. Liu et al. [72] used a queuing model predictor and an online adaptive feedback loop that maintains the response time within a specific limit by enforcing admission control of the incoming requests. Other researchers [32] [67] use profit-based policies that consider the trade-off between the QoS and the profit.

3.2 Dynamic Resource Management Techniques and the Driving Mechanisms

Dynamic provisioning of resources (i.e., allocation and deallocation) to cope with workload had much interest especially after the widely usage of virtualized data centers and the cloud. Significant prior research has been sought to map QoS requirements (e.g., response time) into low level resources (e.g., CPU, Memory, and I/O). Response time is the most considered performance metric in many

studies [47] [114] [105] [56], since it is the metric that can easily be noticed by the end user. Towards managing resources dynamically, the used techniques are as follows:

3.2.1 Vertical Scalability

The advance of the virtualization technology allows system administrators to redistribute resources among the hosted VMs on the same physical host dynamically. This allows the administrator to scale overloaded VM vertically on-the-fly, without interrupting the service. The vertical scalability depends on the probability of that the VMs sharing the same host do not consume all the allocated resources. Examples of the resources that are scaled dynamically is the CPU capacity (i.e., frequency) [47] [114] [38] [95] [105]; number of cores [40] [117]; and memory capacity depending on the balloon driver [56] [95] [38]. Generally, the limitations of vertical scalability are: (i) It requires the cloud provider involvement in the resources redistribution, which is a complex management task in the large-scale shared infrastructure. (ii) It can lead to conflicts on resources that cannot be solved instantly. For example, an additional memory given to a VM, can't be recovered in a small time as same as the CPU capacity can be recovered.

Control theory has been widely used for performance management of Internet applications. The first step towards implementing a feedback controller (a.k.a. loop-back controller) is the selection of the control input(s) and the system output(s) [47]. Typically, control inputs are resource allocation (e.g., memory and CPU allocation), while the system outputs are resource utilization (e.g., memory or CPU utilization). The second and most importantly step is to distinguish between the operational regions. For example, Wang et al. [114] showed that the relation between CPU utilization, as output, and the CPU entitlement, as input, behaves differently in overloaded regions than it behaves in the under-loaded regions (i.e., a bimodal behavior). The third step is determining the reference point for the controller. Gandhi et al. [47] suggested determining CPU and memory reference allocation by the administrator. Nevertheless, Zhu et al. [123] showed that maintaining the response time within a specific limit requires adapting the reference point dynamically. Therefore, they developed a nested loop feedback

controller that calculates the reference CPU utilization dynamically considering the workload variation. The controller designed by Wang et al. [114] is used by Padala et al. [88] to keep the CPU utilization of individual VM instances around 80%. Moreover, the controller designed by [123] is integrated with a memory feedback controller in [56] to keep the memory allocation within a reference point (i.e., 90%). In our work [38], we integrate a CPU controller [123], a memory controller [123], with a proposed heuristic application controller. The application controller optimizes the Apache server parameter *MaxClients* with each vertical scalability of resources, as we explain in Chapter 4.

Based on synthesis queuing model and loop-back controllers, Wang et al. [113] developed a dynamic resource management system that maintains the application response time by automatically determining the utilization targets and the size of the VM. Nevertheless, the authors assume hosting a VM in one physical host and do not consider the possible conflicts on resources by the other co-located VMs.

Using the observed utilization of the CPU, we built in [40] a scalable architecture that depends on static scaling thresholds. However, instead of scaling per a VM instance, as coarse-grain scaling unit, we scale up by increasing the number of cores (on-the-fly) of the overloaded VM dynamically. The vertical scalability allowed the system to avoid the overhead of initializing more VMs horizontally. Yazdanov et al.[117] investigated improving vertical scalability by integrating prioritizing technique and employing proactive scaling. Nevertheless, our empirical observations showed that proactive scaling does not improve vertical scalability, while it is already coping fast with workload variation. Han et al. [54] and Shen et al. [95] integrated the vertical scaling at the resource level (CPU and Memory allocation), with the horizontal scaling at the VM level. Moreover, [54] supports VMs migration to solve the resource conflicts in case of the lack of adequate resources in the physical host.

Statistical Learning means building models describing the application performance. These models can be built online or offline to guide the resource management. By online statistical learning, Padala et al. [87] developed a linear model that captures the relation between resources allocation and performance. Additionally, a multiple input multiple output (MIMO) controller is integrated to

calculate the right allocation that maintains the performance. Padala et al. [87] considered controlling both the CPU and Disk utilization by the co-located VMs in the same physical host to control the performance of the multi-tier system.

3.2.2 Horizontal Scalability

Horizontal scalability provides coarse-grain scalability to the Internet application. As seen in Figure 2.1, the multi-tier application scales out by provisioning more VM instances in the overloaded tier. The problems that are approached by the researchers implementing the horizontal scalability are as follows: (i) Determining the trigger for a scale out or down. (ii) Finding the optimal number of VMs in each tier that minimizes the cost and maintains the performance simultaneously.

Using feedback controllers to guide a horizontal scalability of applications exposes the controlled system to the oscillation around the reference utilization [69]. The oscillation is due to coarse-grain scaling. For example, considering the value 80% as a reference CPU utilization causes a tier with one VM to scale out to two VMs when the utilization goes over 80% (e.g. 90%). In the next control period, the controller will measure the average utilization as 45%, which is much lower than the reference point. A scale down process is initiated followed by another scale out. This oscillatory behavior can continue indefinitely. To address this problem, Lim et al. [69] propose an initial range of the CPU utilization. Depending on the measured CPU utilization, a different targeted utilization is calculated. Lim et al. [68] extended their previous work in [69] by implementing their approach to storage tier to maintain a specified bandwidth.

Urgaonkar et al. [103] use queuing theory to model multi-tier Internet applications analytically. Their approach also determines the number of needed servers in each tier. Urgaonkar et al. [105] extended their previous work [103] by proposing dormant VMs (i.e., VMs assigned minimal server resources) concept. The dormant VMs are prepared with the required software and scaled rapidly when the workload increase, which eliminates the overhead of resource provisioning. A multi-tier system is modeled in [103] and [104] as a closed network of queues. However, because the authors consider a single queue per a tier, their system is able to capture a single resource bottleneck at a time (e.g., CPU utilization or

network bandwidth). To go over this limitation, Stewart, et al. [98] modeled a server as a network of queues representing multiple resources (i.e., CPU, Network, and Disk). This enabled their technique to predict different resources bottlenecks. Nevertheless, modeling the multi-tier system as an open network, as in [98] [23] [32], neglects the impact of the user thinking time, which is an unrealistic assumption for Internet applications [71] [103] [104]. In general, implementing the scalability to Internet applications using queuing theory is complex. The complexity lies in the need for estimating many model parameters such as the arrival rate and the service time of requests at each tier, and other parameters related to congestion effects [98]. This questions the efficiency of implementing queuing theory mechanisms in the production environment.

Many observation-based mechanisms are developed to drive horizontal scalability and maintain multi-tier systems performance. Typically, researchers consider one or combination of metrics, such as the profit [94] [75] [67], the response time [57] [75] [54] [39] [67], and the throughput [75] [67]. In fact, the horizontal scalability driven by online observation is highly suitable for production environments. Therefore, we see a big IaaS provider such as Amazon EC2 [12] employing simple scaling policies with static thresholds to drive a reactive scalability. However, Han et al. [54] proposed combining the horizontal with the vertical scalability to scale application rapidly with fine-grain scaling units. Nevertheless, vertical scalability is not a mechanism provided by public IaaS providers.

Malkowski et al. [75] propose a multi-model controller containing several models that cooperate to optimize the profit and maintain the performance. The multi-model controller interacts with a repository, which contains records of previously observed application system behavior and the configuration. The goal of the repository is to find the configuration that leads to a better performance. Iqbal et al. [57] considered the response time as a metric to drive horizontal scalability. Our practical experience shows that the response time in public IaaS is not a reliable metric for resource management [39]. A similar observation by [74] shows that the response time of a multi-tier system can spike to three times the accepted response time's value despite a stable throughput.

In [39], we implemented a scalable system depends on modeling the application performance. The approach can predict the required number of VMs in each tier.

Moreover, it is able to detect and avoid the performance interference in public IaaS environments. Our approach is a combination of both the statistical learning and the observation-based mechanisms.

3.2.3 VMs Live Migration

Much research has exploited the VM live migration mechanism for coping with dynamic workload fluctuations, as well as providing scalability and load balancing techniques. VMs migration is also used to optimize the power consumption within a data center by packing VMs into less physical hosts and terminating the idle servers. Towards handling dynamic workload changes, many researchers [115] [94] [95] [63] implemented approaches considering migrating overloaded VMs to underutilized hosts. The efficiency of the migration depends on selecting the source and destination host. Typically, migration planning is solved as a bin-packing problem, which is NP-hard problem [63].

Khanna et al. [63] developed a heuristic algorithm for VMs placement. They consider both migration cost and capacity residues as parameters for their algorithm. Beloglazov et al. [21] designed an energy-aware system to help the cloud provider to optimize the power consumption by the data center, while maintaining the performance. VMs migration is used by Sharma et al. [94] and Shen et al. [95] to solve conflict on resources caused by vertical scalability. Das et al. [37] focus on live migration of databases. They developed a technique to minimize the migration's impact on the database performance during migration.

Wood et al. [115] developed statistical learning mechanisms to guide VMs migration. Nevertheless, the live migrating of a VM consumes I/O and CPU and network resources which might contribute in performance degradation of other VMs. Moreover, the migrating of applications that have long-running in-memory state or frequently updated data, such as database, might take too long time causing SLO violations during migration. Additionally, security restrictions might increase overhead during migration process [86].

3.2.4 Software's Parameters Tuning

Choosing the correct parameters for software is crucial for maintaining performance. However, it is not a straightforward job, while it depends on both resource allocation and workload [47]. The workload is typically variable. On the other hand, the vertical scalability exposes the software to different values of resource allocation. If software was unaware of the increase of resources, it may not benefit from this increase. For example, *MaxClients* is a parameter for the Apache web server software. It controls the number of processes that can run concurrently to serve client requests. An Apache sever with a very low *MaxClients* will not benefit from increasing CPU or Memory allocation. Nevertheless, we observed that *MaxClients* has an optimal value for each resource allocation that maintains a higher throughput and better performance [38].

Using a loop-back controller, Gandhi et al. [47] developed a system that tunes two parameters of Apache web server: *MaxClients* and *KeepAliveTimeout*. The system's outputs are the CPU and Memory utilization. The reference points of the CPU and memory utilization are determined by the administrator. However, these reference points are inconsistent when the system experiences variant rates of workload [114] [123]. Chess et al. [34] developed an agent-based solution to automate system tuning; the agents do both controller design and feedback control. However, their system converges slowly, which impedes the system of coping with sudden changes of the workload. Liu et al. [73] compared three types of controllers to optimize *MaxClients* of Apache web server online: (i) an optimizer based on the Newton's method, (ii) a fuzzy controller, and (iii) a heuristic controller. The optimizer based on Newton's method suffered inconsistency with the highly variable data. The fuzzy controller is more robust but converges slowly. Finally, the heuristic controller works well under specific conditions. In Chapter 4, we develop an online heuristic application controller that searches for the optimal value of *MaxClients* by exploiting the gathered measurements during the run time of the system.

Using queuing theory along with optimization techniques, Jung et al. [60] presented off-line techniques to predict system behavior and automatically generate optimal system configurations. The result is a set of rules that can be

inspected by human system administrators and used directly with a rule-based system management engines.

Resource management techniques	Driving mechanisms	Publications
Vertical scalability	Control theory	[114] [123] [88] [113] [56] **[38]**
	Queuing models	[113]
	Observation-based	**[40]** [117] [54] [95]
	Statistical learning	[87]
Horizontal scalability	Control theory	[69] [68]
	Queuing models	[103] [105] [23] [32]
	Observation-based	[57] **[39]** [67] [54] [94] [75] [12]
	Statistical learning	[75] **[39]**
VMs' migration	Control theory	
	Queuing models	
	Observation-based	[94] [95] [63] [21]
	Statistical learning	[115] [95]
Software's parameters tuning	Control theory	[47] [34] [73]
	Queuing models	[73] [60]
	Observation-based	**[38]**
	Statistical learning	

Table 3.1: The state of the art towards the dynamic scalability of Internet applications. The bold references are our papers published in the course of this dissertation. The appearance of an approach in multiple row means that it employs more than a scalability technique or/and driving mechanism.

In Table 3.1, we summarize the most related approaches to our research. The approaches considering several resource management techniques or employing different management mechanisms can be shown multiple times in different rows of the table.

3.3 Mitigating Performance Interference in IaaS Environments

Much research has been done towards effective performance isolation among VMs in the virtualized environment [70] [78] [64] [53], which is a demanding problem for public IaaS environments. Nevertheless, the applications still exposed to the interference by other co-located applications [89] [66] [108] [65]. High-level solutions [24] [100] [44] [39] are developed to distinguish the abnormal VM instances depending on their behavior. These solutions are practical in public environments, where the customer can abandon an abnormal VM instance and replace it with new and healthy one. Nevertheless, the proposed approach by Bodik et al. [24] requires a prior knowledge about the anomalies (i.e., signatures). This fact limits the approach to a set of pre-known anomalies. Tan et al. [100] and Dean et al. [44] designed scalable approaches depending on a VM self-learning (i.e., decentralized learning). The learning allows a VM to distinguish between normal and abnormal states. The approach by Dean et al. [44] outperforms the one in [100] because it does not require training with samples covering both normal and abnormal states. Nevertheless, the additional workload by training process inside the VM can influence the main application performance. To avoid influencing the performance of the running application in the VM, we build performance models of VM instances depending on external monitoring metrics [39]. Our approach is explained in details in Chapter 8.

3.4 Commercial Solutions

Many IaaS providers, such as Amazon EC2 [12], Windows Azure [81], Rackspace [3], and GoGrid [1], offer application programming interfaces (APIs) that allow customers to build their own controllers and automate the scalability in the cloud.

However, a cloud customer can hire a third-party provider who automates the cloud resource management and scaling. Typically, the resource management providers host the management services on their servers and use APIs of each provider remotely, which allows them to manage resources in different IaaS

providers. Customers of third-party scalability providers have web interfaces that allow them to monitor and setup the scalability at different IaaS providers. The providers charge customers monthly for their services. Examples of such providers are RightScale [4] and Scalr [97]. However, Scalr has an open source version that can be hosted freely on customers' servers. Nevertheless, monitoring and controlling resources remotely through the Internet can imply an overhead.

On the other hand, Amazon EC2 [12] provides scalability components and services that can be consumed as same as the other computing and storage services. Integrating scalability services into IaaS environment increases the reliability and reduces the control overhead. On the other hand, it limits the clients to one IaaS provider. In Chapter 5, we explain in details the scalability in Amazon EC2, as a provider enables the scalability to a large number of Internet applications that are already in production environment.

3.5 Summary

In this chapter, we reviewed the approaches that are developed during the last years for managing the scalability and performance of multi-tier applications. We started with the approaches that have been used before the emergence of virtualization technology and cloud computing. Afterwards, we discussed the approaches that exploit the virtualization technology and the IaaS model to scale Internet applications rapidly. Vertical scaling, as a management technique, got the researchers intention due to its rapid response to workload change. Nevertheless, implementing it in big production environments, such as public IaaS environments, is very complex. Similarly, VMs' migration can be used for distributing the workload and maintaining the application's performance. Nevertheless, both VM migration and vertical scalability approaches imply the provider involvement in complex management tasks. On the other hand, in spite of the overhead of provisioning new resources, the horizontal scalability is the widely used resource management technique in production environments nowadays.

Chapter 4

On-premise Hosting for Internet Applications

Virtualization technology has been adopted by many present data centers due to its features, which include workload consolidation, live migration, and on-the-fly virtual machine scaling. Workload consolidation allows hosting multiple VMs with different workload into a single physical host. Using live migration allows moving VMs between physical servers in data centers for balancing workload without interrupting the service. Workload consolidation increases resource utilization which is an opportunity for increasing data centers' revenue. Moreover, it exploits the possibility of maintaining Internet applications performance by redistributing virtualized resources dynamically (i.e., scale vertically) among VMs depending on the workload intensity.

In this chapter, we investigate managing an Internet application scalability and performance using vertical scalability driven by control theory. Currently, the IaaS environment provides a horizontal scalability for Internet applications. Nevertheless, horizontal scalability has several disadvantages: First, it is coarse-grain scaling where a VM is the scale-step size unit. Second, provisioning resources horizontally implies an overhead. Third, scaling some tiers horizontally such as database tier is a complex process. On the other hand, vertical scalability enables a fast and fine-grained scalability (e.g., CPU percentage and Memory in MB) for all tiers of Internet applications.

Nevertheless, scaling resources vertically requires control over the virtual machine manager (VMM) (a.k.a, hypervisor), which is not available by the public IaaS providers. Therefore, in this chapter we consider an on-premise virtualized data center that is fully controlled by the Internet application's provider. Virtualization offers APIs that enable managing and distributing resources among the hosted VMs in a data center. The VMs running an Internet application receive variable workload. We exploit the probability of co-locating underloaded VM with other overloaded VMs in the same physical host to maintain our SLO by redistributing resources between VMs according to the actual demand. However, we study cases when applications compete on resources, and show our approach ability to mitigate the impact of the competition on applications performance.

We implement our approach into Xen environment and consider scaling the web tier which runs Apache web server. The scalability and performance management is driven by a CPU [114] and a Memory [56] feedback controllers. In parallel to them, an application controller is developed to tune the Apache web server's parameters dynamically. Our SLO in this chapter is to keep the response time of the web requests less than a specified threshold. Nevertheless, our architecture can be extended for applications that have tunable parameters such as database applications.

The key contributions of this part of research are as follows:

- We analyzed Apache application performance under different configuration and different CPU and Memory allocation values.
- We developed a dynamic application controller for Apache application to maintain the application performance.
- We built CPU and Memory controllers based on [56], then have integrated the three controllers CPU, Memory, and application optimization controllers for a rapidly scalable application.
- Finally, the proposed architecture was evaluated with extensive experiments on several synthetic workload and experimental setups.

The results show that our proposed architecture can maintain the performance of the controlled application in terms of throughput and response time.

The chapter is organized as follows: Section 4.1 is an analysis of systems and concepts that drive our research. In Section 4.2, we explain our proposed system architecture. In Section 4.3, we describe the experimental setup and analyze the results.

4.1 Overview

In this section, we give an overview of the investigated systems and concepts during our research; we start with a detailed study of Apache web server [92], then we discuss the complexity of using feedback control systems in computing systems, and finally we explain the concerns that accompany using vertical scalability to cope rapidly with the workload variation.

4.1.1 Apache Web Server

Apache web server [92] is one of the most popular web servers. Currently, it powers over 63% of sites on the World Wide Web[1]. Apache [92], is structured as a pool of workers processes that handle HTTP requests. Currently, Apache supports two modes, workers and prefork modes. In our experiments we use Apache with prefork mode to handle dynamic requests (e.g., php pages), because it is more reliable for high rates of traffic. In prefork mode, requests enter the TCP Accept Queue where they wait for a worker. A worker processes a single request to completion before accepting a new request. The number of worker processes is limited by *MaxClients* parameter.

Figure 4.1 shows the result of an experiment in which the Apache web server was tested with different settings of Memory, traffic rate, and *MaxClients*. By monitoring the throughput, we noticed that there was a value of *MaxClients* (e.g. 75) that gives the highest throughput (e.g., 450 req/sec) for specific Memory settings (e.g., 512MB). Before this value there were no enough workers to handle requests, and after this value, performance degraded because of one of the following problems: The CPU spends much time switching between many processes, or the Memory is full so the paging to the hard disk consumes most of the CPU time.

[1] http://loadstorm.com/2011/web-performance-optimization-part-5-apache-server

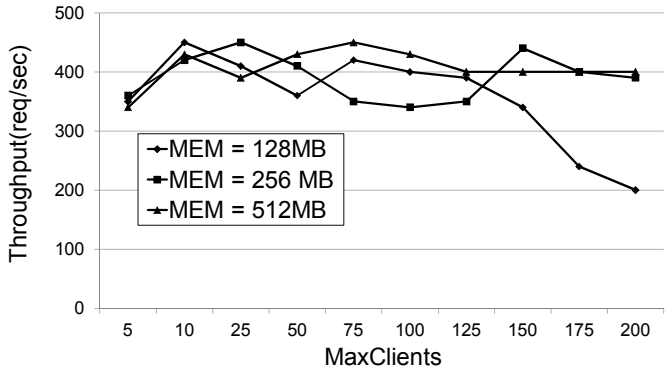


Figure 4.1: Throughput v.s. MaxClients under different hardware settings

To find the optimal value for *MaxClients* dynamically, we developed a heuristic Apache controller running in parallel with the CPU and Memory controllers.

4.1.2 Feedback Control of Computing Systems

Controllers are mainly designed for three purposes [55]: First, regulating an output to be equal or near to a reference input; for example, maintaining the Memory utilization to 90%. Second, disturbance rejection which means that if the CPU is regulated to be 70% utilized, then it must not affected by any other running applications like backup or virus scanning. Third, optimization which can be translated in our system as finding the best value of *MaxClients* that optimize Apache server performance. In terms of the feedback controllers, SLO enforcement often becomes a regulation problem where SLO metric is the measured output, and SLO bound is the reference input.

The choice of the control objective typically depends on the application. Moreover, the same system may have multiple controllers with different SLOs. Unfortunately, identifying Input-output models for computing systems is complex because of the absence of the first-principle models [122]. As a replacement, many researchers [115] [87] [56] [114] considered the black-box approach where the relation between the input and output is inferred empirically. According to Zhu et al. [122], to build a feedback controller able to adjust input-output of a black-box model, a developer has to deal with many challenges: first, the controller may not converge to equilibrium if the system does not have a monotonic relationship

between a single input and a single output. Second, without an estimate of the sensitivity of the outputs with respect to the inputs, the controller may become too aggressive or too slow. Third, the controller cannot adapt to different operating regions in the input-output relationship. For example, Zhu et al. [122] shows that the mean response time is controllable when the CPU consumption is close to the allocated capacity and uncontrollable when the CPU allocation is more than adequate. The "uncontrollable" in this context means that the output is insensitive to changes in the input.

4.1.3 Scaling Resources Vertically

Our approach depends on the ability of virtualization technologies to reallocate the virtualized resource dynamically. However, isolation is a prerequisite for virtualized resources dynamic reallocation. In this section, we discuss the isolation and the scalability of CPU, I/O, and Memory as the following:

The computation power of the physical CPUs (pCPU) is distributed among the VMs by a scheduler. Each VM has one or more virtual CPUs (vCPUs). The vCPU is mapped to run on a pCPU by the scheduler according to specific policies. These policies determine the scheduler characteristics that make it suitable for some environments than others. For instance, Xen [20] has three schedulers: Borrowed Virtual Time (BVT), Simple Earliest Deadline First (SEDF), and the Credit scheduler [33]. Among these schedulers, only SEDF and Credit scheduler have a non-work-conserving mode, which enable the scheduler to cap the capacity of the CPU to specific value (e.g., 50% of the CPU capacity) and prevent an overloaded VM from consuming the CPU capacity of other VMs. Venkatanathan et al. [108] indicate that Amazon EC2 uses Xen Credit scheduler in a non-work-conserving mode. In this chapter, we use the Credit scheduler in non-work-conserving mode to scale the capacity of the vCPU dynamically.

As in the case of CPU isolation, I/O isolation is also the scheduler's job. However, current hypervisors show that an I/O-intensive VM can influence the performance of other VMs. For instance, in Xen [20], I/O device follows a split-driver model. Therefore, only an Isolated Device Domain (IDD) has access to the hardware using native device drivers. Cherkasova [33] and Gupta et al. [53]

demonstrated that the split-driver model of Xen complicates the CPU allocation and accounting. In split-driver-model, the IDD processes I/O operations on behalf of guests' VMs. This behavior allows an I/O intensive VM to overload the whole system. To enhance the accounting mechanism, [53] proposed SEDF-DC. It accounts the CPU usage of an IDD into corresponding guest domains that trigger I/O operations. However, SEDF-DC is still a prototype and it is not implemented to the deployed version of Xen. The fair-sharing of the I/O devices is the default behavior in Xen hypervisor [20]. Nevertheless, the fair-sharing in Xen does not guarantee limiting each VM to its share.

At the initialization time of a VM, the hypervisor allocates an isolated virtual Memory for it. Memory isolation makes the VM unaware of other VMs Memory demand. A Ballooning technique is developed to enable passing Memory pages back and forth between hypervisor and hosted VMs. However, it requires the cooperation of the VM's operating system. Therefore, a VM's operating system should be plugged with the balloon driver to enable the communication between the VM's operating system and the hypervisor. Whenever a hypervisor decides to reduce a VM's Memory size (i.e., reclaim pages from VM and inflate the balloon [111]), it determines the target balloon size. If the VM's operating system has plenty of free Memory, inflating the balloon will be done by just pinning free Memory pages (i.e., prevent access to these pages). However, if the VM's is already under Memory pressure, the operating system should decide about the Memory pages that should be paged out to the virtual swap device[111]. Both VMware and Xen enable dynamic Memory allocation using balloon driver technique. Nevertheless, communication between balloon driver and the hypervisor, in VMware, goes through a private channel, which limits scaling the Memory to the hypervisor. On the other hand, Xen hypervisor provides an interface that allows scaling the VM's Memory size dynamically. It is important to note that the VM cannot acquire virtual Memory bigger than the assigned Memory at booting time. For example, if a VM is initialized with 1GB Memory size, the possible values that can be assigned dynamically are 1GB or lower. Therefore, to give a VM a wide range of possible allocation values, we start the VM with a high Memory allocation to be changed later.

4.2 Proposed Architecture

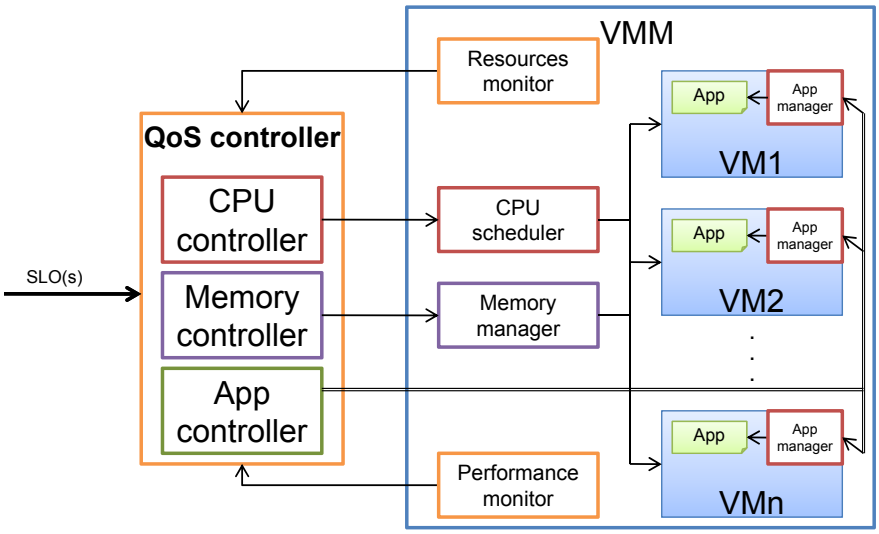


Figure 4.2: Our proposed architecture shows CPU, Memory, and Application controllers running in parallel

Our architecture is centralized around the *QoS controller*, which communicates with several modules implemented into both the VMM's level and VM's level as the following:

- *Resources monitor* module dynamically measures the resources consumption and updates the QoS controller with new measurements. The module depends on *xentop* tool to get the CPU consumption of each VM.

- *CPU scheduler* is implemented to dynamically change the CPU allocation of the VMs according to determined values by QoS controller, this module depends on Xen credit scheduler as an actuator for setting the CPU shares for VMs. The credit scheduler has a non-work-conserving-mode which enables determining a limited portion of the CPU capacity for each VM. The credit scheduler prevents an overloaded VM from consuming the whole CPU capacity of the VMM and degrading the other VMs' performance.

- *Memory manager* is implemented with the help of the balloon driver in Xen. This allows online changing of the VMs Memory. To have a wide

range of the Memory size, we gave the variable *maxmem* an initial high value (i.e. 500MB) in all user domains configuration files then use the *mem-set* command to change the Memory size into the value determined by the controller.

- *Performance monitor* also keeps the controller updated with performance metrics (i.e. the average response time and the throughput). The *performance monitor* is implemented on network device of the VMM, so it can monitor both the incoming and outgoing traffic.

- *Application manager (App manager)* is implemented into the VM's level. Its job is to get a new configuration value for *MaxClients* from the Application controller (App controller), then to update and restart the Apache server with the new configuration.

As seen in Figure 4.2, the input to *QoS controller* is the SLOs. In our approach, the SLO is to keep average response time of Apache web server within specific range, regardless of the workload variation. The outputs of the QoS controller are the calculated CPU capacity, Memory allocation, and *MaxClients* that satisfy the SLO. Inside the QoS controller, we implemented the following controllers:

CPU controller: It is a nested loop controller developed in [123]. The inner controller (CPU utilization controller) is an adaptive-gain integral (I) controller designed in [114]:

$$a_{cpu}(k+1) = a_{cpu}(k) - K_1(k)(u_{cpu}^{ref} - u_{cpu}(k)), \tag{4.1}$$

where

$$K_1(k) = \alpha.c_{cpu}(k)/r_{cpu}^{ref} \tag{4.2}$$

The controller is designed to predict the next CPU allocation $a_{cpu}(k+1)$ depending on the last CPU allocation $a_{cpu}(k)$ and CPU consumption $c_{cpu}(k)$, where the last CPU utilization $u_{cpu}(k) = c_{cpu}(k)/a_{cpu}(k)$. The parameter α is the constant gain that determines the aggressiveness of the controller. In our experiments, we set β=1.5 to allocate CPU aggressively at overloading time and decrease it slowly with the workload decrease.

Equation 4.1 requires determining u_{cpu}^{ref} value. Nevertheless, Figure 4.3 shows that the response time is not only dependent on the CPU utilization, but also on the request rate. Therefore, it is more realistic to have u_{cpu}^{ref} value automatically driven by the application's QoS goals rather than being chosen manually for each application.

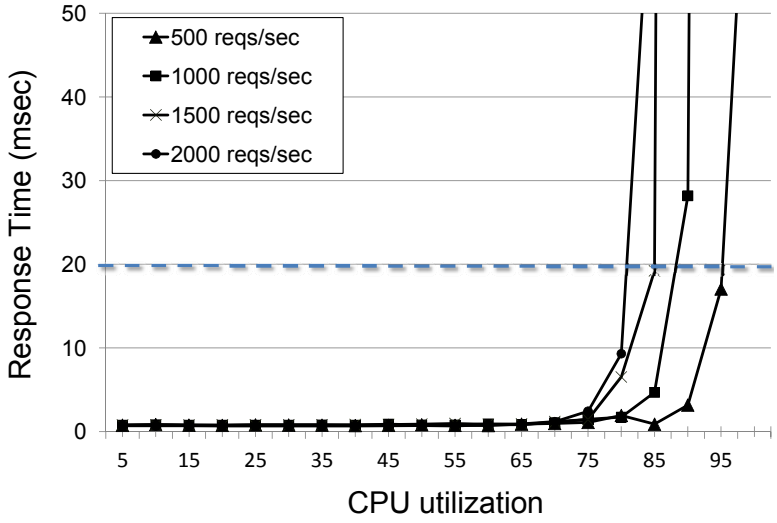


Figure 4.3: Mean response time v.s. CPU utilization under different request rates

For this goal, another outer loop controller is designed by [123] to adjust the u_{cpu}^{ref} value dynamically and ensure that the QoS metric (i.e., response time) is around the desired value. This outer loop controller can be interpreted into the following equation:

$$u_{cpu}^{ref}(i+1) = u_{cpu}^{ref}(i) + \beta(RT_{cpu}^{ref} - RT_{cpu}(i))/RT_{cpu}^{ref} \tag{4.3}$$

Where $u_{cpu}^{ref}(i+1)$ is the desired CPU utilization, $RT_{cpu}(i)$ is the measured response time, and RT_{cpu}^{ref} is the desired response time determined by SLO. The outer controller (i.e., response time controller) ensures that the value fed to the CPU controller is always within an acceptable CPU utilization interval $[U_{min}, U_{max}]$.

In our experiment, we set β to 1.5. The CPU allocation and CPU utilization are limited to the interval [10, 80]. The desired response time is 20 milliseconds.

Memory controller: In our experiments we noticed that increasing the number

of Apache processes can increase the throughput. However, the performance is degraded drastically when the Apache processes consumes the whole available Memory. This enforces the system to swap parts of the Memory into the hard-disk. Consequently, additional workload is added to the overloaded CPU by the big number of processes. To keep the system away from bottlenecks, we implemented the Memory controller designed in [56]:

$$a_{mem}(i+1) = a_{mem}(i) + K_2(i)(u^{ref}_{mem} - u_{mem}(i)) \tag{4.4}$$

Where

$$K_2(i) = \lambda.u_{mem}(i)/u^{ref}_{mem} \tag{4.5}$$

The controller aggressively allocates more Memory when the previously allocated Memory is close to the saturation (i.e. more than 90%), and slowly decreases Memory allocation in the underloaded region. During our experiments, we set u^{ref}_{mem} and λ 90 and 1, respectively. The limits of the controller were set to [64, 512], where the 64 MB is the minimum allowed Memory allocated size, and the 512 MB is the maximum allowed allocated Memory size.

Application controller: Experimentally , we noticed that there was a specific value of *MaxClients* giving the best throughput and the minimum response time, as seen in Figure 4.1. Towards finding this value dynamically, we designed a heuristic Apache controller that monitors four measured values to determine the best *MaxClients*. The monitored values are: response time, throughput, CPU utilization, and number of running Apache processes. The controller saves the best record of these values. The best record is the one satisfies SLO and gives the highest throughput with less CPU utilization. Searching for a better record, Application controller increases *MaxClients* by 10 every five minus.

With each new measurement of monitored values, Apache web server compares the current record with the best record. The current record overrides the best record if it showed better performance. Furthermore, if the App controller noticed a violation of the SLO (i.e., response time) it detect the occurrence of one of two cases:

Case (i) Apache processes starving problem: Apache processes starving problem occurs when Apache server runs a large number of processes, as a result,

the CPU spends most of the time switching between these processes while giving a small slot of time to each process. Such behavior causes requests to spend longer time in application queue, which end up with a high response time or requests timed out. To eliminate this problem, the Apache controller reloads the Apache server with the last best record, this reload reduces the number of running processes, reduces CPU utilization, and consequently reduces response time.

Case (ii) Resources competition problem: A competition on resources is marked by Apache controller when the following states coincide: (i) The response time increases. (ii) Number of running apache processes reaches *MaxClients* value. (iii) CPU utilization decreases (i.e. less than 90%). The reason behind the low utilization in the competition case is that the CPU controller proposes a higher allocation for the CPU, but the competition prevents applying that allocation. In this case, the scheduler (i.e., The Xen's Credit Scheduler) employs its fair share (e.g., 50% in case of two VMs). The reaction in case of the competition on resources is to reduce the number of *MaxClients* by 10, every five minutes, until the SLO is satisfied again.

4.3 Evaluation

Our experiment conducted on a testbed of two physical machines (Client and Server) connected by 1 Gbps Ethernet. Server machine has Intel Quad Core i7 Processor, 2.8 GHz and 8GB of Memory; it runs Xen 3.3 with kernel 2.6.26-2-xen-686 as hypervisor. On the hypervisor are hosted VMs with Linux Ubuntu 2.6.24-19. These VMs run Apache 2.0 as a web server in prefork mode. For workload generation, *httperf* tool [83] was installed on client machine.

In our experiments we deal with two kinds of VMs. First, a static VM that has a fixed allocation of resources during its run time. Second, an elastic VM initialized with specific allocation of resources that can be changed dynamically by the CPU and Memory controllers according to the workload variation.

The examined scenarios are as the following: The first scenario compares a static VM and an elastic VM coping with workload increase. The second scenario considers two elastic VMs, equipped only with CPU and Memory controllers, and

competing on resources. The third scenario considers two elastic VMs, equipped with CPU, Memory, and application controller; and competing on resources. In all our experiments, the SLO is to keep the mean response time less than 20 milliseconds.

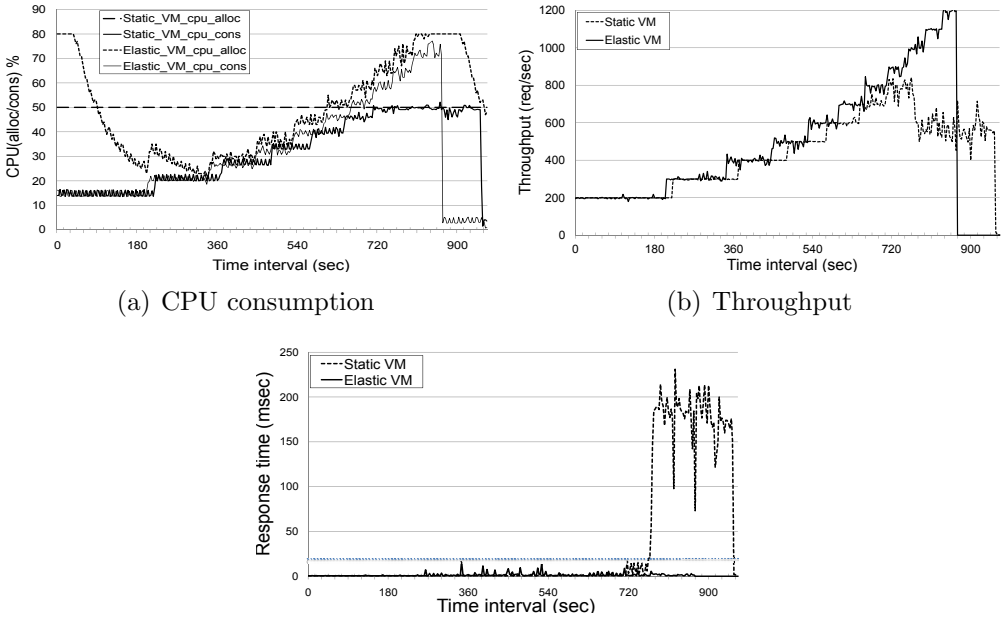
4.3.1 Static VM v.s. Elastic VM

In this part, we assume that the elastic VM is hosted on a physical server with a plenty of resources. So, the server is able to fulfill all requests for more resources. In this experiment, we would like to study our elastic VM ability to cope with the traffic variation and maintaining the specified SLO. To express the improvements, we ran the same experiment onto a static VM with similar but static resources. As a basis for our experiments; we used dynamic web pages requests, in each request, the web server executes a public key encryption operation to consume a certain amount of CPU time. The step traffic initiated with *autobensh* tool [99], it is started with 20 sessions, each session contains 10 connections. The number of sessions have been increased by 10 with each step. The total number of connections for each step is 5000, and the timeout for the request is 5 seconds. The measured throughput for the generated web traffic is seen in figure 4.4(b).

Each step of the charts in Figure 4.4(b) represents the throughput of a specific traffic rate. For example, in the period (0 to 210) seconds, both VMs respond to 200 req/sec successfully without any lost or timed-out requests. In this period of time, both VMs were able to allocate the required CPU capacity that copes with coming requests. In the first period, we notice in Figure 4.4(a) how the elastic VM started releasing the over-allocation of CPU slowly from the highest starting allocation (i.e. 80%) to the predicted suitable value.

This behavior of the elastic VM, allocating resources aggressively then converging slowly to the optimal allocation, enabled it to respond to the whole traffic rates successfully.

On the other hand, the static allocation of CPU enabled the static VM to maintain the SLO until the second 780. Afterwards, the static VM's CPU is overloaded causing requests to wait longer in the application queue, and consequently the response time increased. The overutilization results in a continues



(a) CPU consumption

(b) Throughput

(c) Response time

Figure 4.4: A static VM v.s. an elastic VM responding to a step traffic

period of SLO violation as seen in Figure 4.4(c).

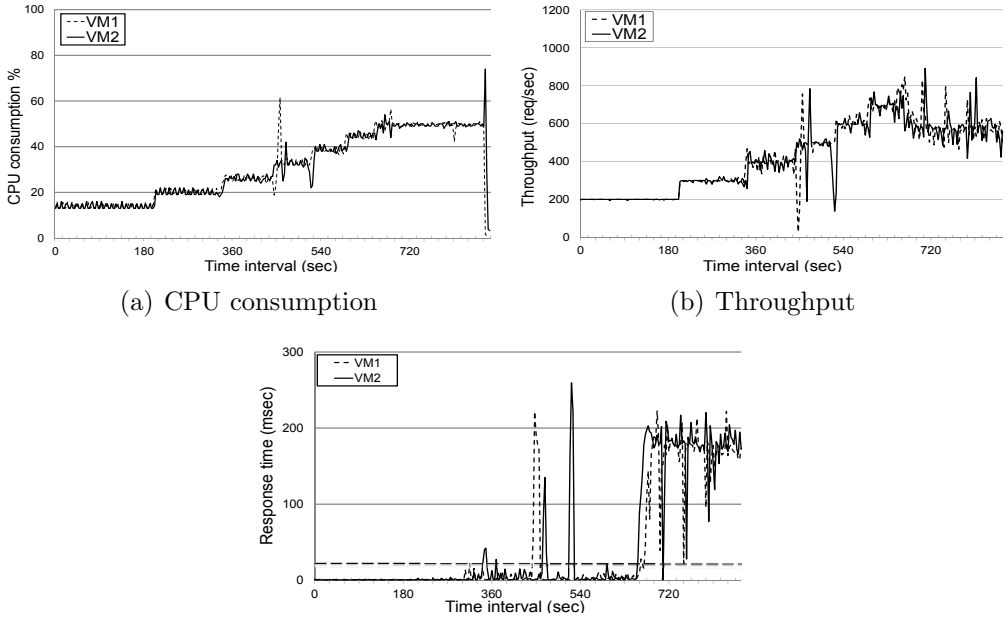
Requests rate(req/sec)	Static VM (timeout %)
900	7.232
1000	15.328
1100	18.258
1200	27.772

Table 4.1: The timeout started after the Static VM received 900 req/sec.

Furthermore, some of the queued requests timed out before being served, the percentage of timed-out requests with the corresponding traffic rate is illustrated in Table 4.1. The table started at 900 req/sec because there was no significant timed-out traffic before this rate. If it is compared to the elastic VM's throughput for the same high traffic rate (i.e. 800 to 1200 req/sec), Figure 4.4 shows how the elastic VM was able to borrow more resources dynamically, serve more requests, maintain a low response time, and prevent SLO violation.

4.3.2 Two Elastic VMs Competing on CPU resources - Without Application Controller

In the previous experiment, we studied the ideal case where the host was able to satisfy the elastic VM's need for more resources to cope with the increase of incoming requests. In this experiment, we study the competition on the CPU between two elastic VMs. Unlike experiments that have been done by [56], where each VM's virtual CPU has been pinned into a different physical core, we pinned the virtual CPUs of two elastic VMs into same physical core to raise the competition level. The same step-traffic has been simultaneously run to the elastic VMs. Nevertheless, in this experiment we ran the elastic VM with only CPU and Memory controllers.



(a) CPU consumption

(b) Throughput

(c) Response time

Figure 4.5: Two elastic VMs (without) Apache controller responding to step traffic

The result of the experiment is illustrated in Figure 4.5. The figure shows that elastic VMs were not able to cope with the traffic rate higher than 800 req/sec while the host committed only 50% of the CPU power for each VM starting from second #660 as seen in Figure 4.5(a). Due to competition on the CPU, many

requests are queued for a long time causing a high response time and continues violation of SLO, as seen in Figure 4.5(c). Moreover, many other requests are timed-out before being served as seen in second and third columns of table 4.2. From the above experiment, we can conclude that elastic VM can improve the performance when the host has more resource to redistribute. However, in case of competition on resources and under the fair scheduling, elastic VM (without) Apache controller merely behaves as a static VM.

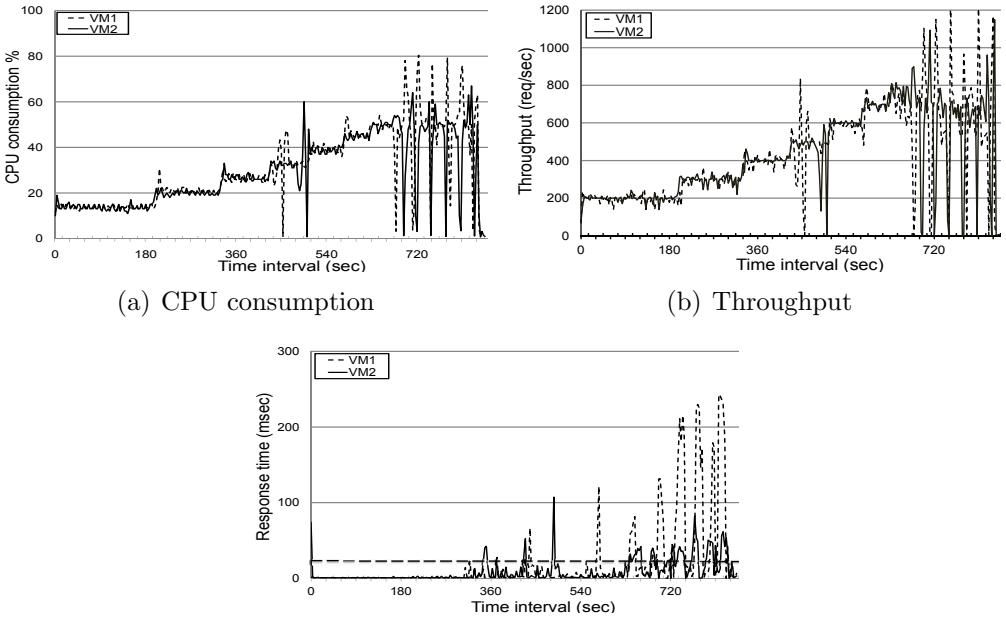
4.3.3 Two Elastic VMs Competing on CPU Resources - With Application Controller

The previous experiment is repeated on two elastic VMs (with) Apache controller, Figure 4.6(a) shows that in spite of the limited CPU capacity (i.e., 50%) available to each VM, starting from second #660, the Apache controller do two improvements. First, the moment of the Apache reload was a good chance for the other Apache server to have more processing power and serve more requests, as seen in Figure 4.6(a). Second, after the reload, the Apache servers are tuned with the last best *MaxClients* value.

	VM1	VM2	VM1	VM2
(req/sec)	Timeout requests(without)		Timeout requests(with)	
800	4.0%	0%	0%	0.2%
900	13.3%	23.8%	8.8%	8.2%
1000	20.5%	23.2%	16.52%	17.0%
1100	25.0%	35.0%	21.0%	22.0%
1200	31.0%	37.0%	26.2%	27.8%
	SLO violation(without)		SLO violation(with)	
	23.9%	26.4%	14.7%	16.8%

Table 4.2: Two elastic VMs (without) Application controller v.s. two elastic VMs (with) Apache controller responding to step traffic

As seen above, the proposed Apache controller not only looks for the optimal *MaxClients* value, but also eliminates performance bottlenecks by keeping a history of the last best running configurations.



(a) CPU consumption

(b) Throughput

(c) Response time

Figure 4.6: Two elastic VMs (with) Application controller responding to step traffic

4.4 Use-cases for Vertical Scalability

4.4.1 Database Vertical Scalability

Database scale out (i.e., scaling horizontally) have been discussed in literature [57][48]. Nevertheless, a little attention has been given to the overhead of initiating new replicas in database tier. Some providers offer vertical scalability for relational database (e.g., Amazon RDS [13]). Nevertheless, Amazon RDS implies restarting the VM instance to apply the new assigned capacity, which interrupts the service of Internet applications.

The last few years have witnessed the emergence of very large-scale Internet applications, such as Twitter and Facebook, that manipulate a massive and highly distributed amount of data. To cope fast with the increase of the data size, a new database system called NoSQL (interpreted as "not only SQL") had been emerged. NoSQL database is not built primely on tables and relations as the relational database systems. It is built on the key-value pairs, which gives it

the possibility to scale linearly and maintain high performance. Nevertheless, NoSQL database is known by non-adherence to the transaction characteristics: ACID (Atomicity, Consistency, Isolation, Durability), which makes it suitable for some applications rather than others. For example, losing the transactions' atomicity or consistency can cause big problems for an travel agencies, banking systems, or online retailer.

In prior research [40], we considered scaling a relational database instance vertically without interrupting the service. The scalability is controlled by static thresholds to increase or decrease number of cores, according to workload variation. The vertical scalability allowed the database tier to cope rapidly with the workload increase. Typically, public cloud providers deal with different periods of workload. A good management of the workload (e.g., hosting complementary workload on the same physical host) exploits the possibility of providing more resources to the overloaded VM instances without impacting the other customers. It also allows selling the spare capacity in the provider's infrastructure, as we explain in next section.

4.4.2 Increasing Spot Instances Reliability

Typically, IaaS providers deliver their services as reserved or On-Demand instances. Spot instances, a purchasing model invented by Amazon EC2, is a complementary service that allows customers to bid on the free capacity in the provider data centers. Spot instances offer the customers a reduction in the price over the other services. Nevertheless, the reduction in the price is on account of the reliability, which limits the Spot instances to time-insensitive workloads (i.e., batch jobs).

To provision a Spot instance, a customer determine a bid value. Whenever the bid value is greater than or equal the Spot instance's price, the provider initializes the instance for the customer. Whenever the price goes over the bid value, the VM instance is terminated. To ensure fair trading, the provider does not charge customers for the interrupted partial hours. Our experiments [42] [41] show that uncharged time could rise up to 30% of the instance total run time, which means a reduction in the provider's revenue.

In prior research [42] [41], we investigated scaling the VM instance capacity dynamically with the market price. So, instead of initiating new instances (i.e., scale horizontally), additional capacity can be purchased dynamically for the already running systems (i.e., scale vertically), such as databases. Our approach is evaluated using real historical traces from the Spot instances market. It increases the VM instance reliability, and in the same time increases the provider revenue.

4.5 Limitations of the Proposed Approach

The limitations of our approach are inherited from the limitations of the vertical scalability. First, it assumes the provider involvement in the application performance management, which is a complex task in large-scale infrastructures. Second, the vertical scalability can lead to conflict on resources, which exposes the performance of other costumers' applications to degradation. Third, vertical scalability is limited to one physical host. The live migrating of VMs into other hosts can solve conflicts on resources and the limited capacity of physical host problems. Nevertheless, live migration implies an overhead. Moreover, planning the migration of VMs with a dynamically changing workload is a complex problem. The above problems have been studied intensively during the last few years. Nevertheless, they still a hot research topics.

4.6 Summary

In this chapter, we employed the control theory to provide a fast coping with the workload variation in consolidated environments. Our system includes three controllers CPU, Memory, and Application running in parallel to preserve the intended SLO. The Application controller dynamically tune the web server parameter *MaxClients* to achieve a high throughput, mitigate the SLO violation, and reduce the ratio of lost requests. We evaluated our system in a Xen-based virtualized environment. The Application controller showed efficiency for mitigating the impact of competition on resource on an application's performance.

Chapter 5

Cloud Hosting for Internet Applications

The last few years have witnessed the emergence of cloud computing as a rapid, limitlessly scalable, and cost-efficient alternative in contrast to the in-house (i.e., on-premise) data centers. The IaaS model delegates more control to the customers over the provisioned resources. Hosting Internet applications in the IaaS environment is an efficient way to start a new and a sustainable business that expands the IT infrastructure gradually with the business growth. Moreover, on-demand provisioning provides a cost-efficient way for already running businesses to cope with unpredictable spikes in the workload. Nevertheless, an efficient scalability for an Internet application in the cloud requires a broad knowledge of several areas, such as scalable architectures, scalability components in the cloud, and the parameters of scalability components.

Multi-tier architecture has been used for decades to provide scalability for Internet applications. The emergence of the IaaS model enriches the multi-tier architecture with components (i.e., services) that help scaling resources dynamically (based on the actual demand) to maintain the Internet application performance. To automate the performance management, IaaS's customers are provided with tools that enable provisioning and terminating resources remotely. In this chapter, we study the impact of the introduced components and their parameters on the performance of Internet applications. We focus in our study on Amazon EC2

[12] since it is a pioneer of dynamic scalability in the IaaS environment. Nevertheless, we map Amazon EC2 concepts to other well known IaaS providers for comparison purposes.

Our contributions in this chapter are as the following:

- Study and compare the scalability components in several IaaS providers.
- Define the scalability parameters that have crucial impact on an Internet application performance.
- Investigate the problems that can impact the performance of an Internet application hosted in the IaaS layer.

In the next section, we study the architectures, components, and parameters necessary to implement a scalable Internet application in IaaS layer. In Section 5.2, we investigate the possible sources of the performance degradation of Internet applications. Finally, we conclude our study in Section 5.3.

5.1 A Scalable Internet Application in IaaS

Dynamic scalability is not only crucial for IaaS customers, but also for PaaS and SaaS providers [107]. IaaS providers offer architectures and components that ease scaling applications. On the other hand, a non-trivial part of the task lies in the customer's side. For example, the customer should design policies that control provisioning resources dynamically. Designing these policies depends mainly on the customer's understanding of Internet application behavior and demand. Due to the vast number of the hosted applications in the IaaS environment and the variant behavior and demand of each application, customers should not expect IaaS providers to monitor the performance of each hosted application. In fact, what an IaaS provider describes in the Service Level Agreement (SLA) is the running time of VM instances. For that reason, maintaining the performance of an application hosted in IaaS remains the customer's responsibility. In this section we begin with a detailed scalable architecture and then define the main components followed by the related parameters that can have impact on application performance.

5.1.1 The Scalable Architecture

Before the cloud emergence, multi-tier architecture has been used to provide the scalability for the Internet applications. Each tier of the multi-tier architecture has a cluster of physical nodes. The tier can be scaled out by adding more nodes. However, adding a physical node is time consuming job that cannot be done without human interaction.

The later development of IaaS model has been enriched the multi-tier architecture with many components that can be consumed as web services to provide a reliable and a dynamically scalable architecture. For example, instead of running physical servers in each tier, customers can run a number of VM instances that can be increased or decreased dynamically according to the workload. For fault tolerance, many VM instances in Amazon IaaS model can be grouped into one Auto-Scaling group that maintains running a specific number of healthy VM instances all the time. In addition to VM instances as computation units, Figure 5.1 shows auxiliary components, such as the Elastic Load Balancer (ELBs), network storage (e.g., S3 and EBS), and monitoring and notification services.

Consuming these services may include additional cost. However, the offered services are manageable, reliable, and fault-tolerant compared to customers self-created components. We study in details the components that are developed to improve the Internet application scalability in next section.

5.1.2 Scalability Components

With the advance of cloud computing infrastructure, many services and concepts have emerged to facilitate and support scalable and reliable Internet applications. In this section, we explain the services and concepts that are related to application scalability using Amazon EC2's terminology while also mapping them to other providers' terms.

1. **Amazon Machine Image (AMI):** AMI is a pre-configured operating system image that can be used to create a VM instance. Windows Azure, [81] as well as Amazon EC2, allow the clients to upload their own image or select from a list of available images. Different providers and communities

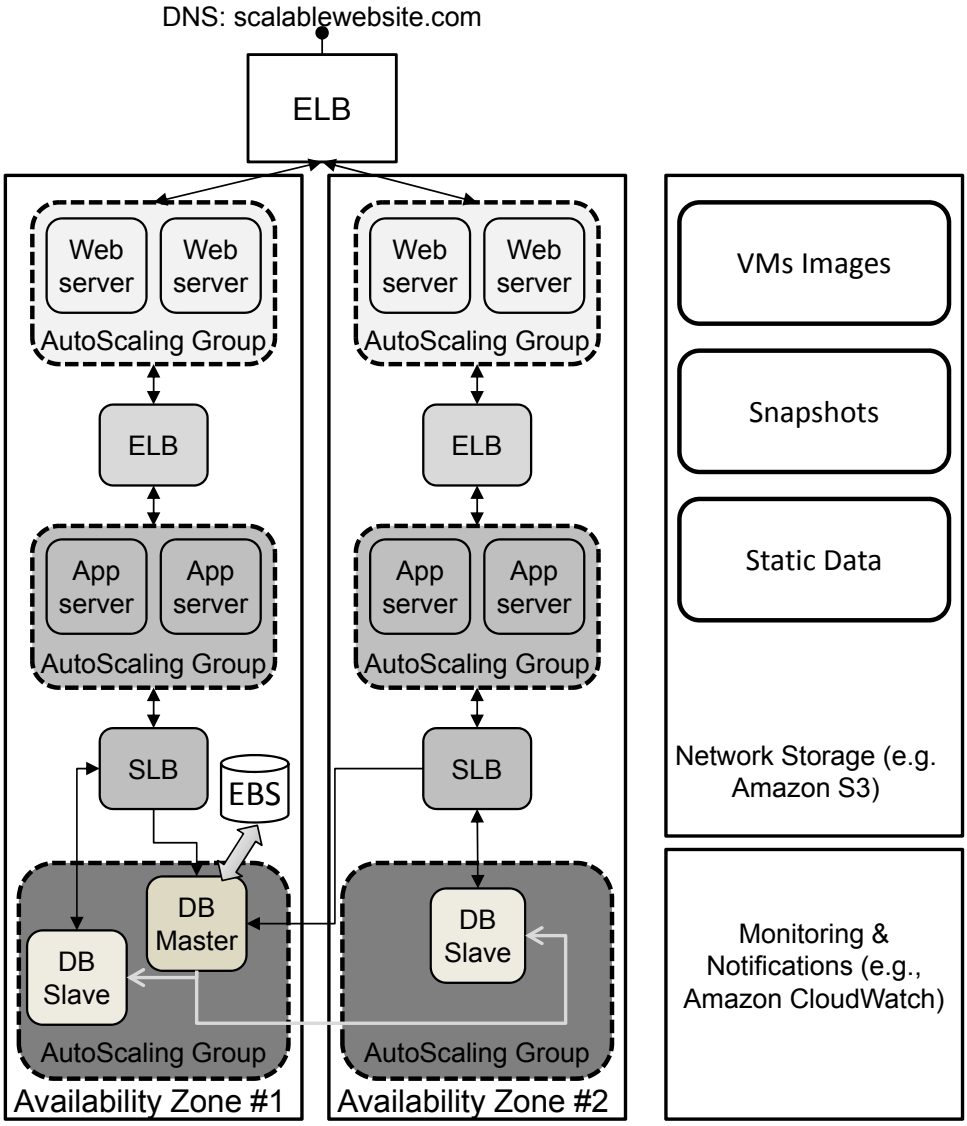


Figure 5.1: A detailed multi-tier scalable architecture

offer images with software stacks to deliver their software on-demand. These images are stored in a non-volatile repository (e.g., Amazon S3 [12]). Most of the IaaS providers also allow users to customize VM images and create their own images as snapshots.

2. **Amazon Simple Storage Service (S3):** S3 is a simple web service that provides a fault-tolerant and durable data storage. The data is stored as

objects replicated across different geographical regions for higher availability. Stored objects can be accessed using URLs. S3 is optimal for storing static data that is delivered to users directly without manipulation. In addition, it is used to store VM images. The rate of data transfer varies from one provider to another according to the used technology. For example, the data transfer rate in RackSpace [3] is measured as 22.5 MB/s[76]. This is due to the low overhead of running a VM instance compared with Amazon EC2 and Windows Azure, as we explain in Section 5.2.2.

3. **Amazon Elastic Block Store (EBS):** EBS is a block level storage volume that persists independently from the VM instance life. Unlike the local storage which can be lost after a failure or a planned termination of a VM instance, the EBS volume lasts permanently. Consequently, it is used for applications that need permanent storage like databases. At any time a VM instance fails, the EBS volume can be re-attached to another healthy VM instance. Despite the fact that the EBS volumes are stored redundantly, to reduce the recovery time from a failure, users can periodically take snapshots of these volumes. In addition, for high durability, the snapshots also could be stored in S3 storage. As an example, in Figure 5.1, a best practice is to map the Master database to EBS storage. Whenever the database instance fails, we can remap the storage volume to another instance and restore the database to operational mode quickly. To balance the workload on database tier we can dump the database into S3 storage that can be used to initiate Slave instance as we explained in Section 5.2.2.

4. **Regions and Availability Zones:** Cloud infrastructure is designed to offer a fast and reliable service globally. As a result, data centers of a cloud provider are distributed to span more geographical location areas (i.e., Regions). Within each Region, there are many Availability Zones that are engineered to be isolated from failure propagation. The networking between the Availability Zones within the same Region is inexpensive and induces a low networking latency. On the other hand, the networking between VM instances within Availability Zones located in different Regions implies networking through the Internet. As a result, the cloud customers are not

advised to split application tiers into different Regions, even for reliability and fault-tolerance.

5. **Static Load Balancer (SLB):** Also referred to as a Load Balancer (LB) or a dispatcher. Usually, it is a VM running a third-party software (e.g., HAProxy, Nginx, and Apache-proxy) to distribute workload across many back-end VM instances (i.e., replicas). The redirection of requests to the back-end instances follows a specific algorithm. Round robin is a widely used algorithm in Load Balancers. In case of unequal size of back-end instances, weighted round robin can be used to direct a quantity of the traffic proportional to the instance capacity. In case of databases, especially when the majority of the requests dispatch read queries, a load balancer can be stood in front of the database tier. The database tier itself can be split into a Master database instance, and one or more Slave instances. The write quires are directed by the load balancer to the Master database while the read queries are directed to the Slave instances. To keep data consistent, the Master and Slave instances should not be located in different Regions to avoid high latency synchronization through the Internet. Nevertheless, to scale database into different Regions, other techniques like database sharding [36][6] can be used.

6. **Elastic Load Balancing:** The challenge with the SLB is that they are in need to have up to date lists of the available healthy replicas behind it. Whenever a replica fails or does not work properly, it should be excluded from the list to avoid losing or delaying the routed traffic. On the other hand, whenever a new replica is initiated, it receives a new IP address that is unknown to SLB. For third-party load balancers, it is the Internet application owner's job to mange registering and de-registering instances to the SLB. This implies running an additional component to interface with the load balancer and update the replicas list with each exclusion or addition of a replica. Moreover, SLB owners should run additional components to allow the balancer to distinguish between healthy and non-healthy replicas. Alternatively, Elastic Load Balancer of Amazon EC2 is supported with additional control component that keeps watching the status of the replicas,

whenever a VM instance does not respond properly; it is excluded from the replicas to prevent routing traffic to it. When the instance recovers to healthy mode, ELB can consider it in the possible replicas again. It is important to note that registering and de-registering instances to the Load Balancer is not part of the ELB job. They are accomplished by using the so-called - Auto-Scaling Group, which is explained next. On the contrary to ELB as a software load balancer, GoGrid [1] offers a hardware load balancer that has a rich interface with many functions. One of the distinguishing features of the GoGrid load balancer is the ability to have log files format similar to apache style access log. Furthermore, it has an important feature called connection throttling, which allows the load balancer to accept only a pre-defined number of connections per an IP address. By this, the load balancer can mitigate malicious or abusive traffic to the applications.

7. **Auto scaling group:** It is a concept by Amazon EC2 that keeps a healthy group of instances running under a unique name. At the creation time of the group, the user can specify the minimum number of the healthy instances that should be available all the time. Whenever a VM instance does not work well, the group controller replaces it automatically with a new one. Connecting the auto scaling group with an ELB is necessary to provide the ELB with an updated list of the available running replicas within the scaling group.

8. **Auto scaling policies:** Auto scaling policies should be attached to a specific scaling group. They describe how the scaling group should behave whenever it receives a scale out or down trigger.

9. **CloudWatch:** A web service that enables monitoring various performance metrics, as well as configuring alarm actions based on the status of the monitored metrics. For example, the user can set up CloudWatch to send an email or trigger scalability when the CPU utilization of a database instance goes over 70%.

10. **Route 53:** In reality, ELB is limited to one region. As a result, Amazon offers Route 53 [16] as a scalable and highly available Domain Name System

(DNS). It allows scaling an Internet application globally for less latency and higher reliability. With Route 53, Internet application users can be directed to the closest region according to their geographical location and therefore, the users will be served from the closest data center. This allows a geographical distribution of the load and reduces the traffic latency.

We summarize the main scalability components that are implemented by major public IaaS providers in Table 5.1.

Amazon EC2	**Windows Azure**	**Rackspace**	**GoGrid**
AMI	Images	N/A	GoGrid Server Images (GSIs) and PartnerGSIs
ELB	N/A	Cloud Load Balancer	F5 Load Balancer (Hardware)
EBS	Windows Azure Drives	Only a local storage	N/A
S3	Azure Blob Storage	Rackspace (cloud files)	Cloud Storage
Regions and Availability zones	Regions but no Availability zones	Regions but no Availability zones	Regions but no Availability zones
Scalability Group	N/A	N/A	N/A
Scalability Policies	N/A	N/A	N/A

Table 5.1: Scalability components of Amazon EC2, Windows Azure, Rackspace, and GoGrid

As shown in Table 5.1, Amazon EC2 has all the components that are necessary for efficient scalability. For the other providers, third parties like RightScale [4], open source management tools like Scalr [97], or internally implemented controllers are necessary to implement an automated scalability.

5.1.3 Scalability Parameters

After explaining the scalability components in Table 5.2, we highlight some of the parameters that should be set by the customer and have crucial impact on the application performance.

Component	Parameter	Description
Auto Scaling Group	default-cooldown	The time period that should pass after a successful scaling to consider a new one. This value can be determined globally per scalability group or individually per each scaling policy.
Auto Scaling Policy	cooldown	Depending on the incoming workload's fluctuation, customers can determine the best *cooldown* period of time that should pass after each scale either up or down.
Auto Scaling Policy	adjustment	This parameter determines the size of the scaling step. The positive values means scaling out, while negative values means scaling down. In Section 7.1.3, we study the impact of the size of the step on both the cost and the performance.
CloudWatch	metric-name	The metric to be monitored. In Section 7.1.1, we concentrate on the the CPU utilization.
CloudWatch	threshold	The threshold value at which a determined arrangement will be carried, such as initiating a new instance when the monitored metric (e.g., CPU utilization) is higher than the threshold 70%.
CloudWatch	period	The time frame (in seconds) in which Amazon CloudWatch metrics are collected.
CloudWatch	evaluation-periods	The number of consecutive periods for which the value of the metric must be compared to the threshold.

Table 5.2: Scalability parameters that have impact on the scalable application performance

In the next section, we explain a practical example for configuring scalability for one tier of an Internet application using Amazon EC2 scalability components.

5.1.4 Configuring Automatic Scaling in the IaaS Layer

In the following example, we summarize how a cloud's customer can enable the scalability to specific tier using the components offered by Amazon EC2. The purpose of this example is to express the parameters that have a high impact on the application performance. In this example, we assume a web tier that should maintain at least two VM instances, and can scale out to fifteen instances of type *m1.small*. The group adds one instance per a scale out operation, and terminates one instance per a scale down operation. The scale out is triggered when the aggregated CPU utilization of all the instances in the scalability group goes over 70%. On the other hand, the scale down is triggered when the aggregated CPU utilization of all the instances in the scalability group goes under 30%. The system will not scale out before three minutes of the last scale out operation, and will not scale down before five minutes of the last scale down.

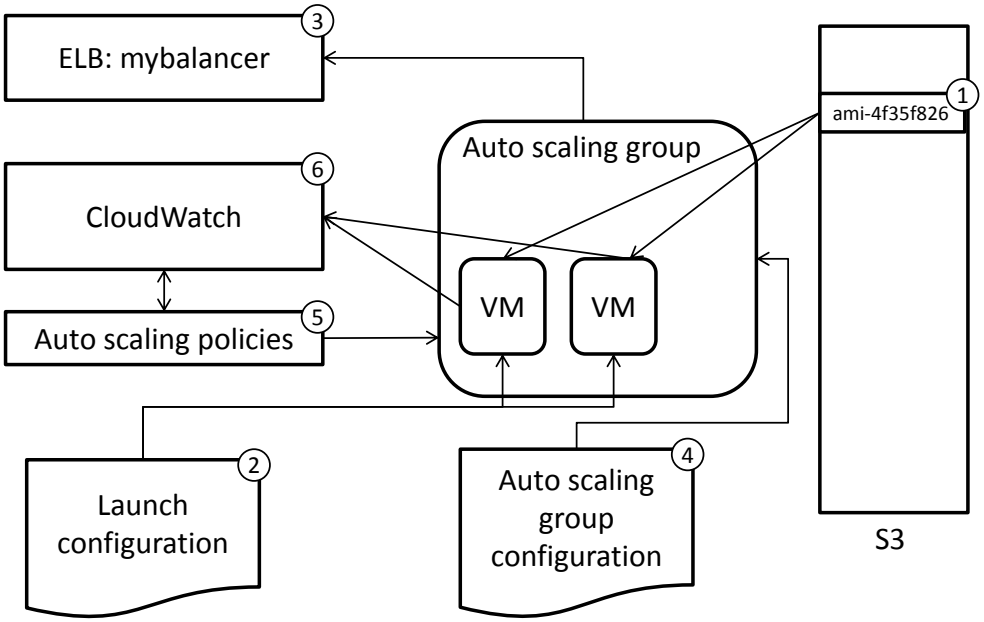


Figure 5.2: The main components that support automatic scalability in Amazon EC2

1. **Prepare an image to run:** As explained in Section 5.1.2, customers can create their own instance or pick one of the available images offered by the

provider. For example, we consider running an instance from the image ami-4f35f826, where the customer is supposed to install a web server and configure it with the IP or the DNS of the load balancer of the application tier. The customer can copy the html pages to the web server folder in the VM image. If the html code is updated frequently, it is more practical to keep the code in external storage (e.g., S3) and retrieve it at the VM initializing time.

2. **Launch configuration:** For auto scaling, the customer should predetermine the launch configurations. To create launch configurations, customers should determine a unique name of the configurations, a valid instance id, the type of the instance, the name of the key pair, and the security group. An example of creating a launch configuration is as follows:

```
as-create-launch-config my_launch_conf_group
--image-id ami-4f35f826 --instance-type m1.small
--key my_key --group my_group --monitoring-enabled
```

   The key and security group can be created using the dashboard. More details can be found in Amazon [12].

3. **Running a load balancer:** If a user decided to run ELB, the CNAME of the Internet application should point to the DNS name of the ELB not the IP. It appears that, Amazon EC2 does not dedicate a public IP for each ELB. In our example, we consider running an ELB called *mybalancer*. It is necessary to determine both the incoming port of the ELB and the forward port that the replicas are listening to. The elastic load balancer should also be configured with metrics that help it to abandon non-healthy replicas depending on predetermined criteria. Running and managing an ELB can be done either by command line [15] or through the dashboard.

4. **Auto scaling group configuration:** To create a scaling group, customers should determine a unique name for it, a launch configuration, an availability zone, a minimum number of instances, a maximum number of instances, and a grace period in seconds. The purpose of the grace period is to give the

system a time to stabilize after each initialization of a VM instance within the group. The command of creating a scaling group can be as follows:

```
as-create-auto-scaling-group my_scaling_group
--launch-configuration my_launch_conf_group
--availability-zones us-east-1a --min-size 2
--max-size 15 --load-balancers mybalancer
--health-check-type ELB --default-cooldown 120
```

5. **Auto scaling polices:** In our example, to create a scale out policy that should be triggered by the CloudWatch whenever a specific condition is fulfilled, we run the following command:

   ```
   as-put-scaling-policy --auto-scaling-group
   my_scaling_group --name scale-out --adjustment 1
   --type ChangeInCapacity --cooldown 180
   ```

   A similar policy, but for scale down can be as follows:

   ```
   as-put-scaling-policy --auto-scaling-group
   my_scaling_group --name scale-down
   "--adjustment=-1" --type ChangeInCapacity
   --cooldown 300
   ```

   As displayed above, to create a scaling policy, the customer should configure these parameters: the name of the auto scaling group, a unique scaling policy name, the size of the scaling step, the type, and the *cooldown* time in seconds. If the *cooldown* is not determined at the creation time of the scaling policy, the value of *–default-cooldown* scaling group will be considered as a *cooldown* value. The positive scaling step (i.e., adjustment) means adding the specified number of instances to the scaling group, while negative adjustment means removing the specified number from the scaling group. More details about the command parameters can be found in [14].

6. **CloudWatch:** provides monitoring service that allows customers to watch their application performance. To trigger scaling policies, CloudWatch

should be configured either with CloudWatch command line [11] or through the web interface. Amazon offers an easy web interface that enables creating metric alarms. There are many metrics to monitor, including single instance metrics or aggregated metrics. In our example, we select the aggregated metric of an auto scaling group while it describes the whole group performance. The command that is necessary to trigger "scale-out" policy when the aggregated CPU utilization of the scaling group "my_scaling_group" is measured three times consecutively higher than or equal 70%, is as the following:

```
mon-put-metric-alarm  --alarm-name HighCPUAlarm
--alarm-description "Alarm when the aggregated CPU
utilization of my_scaling_group is greater than or equal 70%"
--metric-name CPUUtilization --namespace "AWS/EC2"
--statistic Average --period 60 --evaluation-periods 3 --threshold 70.0
--comparison-operator GreaterThanOrEqualToThreshold
--dimensions "AutoScalingGroupName=my_scaling_group"
--unit Percent --alarm-actions = "arn:aws:autoscalin.."
```

The parameter *alarm-actions* contains the Amazon Resource Number (ARN), which is a unique identifier for each scaling policy. It is generated at creation time of the scaling policies and can be retrieved with the command line *as-describe-policies*. A similar alarm is necessary for scaling down at low CPU utilization but with different *comparison-operator*, *threshold*, *evaluation-periods*, and *alarm-actions*.

Currently, CloudWatch provides a free mode where the metrics are measured at five-minute interval. Based on our experience, free mode is not efficient for those applications having frequent changes in the workload. The other choice offers more frequent measurement (i.e., one-minute interval) by setting what is called a detailed monitoring of an instance; however it is charged monthly per an instance. Furthermore, for both modes, customers will be charged monthly per alarm and per thousand API requests. More details about the CloudWatch can be found in [10].

5.2 Facts Impacting the Performance of Scalable Applications in IaaS Environments

An Internet application hosted in IaaS layer is exposed to periods of performance degradation due to two main facts: First, the performance interference which is caused by consolidating different workloads belonging to different customers on the same physical host. We studied it intensively and proposed a solution to mitigate its impact on application's performance in Chapter 8. Second, the resources provisioning overhead which leads to periods of under-provisioning with each trigger to the scale out process.

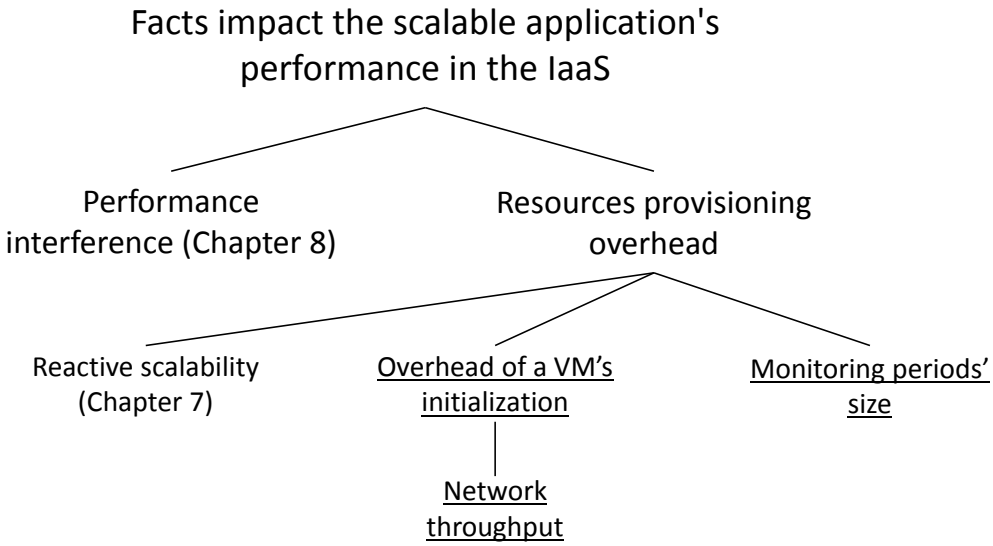


Figure 5.3: Facts impacting the scalable application's performance in IaaS environments

According to our practical experience, we attribute the overhead of resources provisioning in IaaS layer to three facts:

1. The reactive implementation of the scalability in the cloud (i.e., Amazon EC2).
2. The overhead of a VM's initialization which itself is influenced by the network throughput.
3. The monitoring periods' size.

Because of the reactive implementation of the scalability in the cloud, the scale out is triggered only when the system experiences an overloading state. For example, a web tier scale out can be triggered when the CPU utilization is equal to or higher than the scale out threshold 70%. But, what is the best scale out threshold? What happens if we rise up the scale out threshold to 80%? Will this reduce the cost? And what is the application performance for each scalability threshold? We answer these questions in Chapter 7.

In the following sections, we explain the impact of the underlined facts, seen in Figure 5.3, on the provisioning overhead of resources in the IaaS layer.

5.2.1 Initializing a VM in IaaS Environment

In this section, we explain the stages of initializing a VM in the IaaS layer. The goal is to identify the sources of the overhead, then to classify them according to the contributor. In other words, we want to distinguish between the stages at which the customer behavior can impact the time of bringing a VM to operational state from the stages that are fully influenced by the provider behavior.

In Figure 5.4, we use two types of boxes to refer to two types of stages. The dotted line boxes refer to the stages where customers can have impact on the completion time of the stage. The solid line boxes show stages that are completed entirely by the provider, and its completion time is totally dependent on the provider algorithms and the current demand on the data centers.

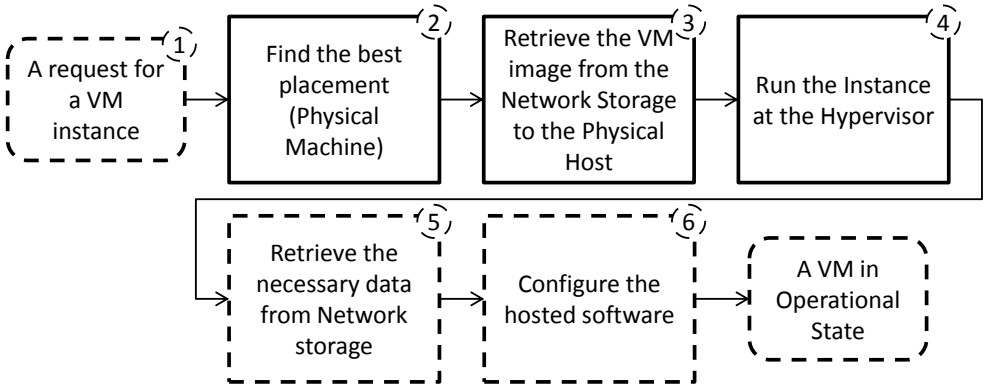


Figure 5.4: Initializing a VM in the cloud

The stages can be explained as follows:

1. The request for a new VM is initiated by the user either manually or by a scalability controller.

2. After receiving the request, the provider runs an algorithm to find the best physical host for hosting the new VM instance. Mao et al. [76] have shown that machine size, operating system, time of the day, and number of instances have different weights of impact on the VM's start up times

3. After finding a suitable physical host, the VM image is copied through the network.

4. Once the VM image is copied completely to the physical host, the hypervisor starts it. Running a VM in the cloud includes: booting the operating system, configuring the network setup (e.g., assigning a private IP and public domain name for a VM running instance), copying the public key for accessing the VM in case of Linux, or generating a random password in case of Windows operating system.

5. The best practice to assure that the new initiated VM has up-to-date data is to store data in a network storage. For instance, to run a web server with the last version of html pages, at run time, the server should be pointed to the repository where a tar ball of html files can be retrieved and extracted to the proper folder on the web server. Same procedures should be followed for the application and database instances. This can be implemented through scripts that executed at the VM start up time. However, customers should avoid retrieving huge amounts of data that may delay bringing VM instances into an operational mode. A long delay can make the dynamic scalability non-efficient as we explain in Section 5.2.2.

6. Whenever the user gets the domain name of the initiated VM, he can access it for configuring the hosted software. Users should avoid installing software at VM's start up time, while this can delay bringing the VM to the operational mode. The best practice is to pre-install the required software and packages to the VM image and prepare a script that runs automatically at the VM start up time to do the required configurations. These config-

urations may include passing the acquired private IP (i.e., internal IP) to another VM (e.g., the static load balancer).

As seen above, part of the delay in VM initialization is attributed to the provider. However, user should avoid any practices that can delay moving the initiated VM instance to the operational state.

5.2.2 Network Throughput Impact on the Application Scalability

A practical study by [76] shows that initiating a Linux instance at Rackspace [3] takes half of the time in comparison to Amazon EC2 [12]. It is due to the data transfer rate between VM and network storage (i.e., images repository), where it is 22.5 MB/s at Rackspace, while it is measured as 10.9 MB/s at Amazon EC2.

Considering the network throughput limit, we discuss scaling out a database tier horizontally by adding more Slave instances (i.e., read only instances). For this purpose, we calculate the time required to bring a new Slave database instance to operational mode. In our setup, we consider the typical setup for a scalable database in the cloud. Consequently, we assume an old dump of the whole database is stored in cloud storage (e.g., Amazon S3). Furthermore, an incremental backup of the binary files is also stored periodically to the storage as described in [45]. The non-compressed dump of our database is 1.1GB. To reduce the transferring time from the networking storage to the new VM, the database can be compressed to a lower size (i.e., 153MB). We assume that the Slave instance has a new and up-to-date MySQL installation. At the initialization time of the VM, we run a script that copies the compressed dump file from S3 storage to the VM local storage. The dump file is extracted and used to restore the database. Afterwards, we retrieve all binary log files that had been uploaded to S3 during the Master database running time. Theses logs also applied to the new database. Until this moment; new Slave instance do not know the Master database node. To point the new Slave VM to the Master node we should run a command similar to the below at the Slave node:

```
CHANGE MASTER TO
   ->MASTER_HOST='master_host_name',
   ->MASTER_USER='replication_user_name',
```

```
->MASTER_PASSWORD='replication_password',
->MASTER_LOG_FILE='recorded_log_file_name',
->MASTER_LOG_POS=recorded_log_position;
```

This command keeps the Slave node up-to-date with any changes to the Master node. We now consider the Slave node is ready to receive queries. It is now the time to update the static load balancer with the new available Slave replica and start it in operational mode. The average time required to run a new Slave database on *m1.small* instance at Amazon EC2 can be measured as follows:

1. Initializing a small instance in Amazon EC2: 100 seconds
2. Copying the compressed dump of database to the new VM instance: 16 seconds
3. Extracting the database dump and importing it to the new database: 255 seconds
4. Retrieving and applying the incremental binary logs: 130 seconds
5. Get the last updates from the Master node, restart the Slave, and update the load balancer with the IP address of the new Slave node: 68 seconds

As shown above, initializing a relatively small Slave database instance can be done, at best, in 569 seconds, which can be estimated as 10 minutes. Our measurements are exposed to increase if the size of the database dump is bigger or the number of incremental binary logs is higher. We should remember that our database is considered very small compared to large databases running in a production environment. Large databases require longer time to initialize a Slave VM from scratch, which raises the question about the efficiency of scaling-out a database Slave instances dynamically.

5.2.3 Control Window's Size

Understanding the workload characteristics is crucial to determine the control window's size (CWS). A small CWS allows the system to cope faster with the

workload; however, it can lead to either inconsistent scale out steps and consequently a higher total cost, or inconsistent scale down steps that degrades the performance. On the other hand, a large CWS causes the system to react slowly with the workload, which leads to periods of under-provisioning (i.e., performance degradation).

CWS is determined by two parameters. The first parameter is the *period*, which is 60 seconds at minimum in *CloudWatch*. The second parameter is *evaluation-periods*, which determines the number of the consecutive periods at which the measured value exceeds the threshold. The control window's size is calculated by the following relation $CWS = period * evaluation\text{-}periods$. To increase the scaling steps consistency, customers can use *default-cooldown* at the level of the scaling group, or override it by *cooldown* for individual scaling policies. Typically, the *cooldown* time should be less than or equal to the CWS, otherwise it can interfere the CWS as the following: $\widehat{CWS} = CWS - cooldown$, where $\widehat{CWS}$ is the new control window's size after a scale out or down.

5.3 Summary

In this chapter we summarize our practical experience in hosting a scalable Internet application in IaaS. As seen above, a deep understanding of many concepts, components, and parameters are necessary to run applications in IaaS properly. We invest this experience in Chapter 6 to develop our simulator ScaleSim, which considers the main components and parameters that have impact on the application performance. In Chapter 7, we go further by optimizing some parameters for better performance.

Chapter 6

A Large-scale Internet Application Evaluation

Many approaches are developed to improve Internet applications performance in the cloud [57] [67] [54] [94] [75]. Nevertheless, less attention is given for evaluating the scalability implementation in the current running production environments. Thus, we evaluate the current implementation of the scalability in the large-scale production environments, namely, Amazon EC2. Amazon EC2 is one of the biggest cloud providers offering an infrastructure for hundreds of satisfied customers[1]. The current scalability implementation in Amazon EC2 is used as a reference point for evaluating our proposed improvement in scalable applications performance.

As a matter of fact, large-scale experiments in the public IaaS imply a big cost, and are hard to repeat. On the other hand, prototyping systems on small test-beds do not guarantee their feasibility at large-scale. To overcome these issues, we develop ScaleSim simulator. It simulates the scalability components of Amazon EC2 [12]. The simulator is implemented into modules to allow other researchers to implement, run, and compare their algorithms at a large-scale level. To realize our results, we imparted the simulator with models that are extracted from a real cloud environment. In this chapter, we simulated RUBiS benchmark [31] at a large-scale level. However, the same procedures are applicable to any

[1]https://aws.amazon.com/solutions/case-studies/

Internet application running in production environment.

This chapter is organized as follows: In Section 6.1, we model the application performance in real environment, which is necessary to achieve realistic simulated results with an affordable cost. In Section we describe and evaluate the developed simulator (ScaleSim). Finally, in Section 6.3, we summarize our contributions in this chapter.

6.1 Modeling an Internet Application

Modeling an application behavior is crucial for maintaining the Internet application performance and avoiding resource bottlenecks. Nevertheless, modeling an Internet application is complex due to the nature of their architecture. For instance, each tier in the Internet application runs different software which itself has a different behavior. Moreover, the dependency between the Internet application tiers propagates the impact of resource bottlenecks from one tier to the others [57][103][119]. For instance, a database tier is known to be an I/O intensive application that requires a huge memory. At any time the allocated RAM exceeds 90%, the operating system starts paging into the virtual memory allocated at the hard disk. The swapping results in more I/O operations and consumes much of the CPU time. Consequently, it degrades not only the performance of database tier, but also the performance of the whole Internet application. In Chapter 4, we have avoided resource overloading by the vertical scalability. However, in this chapter we consider the public IaaS model, which allows only the horizontal scalability. In public IaaS model, a customer can set scalability policies with static thresholds that guide resource provisioning for maintaining the utilization within a specific limit.

However, what are the thresholds that maintain a high performance? Commonly used thresholds are (70% for scale out, 30% for scale down) and (80% for scale out, 40% for scale down)[1]. Nevertheless, our experiments show that the scale out and scale down thresholds have big impact on the system performance. These thresholds should be defined carefully for each tier depending on the running software, as we explain in Section 6.1.1.

[1]http://awsdocs.s3.amazonaws.com/AutoScaling/latest/as-dg.pdf

In our experiments, we used RUBiS benchmark [31] as an Internet application. It is an online auction web site developed at Rice University to model basic functions of ebay.com system. It is widely used for modeling multi-tier systems [87] [88] [95] [105] [44] [95] [23]. The original implementation of RUBiS uses a variety of open source software products including JBoss, JOnAS, Tomcat, and Apache. In our experiments, we considered the RUBiS implementation consisting of Apache as a web server, Tomcat as an application server, and MySQL as a database.

6.1.1 Physical Setup

To reduce the experiment budget, we installed both the workload generator and the load balancer inside Amazon EC2 infrastructure. Both the web and the application were run on instances bundled to Amazon S3. A write only database (i.e., Master database) was created from an instance mapped to EBS storage for permanent storage, while a read only a database (i.e., Slave database) was created from a bundled image stored at Amazon S3. We chose a small instance for the web, the application, the Slave databases, and the load balancers. For the Master database and the load generator we ran medium instances (i.e., *m1.medium*). To avoid the impact of other tiers on the tier that is under analysis, we created many replicas in the other tiers that keep the CPU utilization in these tiers around 30%. As an example, to model CPU utilization of the web tier, we ran four instances of application tier and two instances of Slave database.

The generated workload was step traffic increasing the number of simultaneous clients gradually. In our experiments, we considered the 95^{th} percentile response time, which means that 95% of the measured response times of all requests is less than or equal to a given value (e.g., 95% of the requests is less than 100 milliseconds).

6.1.2 Web Tier and Application Tier Thresholds

The CPU utilization increases, as the number of requests increases. The relation between the number of requests and the CPU utilization is linear during the experiment run time. On the other hand, the response time increases exponen-

tially with the CPU utilization. At some high values of the CPU utilization, the response time increases dramatically because the requests spend longer time in the queue of the system waiting for processing. Our goal from this analysis is to determine the CPU threshold that keeps the response time within a specific limit. In our system, we consider 100 ms as a higher limit of response time, meanwhile a response time around this value gives the user the feeling that the system is reacting instantaneously [85]. Figure 6.1 demonstrates the following: to keep the response time of 95% of the requests less than or equal to 100 ms, the CPU utilization of each VM instance at web tier and application tier should be less than or equal to 70% and 62

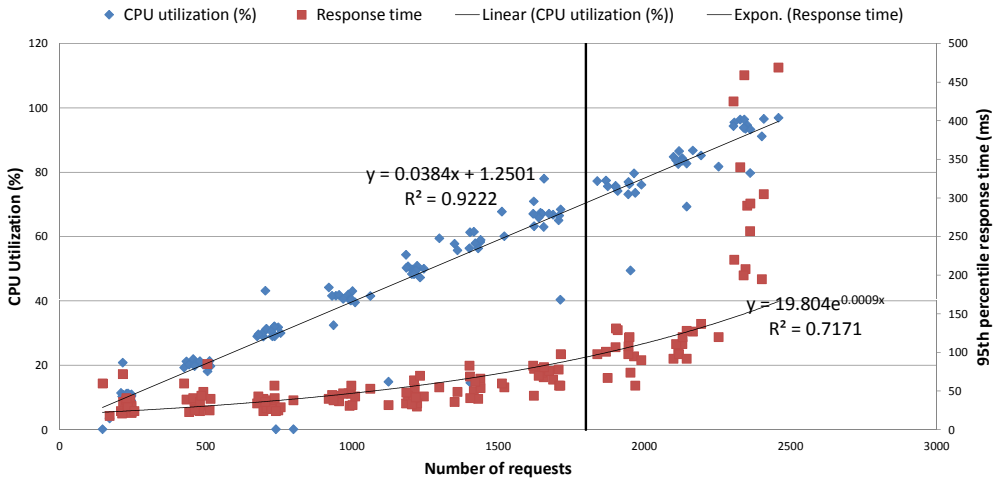


Figure 6.1: Web tier performance threshold

6.1.3 Database Tier Thresholds

Internet applications experiencing a high workload on the database tier should increase the database tier capacity by either scaling up vertically into more complex and expensive hardware server; or scale out horizontally into low-commodity hardware. Wikipedia is one of the big systems that adopted the horizontal scaling approach to the database tier using Master/Slave architecture. Wikipedia receives 50,000 http requests per second that results in 80,000 SQL query per a second [82].

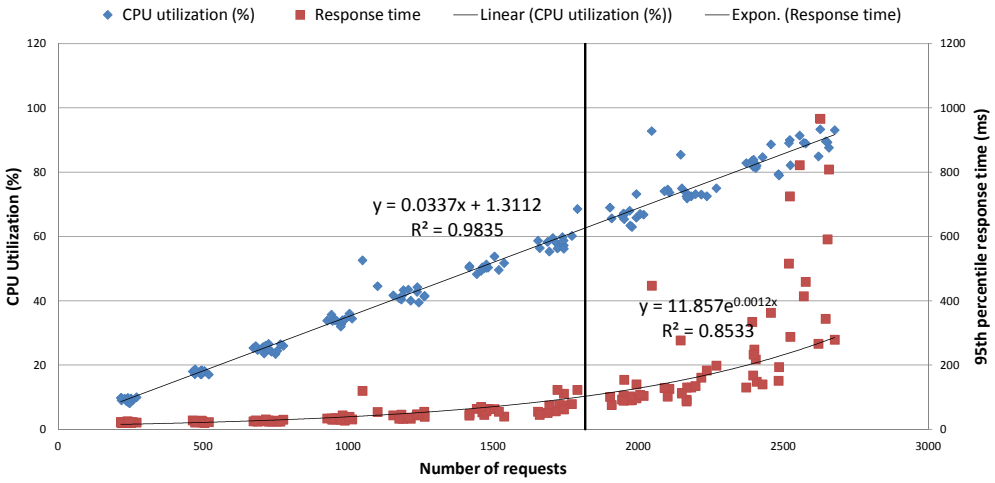


Figure 6.2: Application tier performance threshold

In our setup, we assumed only modeling the Slave instances. To split the workload according to the query type (i.e., write or read-only), we used MySQL proxy [84]. Although MySQL proxy is still in Alpha release, it showed robustness in splitting the queries and balancing the load among the Slave replicas in our experiments.

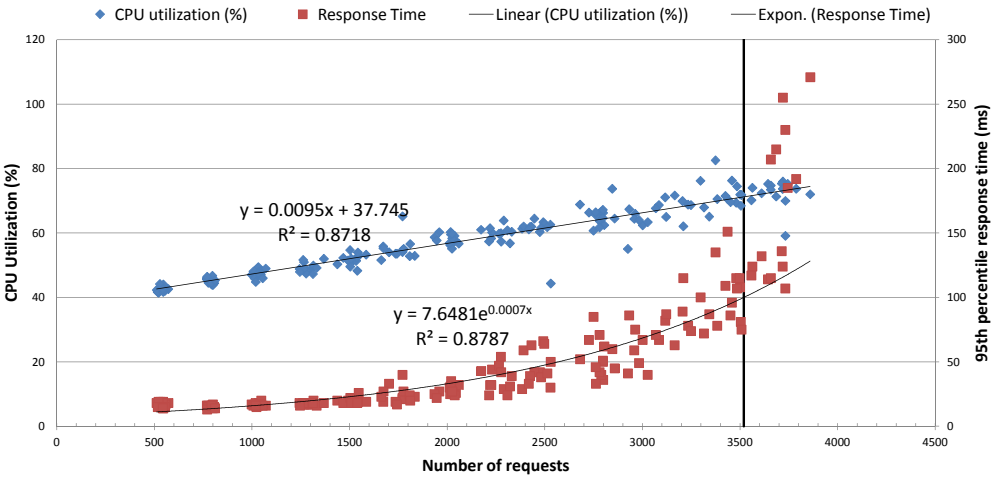


Figure 6.3: Read only (i.e. Slave) database tier performance threshold

For the same setup but with five instances at web tier and four at application tier, the result in Figure 6.3 shows the following: to keep the response time of 95% of the requests less than or equal to 100 ms, the CPU utilization of each

instance of read only databases should be less than or equal to 72%.

Our experiment shows that each tier has a different performance threshold. If the customer failed to determine the proper threshold for each tier, the whole Internet application performance will be degraded. For example, in the case of our setup, the auction website owner (i.e., RUBiS) has a Service Level Objective (SLO) to keep the response time of 95% of the requests less than 100 milliseconds. If the commonly used scale out threshold (e.g., 70%) is set up for the scale out of the application tier, the system will violate the SLO during the periods when the CPU utilization of VM instances at application tier is higher than 62% but less than 70% (i.e., High, but not high enough to scale out).

6.1.4 ARX Models Extraction

In this section, we model the VM instance resource utilization as a black-box. The relation between the input (i.e., URL requests to the Internet application) and the output (e.g., CPU utilization) is inferred empirically. To have online measurements of the VM instance under analysis, we built a Java client continuously monitor the basic metrics: CPU utilization, Memory allocation, Network IN/OUT rate, and Disk read/write rate. In addition to the monitored metrics, we enabled the load balancer to log URL requests. The monitored metrics are synchronized with the request log file. For each monitoring interval (i.e., one minute in our case) we count the number of requests of each type. We consider 18 types of requests in RUBiS benchmark depending on the URL. The vector of requests rate is the input for our multiple-input, single-output (MISO) model, while the output is the modeled metric (e.g., CPU utilization of the web tier). To have accurate models, we had the samples only from the area where the system runs linearly. In other words, we discard the regions where the response time increases unexpectedly. More details about the models extraction and validation are presented in Section 8.3.2.

We start with a single-input, single-output (SISO) ARX model to learn a linear relationship between input u and output y as $y = f(u)$, then we extend the SISO to MISO model. The input u and output y are sampled at time k as u_k and y_k respectively. The input-output relationship can be represented by the

following difference equation:

$$y_k + a_1 y_{k-1} + ... + a_{n_a} y_{k-n_a} = b_1 u_{k-n_k} + ... + b_{n_b} u_{k-n_k-n_b+1} + \epsilon_k \tag{6.1}$$

The parameters n_a and n_b reflect how strongly previous steps affect the current output, while n_k represents the delay between the effective input u_k and output y_k. For instance, $n_k = 0$ means a direct coupling between input and output. A compact way to write the difference equation is:

$$A(q)y_k = B(q)u_k + \epsilon_k \tag{6.2}$$

If we consider q as a delay operator, we can interpret $A(q)$ and $B(q)$ as follows:

$$A(q) = 1 + a_1 q^{-1} + ... + a_{n_a} q^{-n_a} \tag{6.3}$$

$$B(q) = b_1 q^{-n_k} + ... + b_{n_b} q^{-n_k-n_b+1} \tag{6.4}$$

The white noise term ϵ_k is usually small for a model with high fitness score, so we can extract the adjustable parameters to the following:

$$\theta = [\ a_1\ a_2\ ....\ a_{n_a}\ b_1\ b_2\ ...\ b_{n_b}\ ]^T \tag{6.5}$$

If we define a column φ_k as follows:

$$\varphi_k = \left[\ -y_{k-1}\ ....\ -y_{k-n_{an}}\ u_{k-n_k}\ ...\ u_{k-n_k-n_b+1}\ \right]^T \tag{6.6}$$

Then, the estimator $\hat{y}_k$ of y_k can be calculated as follows:

$$\hat{y}_k = \varphi_k^T \theta \tag{6.7}$$

If we have N measurements of input u_k and output y_k, then the goal is to find

θ that results in the lowest quadratic error:

$$\epsilon = \frac{1}{N}\sum_{k=1}^{N}(\hat{y}_k - y_k)^2 \tag{6.8}$$

Using Least Squares Method (LSM) for θ, we can find $\hat{\theta}_k$ that minimizes the estimated error ϵ as follows:

$$\hat{\theta}_k = \left[\frac{1}{N}\sum_{k=1}^{N}\left(\varphi_k\varphi_k^T\right)\right]^{-1} f\left(N\right) \tag{6.9}$$

where:

$$f\left(N\right) = \frac{1}{N}\sum_{k=1}^{N}\varphi_k y_k \tag{6.10}$$

The SISO model shown in equation 6.1 can be extended to the MISO model, which considers C categories of requests. Each category rate results in a different consumption of resources. If we refer to category i of requests as u^i, then we should derive the new relationship $y = f(u^1, u^2, ..., u^C)$. In this case, the coefficients of request i can be presented as $\left[b_1^i\ b_2^i\ ...\ b_{n_b}^i\ \right]$. The equation 6.1 is updated to consider multiple inputs as follows:

$$\begin{aligned} y_k + a_1 y_{k-1} + ... + a_{n_a} y_{k-n_a} \\ &= b_1^1 u_{k-n_k}^1 + ... + b_{n_b}^1 u_{k-n_k-n_b+1}^1 \\ &+ ... \\ &+ b_1^C u_{k-n_k}^C + ... + b_{n_b}^C u_{k-n_k-n_b+1}^C \end{aligned} \tag{6.11}$$

Accordingly, new values of θ and φ that consider multiple inputs are as follows:

$$\theta = \left[\ a_1\ a_2\ ....\ a_{n_a}\ b_1^1\ b_2^1\ ...\ b_{n_b}^1\ .....\ b_1^C\ b_2^C\ ...\ b_{n_b}^C\ \right]^T \tag{6.12}$$

$$\varphi_k = [\ -y_{k-1}\ ....\ -y_{k-n_{a_n}}\ u^1_{k-n_k}\ ...\ u^1_{k-n_k-n_b+1}$$
$$.....$$
$$u^C_{k-n_k}\ ...\ u^C_{k-n_k-n_b+1}\ ]^T \tag{6.13}$$

After extracting θ parameter and defining φ_k for multiple inputs model, we can use equations 6.7 to 6.10 of SISO model to find $\hat{\theta}_k$ that minimizes the estimated error ϵ of MISO model.

In our experiments, we observed that setting $n_a = 2$ and $n_b = 2$ reduced the parameters search space without degrading model's accuracy. Moreover, we did not consider any time delay by setting n_k value to zero while the sampling window size is 60 seconds, which is long enough to hide small delays of request impact on each tier. Typically, common least square algorithms have polynomial time complexity $O(u^3v)$ when solving v equations with u variables [119]. However, we solve these equations once at off-line time for each resource. At run time, we calculate $\hat{y}_k$ with a linear complexity $O(n)$, where $n = n_a + n_b * C$.

6.2 Developed Simulator (ScaleSim)

Our simulator (ScaleSim) is built on top of the CloudSim [29]. CloudSim [29] is a framework for modeling and simulating cloud computing infrastructures and services. It allows a fast evaluation of the new developed algorithms. Nevertheless, it does not contain the scaling components that are required for our research; therefore we implemented our components as external modules interacting with the core of the CloudSim.

We implemented ScaleSim to examine the current implementation of the scalability in Amazon EC2. Our simulator was built as components that can be easily customized by other researchers. Moreover, we depend on meta-data files (i.e., xml files) for configuration. With each new run, the components fetch the last update for the configuration file. Figure 6.4 shows the components of our developed simulator. The interaction between the components is explained as follows:

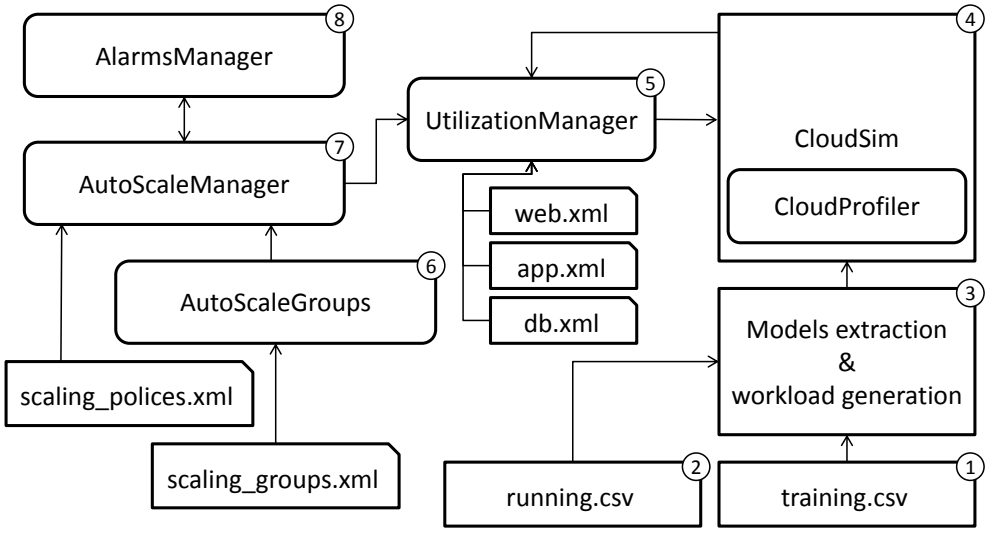


Figure 6.4: Our implementation for cloud computing scalability (ScaleSim)

1. Training file (e.g., *training.csv*) contains real measurements of an application in a physical environment. The measurements consist of the rate of each considered URL request and the utilization of the monitored resources. The file is built as described in Section 6.1.4. However, the same procedures can be implemented to any other Internet application.

2. Running file (e.g.,*running.csv*) contains an artificial generated workload. To achieve more realistic results, the workload should be generated to mimic the real behavior of the Internet applications. We explain in details the workload generation for our experiment in Section 6.2.1.

3. Models extraction and workload generation modules do two tasks: first, it reads the training files to extract models. Second, it calculates the consumption of resources. The expected consumption of the resources is passed to *CloudProfiler* object. It is an object resides in CloudSim to build the data centers and the brokers that manage the coming workload.

4. In CloudSim we implemented a simple Datacenter to avoid internal optimization of resources (e.g., VM migration) that might influence our simulation results. At the start of the simulation, a new object of Utilization-Manager is created.

5. Whenever *UtilizationManager* is started, it creates an object of *AutoScaleManager* class. In fact, *UtilizationManager* is considered as the actuator for the scalability commands that are received from *AutoScaleManager*. *UtilizationManager* has a direct monitoring of resources in CloudSim environment. It passes these measurements to *AutoScaleManager* that decides about scaling out or down to cope with incoming workload. Usually, starting a new VM commands are passed to CloudSim including the profile of the VM image to be started. In our simulator, the VM profile also includes the total required time to put a VM to the operational mode.

6. *AutoScaleGroups* is an object implementing the same concept of the scalability groups in Amazon EC2. It maintains the number of the VMs in a group to the number predetermined by the Internet application owner. For example, it can be configured to guarantee that the minimum number of instances at each tier is one. Moreover, it can be configured to prevent the number of instances in a specific tier from exceeding a pre-defined number of instances.

7. *AutoScaleManager* is the component that is responsible for the scalability algorithm intelligence. In case of reactive scaling (e.g., current implantation in Amazon EC2), the scalability is controlled by both the scaling policies as input from the application owners (i.e., *scaling_polices.xml*) and the AutoScaleGroups. To employ proactive scaling algorithm, *AutoScaleManager* can be developed to consider historical measurements for coming workload prediction, as we present in Chapter 7.

8. *AlarmsManager* is a queue receiving a continued stream of alerts. The alerts are initiated at *AutoScaleManager* whenever the utilization matches any of scaling polices. Each alert contains attributes (e.g., timestamp, scaling group, scale direction, and evaluation periods) that help the *AlarmsManager* manager to group the alerts and pass the scalability decision to *AutoScaleManager* at the proper time.

6.2.1 Workload Generation

To simulate a realistic large-scale running of an Internet application, we need both a real request arrival rate and a real user behavior (e.g., flow pattern and thinking time) to keep results consistent [80]. RUBiS benchmark has flow probability matrix M which is NxN matrix describes N states of the system. Each element of the probability matrix (i.e., Mij) describes the probability that the workflow j follows workflow i [96]. According to probability matrix, a website receives a specific percentage of each request type [30]. Similarly, we calculate the probability of the appearance (i.e. the requests rate) of each request type.

To mimic a realistic arrival rate of users and workload variation, we used the world cup 1998 workload [77] traces. They are apache log style traces of 1.35 billion requests initiated to the world cup 1998 official website for over three months period. For each period of time (i.e., one minute in our case) we multiply the number of requests by the probability of each of the RUBiS benchmark requests. The result is the rate of each considered request of RUBiS benchmark (i.e., 18 requests in our case), which is stored in *running.csv* file, as explained in Figure 6.5. The rates vector for each time window is used to calculate the consumption of resources at each tier.

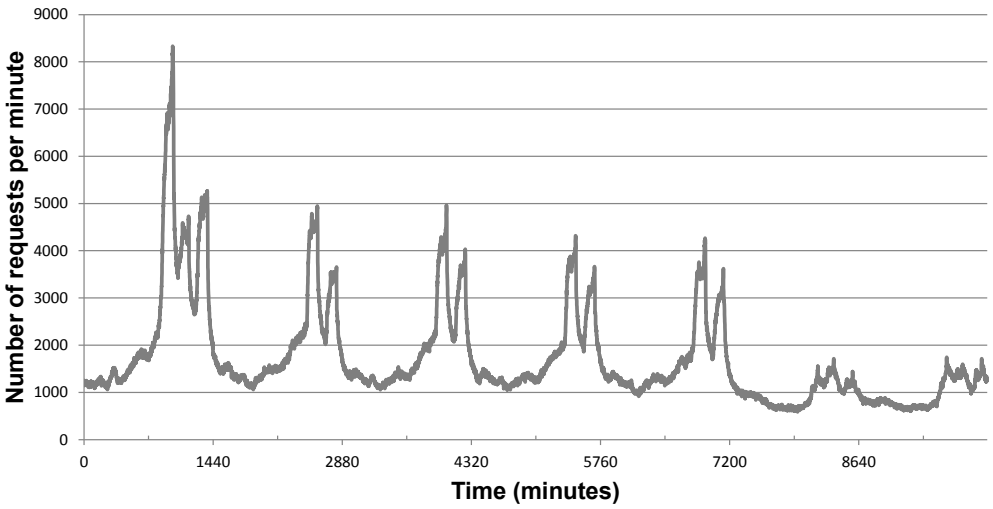


Figure 6.5: The rate of requests to the official website of the world cup 1998 for one week started at June 15th, 1998

6.2.2 Evaluation

The simulation is run with the parameters described in Table 6.1. The explanation of each parameter is shown in Table 5.2.

Parameter	Web tier	App tier	Db tier
cooldown	300 seconds after scale out or down	300 seconds after scale out or down	300 seconds after scale out or down
adjustment	1 for scale out, -1 for scale down	1 for scale out, -1 for scale down	1 for scale out, -1 for scale down
metric-name	CPU utilization	CPU utilization	CPU utilization
threshold	70% for scale out, 30% for scale down	62% for scale out, 30% for scale down	72% for scale out, 30% for scale down
period	1 minute	1 minute	1 minute
evaluation-periods	5	5	5

Table 6.1: The parameters used for the simulation results seen in Figure 6.6

As shown in the Table 6.1, we used a different scale out threshold for each tier depending on the observations in Sections 6.1.2 and 6.1.3. The considered metric in our simulation is the CPU utilization. The scale out or down step size is one VM instance for all tier. The CPU utilization is compared with the threshold every minute. However, the scale out or down is only triggered when the threshold is exceeded five times in sequence. After a scale out or down, the system counts five minutes before accepting a new scaling trigger.

In addition to parameters in Table 6.1, we consider the initialization time of the VM instance. In all tiers, we consider the small Linux textitm1.small VM instance described by Amazon EC2 [12]. Mao et al. [76] measured the initialization time of Linux instances at Amazon EC2 to be 96.9 seconds in average, meanwhile it is measured to be 44.2 seconds in RackSpace in average. In our experiments, we used 60 seconds interval as the initialization time of instances. Clearly, the initialization time value mainly depends on the adopted technology by the provider. So, in our simulator, we keep it as a modifiable parameter associated with the VM type. Nevertheless, as we explain in Section 5.2.2, the

database instance initialization may require longer time compared with web and application tiers. In this experiment, we consider the required time to bring a database VM instance to operational mode is 10 minutes.

The result of running the simulation with the parameters in Table 6.1 is seen in Figure 6.6 .

Figure 6.6 shows simulating our Internet application with the parameters in Table 6.1. It shows *Aggregated CPU utilization*, which is the sum of the CPU utilization (i.e., the workload) of all instances running in a specific tier (i.e., the workload); and *Available CPU Capacity*, which is the accumulated CPU capacity by all instances running in a specific tier.

The *Available CPU Capacity* is calculated by the following formula:

$$\textit{Available CPU capacity} = \textit{Number of VMs} * \textit{the performance threshold} \quad (6.14)$$

For example, if five instances are running in the application tier, the *Available CPU capacity* $= 5 * 62 = 310$. We consider the performance threshold value, because any CPU utilization higher than this value results in SLO violation.

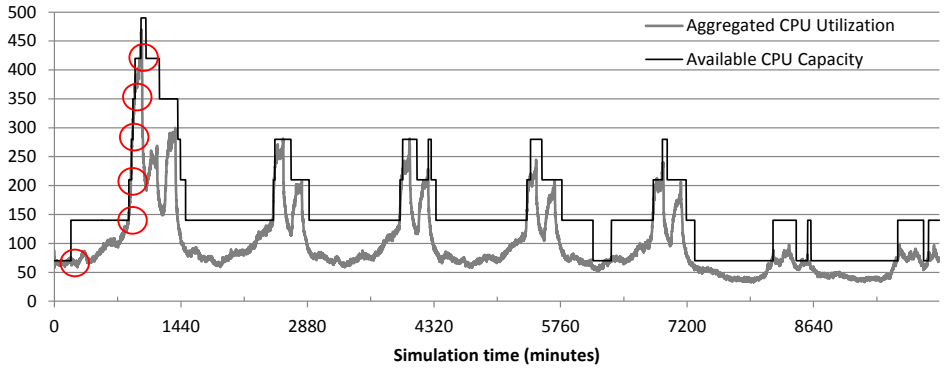
Our goal is to keep the *Available CPU Capacity* always higher than the *Aggregated CPU utilization*. However, Figure 6.6 shows some moments when the *Aggregated CPU utilization* coincides or goes the *Available CPU Capacity*. We marked these moments for the first day simulation. These moments record SLO violation. The percentage of the SLO violation time to the total run time of the system is presented in Table 6.2.

Tier	**Web**	**Application**	**Database**	**All (total)**
Cost ($)	28.56	21.36	14.80	64.72
SLO violations (%)	1.9	2.4	1.39	5.29

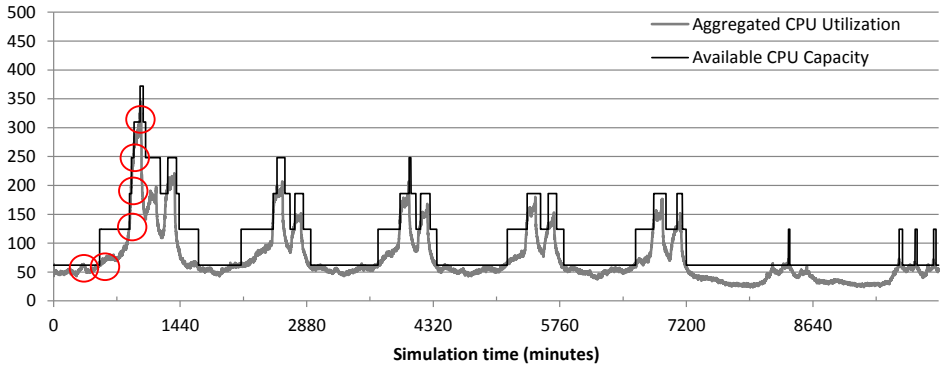
Table 6.2: Results of the simulation described in Section 6.2.2 for one week

A multi-tier system behaves as a network of queues [103], so the delay in one tier influences the response time of the whole system. Therefore, we record a violation whenever any of the three tiers violates the SLO.

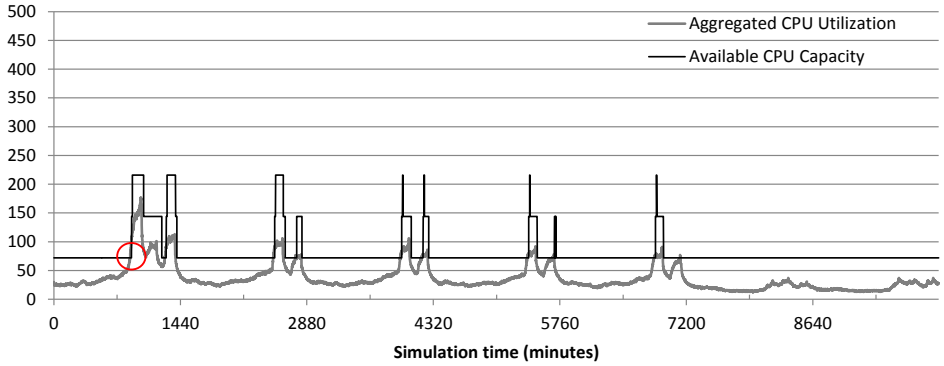
In addition to the SLO percentage, in Table 6.2, we calculated the total cost of running VM instances at each tier. The cost was calculated by counting the



(a) Web tier dynamic scalability simulation



(b) Application tier dynamic scalability simulation



(c) Database tier dynamic scalability simulation

Figure 6.6: Simulating scalability of the multi-tier system with the parameters in Table 6.1

running hours of each instance multiplied by the price of the *m1.small* instance running at Amazon EC2 east-coast data centers, which is $0.08 at the time of writing this dissertation. However, the price and the parameters of the instances

can be easily modified in our developed simulator. As same as Amazon EC2, we charges the partial hours as full hours. Moreover, we observed that whenever the *Scaling groups* in Amazon EC2 initiates a termination command for an instance to scale down, the instance with the longest runtime is terminated. We use the same selection way in our simulator. However as a future work, we plan to study optimizing the cost by terminating instances that are more close to the end of the charging unit (i.e., one hour) of the on-demand instances.

According to the scalability policy in Table 6.1, whenever the CPU utilization of the database tier goes over than or equal to 72%, the system starts a scale out to prevent the SLO violation. However, the scale out is not triggered before five evaluation periods of CPU utilization, as set in Table 6.1. After triggering a scale out, the system goes into the sequence described in Section 5.2.2, which is measured as 10 minutes. So, with the current setting, database is supposed to contribute mostly to the SLO violation in the simulated applications. Nevertheless, the violation caused by database tier is less than the violations by web tier and application tier. The results are explained as the following:

1. Due to our workload, there were few scales per a day (i.e., two at most) in the database tier compared to the web and the application tier.

2. Due to parameters shown in Table 6.1, each scale out in the web tier or the application tier cannot be accomplished in less than 6 minutes calculated as 5 minutes (as a control period time) plus one minute (as the VM instance's initialization time) .

According to our experience, implementing dynamic scalability to the relational databases is not common in production environments. Most likely, this is due to the sensitivity of the database tier whereas any corruption or missing of the data can be harmful for the whole business. Furthermore, huge databases require longer times to create read only replicas dynamically. So, in the next chapter, we consider running an adequate number of the database replicas in the database tier all the time, and concentrate our analysis to the web and application tiers.

6.3 Summary

Our main goal of this chapter is to evaluate the current implementation of the scalability in the public IaaS (i.e., Amazon EC2). However, the large-scale experiments in the cloud are costly. On the other hand, the small test-beds and the prototypes may not express the Internet application behavior at large-scale. To overcome these issues, we built our simulator (ScaleSim). The ScaleSim simulates the scalability components of Amazon EC2. To achieve a realistic simulation results, we modeled the tiers of the Internet application in the physical environment. Then, we fed the extracted models to the simulator. The modeled application in this chapter was RUBiS benchmark, and the initiated workload was a workload generated from the access logs of the 1998 world cup website. With this setup, the SLO violations percentage is measured as 5.29% of the total system run time. In the next chapter, we study reducing SLO violations to improve the QoS.

Chapter 7

Reactive v.s. Proactive Scalability in IaaS layer

In the previous chapter, we evaluated running an Internet application at large-scale level. The scalability is implemented using static thresholds, which is a commonly used technique in the IaaS production environments. The static thresholds enable the application to scale dynamically and cope with the workload variation. Nevertheless, the results in the previous chapter show periods of the SLO violation with each sudden increase of the workload. The SLO violation occurs due to the resources provisioning overhead, as we explained in Section 5.2.

In this chapter, we propose two approaches for mitigating the impact of the overhead of initializing resource on the scalable Internet applications' performance. The proposed approaches are customer-oriented solutions contrary to many other approaches [117] [54] [95] [94] [63] [21] considering the provider involvement in the application performance management. Being customer-oriented solutions makes our approaches feasible in current IaaS environments.

The first approach assumes keeping the current reactive scalability and only tuning the *scalability thresholds* for a better performance. The second approach considers a proactive scalability to provision resources in advance and reduce the SLO violations. Both approaches exploit the trade-off between the cost and the performance. As in the previous chapter, the SLO is to keep the response time less than 100 milliseconds.

In Section 7.1.1 we study the impact of tuning scale thresholds on the system performance and the total cost. In Section 7.2, we study the proactive scalability using several prediction algorithms and determine the pros and cons of each algorithm. In the same section, we develop our prediction algorithm to overcome the drawbacks of the other prediction algorithms. In Section 7.3, we compare the performance of our proposed prediction algorithm with the other prediction algorithms. Finally, we wrap up our contributions in Section 7.4.

7.1 Reactive Scalability

7.1.1 Scalability Parameters Tuning

In this section, we study tuning the *scalability thresholds* of an application to mitigate the impact of resource provisioning overhead. As a start, we should distinguish between two terms used frequently in this chapter: First term is the *performance threshold*, which is the threshold after which the system performance degrades dramatically (i.e., SLO is violated with high probability). Second term is the *scalability threshold*, which is the threshold after which the system will scale out or scale down regardless of the performance. For example, as shown in Section 6.1.2, web tier in RUBiS benchmark has 70% CPU utilization as a *performance threshold* and *scalability thresholds* as well. However, we try different values of *scalability thresholds* to find the best performance with the lowest cost.

In Figure 7.1, we assume that 70% is the *performance threshold*. If we pick up a scale out threshed lower than the *performance threshold* by Δh, we can increase the probability of scale out before approaching the *performance threshold*. This implies longer periods of over-provisioning and therefore increases the cost. On the other hand, a higher scale out threshold exposes the VMs to higher values of CPU utilization ranges between 70% and 70%+ Δh, which can lead to SLO violation. Nevertheless, a higher scale out threshold implies running less VMs and a lower total cost. Same concept is applied to scale down threshold. A very low scale down threshold will keep more running VMs, which increases the cost, but allows the system to cope better with the workload increase. On the other hand, a very high scale down threshold will increase the probability of turning

off VMs (i.e., reduces the cost), but will expose the system to under-provisioning whenever the workload increases.

So, the question is, what are the best values for Δh and Δl to reduce the SLO violation and do not increase the cost dramatically? We consider this as a multi-objective optimization problem.

Figure 7.1: *Scalability threshold* values as a trade-off between cost and performance

Our optimization problem has two goal functions: G_c, that represents the minimum cost and G_v, that represents the minimum SLO violation. An optimal decision need to be taken in the presence of trade-offs between the two goals. However, building mathematical model of our optimization functions is very complex. As a solution, we first convert the original multi-objective function into many single-objective optimization problems, then we compute all representative set of Petro optimal solutions [46] [35]. Therefore, we depend on the simulation as a method to calculate y_c, the feasible cost for an input x, and y_v, the feasible SLO violation percent for input x, where x is a subset of I.

The set I is the possible inputs. In our case, possible inputs for scale out threshold is a range of integer values including the *performance threshold* as a median. Fortunately, these inputs are limited by the computing resources nature. For example, CPU utilization ranges from 0% to 100%. Moreover, as seen in

Section 6.1, we observe that high CPU utilization values, such as 80% or higher, are harmful to the system performance, which reduces the range of the possible inputs.

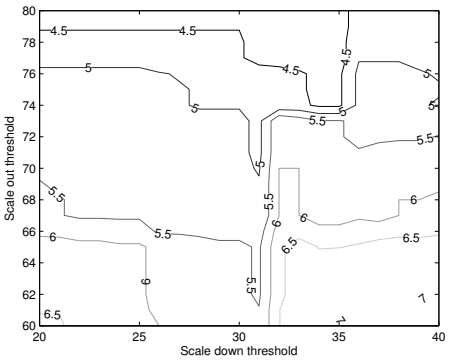
Generally, the scale out and scale down thresholds ranges should not overlap. In our experiments, we consider 30% as the median value for the scale down threshold range, and a value close to *performance threshold* as the median for scale out threshold range.

Since we have different parameters in the system, we fix them all and only vary one parameter per a simulation. As an example, with fixed *scalability thresholds* of application tier (i.e., 62%, as a scale out threshold and 30%, as a scale down threshold) we evaluate a range of a scale out thresholds for web tier starts at 60% and ends at 80%. For each value we run a complete simulation and calculated the cost and the number of SLO violations. To be sure about the observations consistency, we run individual simulation for the first three days in our generated workload described in Section 6.2.1. Moreover, we tried different values of scale down threshold ranges between 20% and 40%. So, the x-axis of all sub-figures 7.2(a) to 7.2(f) is the tested values for scale down threshold, while y-axis is the tested values for scale out threshold. The same setup is repeated to the application tier. With fixed *scalability thresholds* of the web tier (i.e., CPU utilization 70%, as a scale out threshold and 30%, as a scale down threshold) we evaluate a range of a scale out thresholds for application tier starts at 50% and ends at 70%. For the scale down threshold, we tried different values range between 20% and 40%. The results of the described settings are seen in Figure 7.3. In all experiments, we consider adequate number of database instances (i.e., three instances), which prevents any SLO violation by database tier.

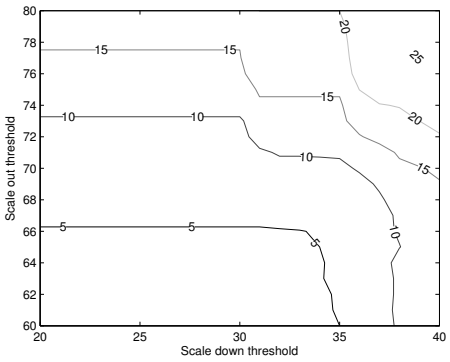
The SLO violation describes the percentage of time where the response time of the Internet application (95^{th} response time) is probably higher than 100 milliseconds. It is calculated by finding the percentage of the number of minutes at which the measured CPU utilization is higher than the *performance threshold* to the total run time (i.e., 1440 minutes).

From Figures 7.2 and 7.3, we have the following observations:

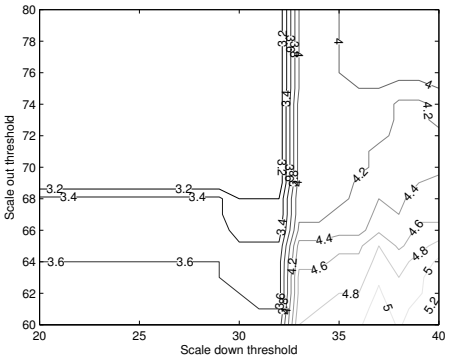
1. Scale out threshold tuning:



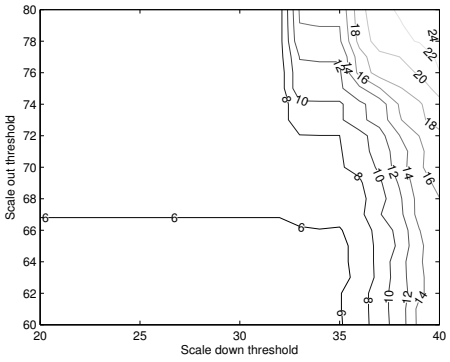
(a) Cost ($) - First day



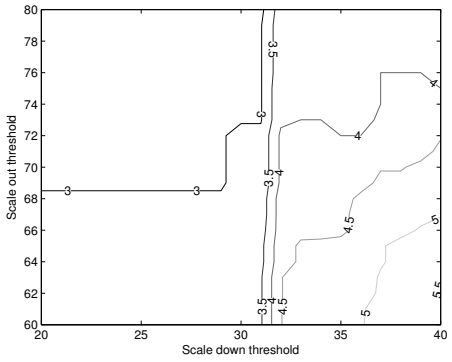
(b) SLO violation (%) - First day



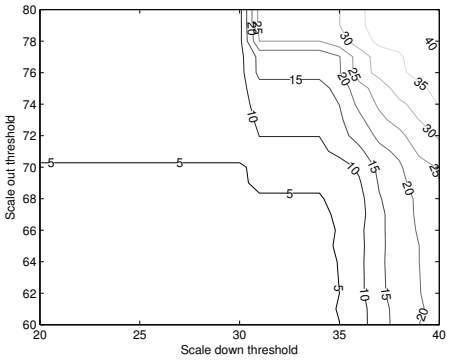
(c) Cost ($) - Second day



(d) SLO violation (%) - Second day

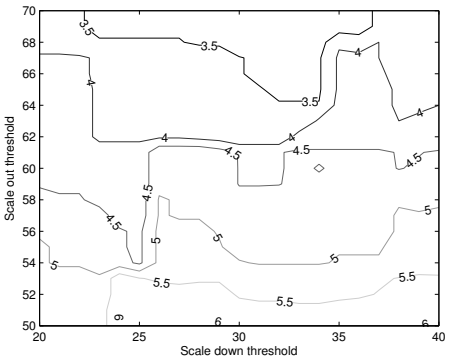


(e) Cost ($) - Third day

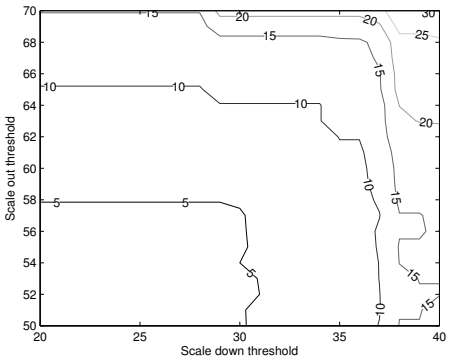


(f) SLO violation (%) - Third day

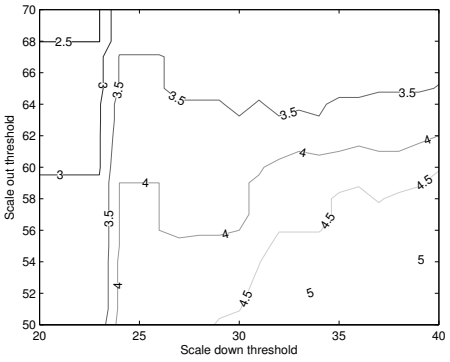
Figure 7.2: The impact of *scalability thresholds* on cost and SLO violations at the web tier. The *performance threshold* of web instances is 70%



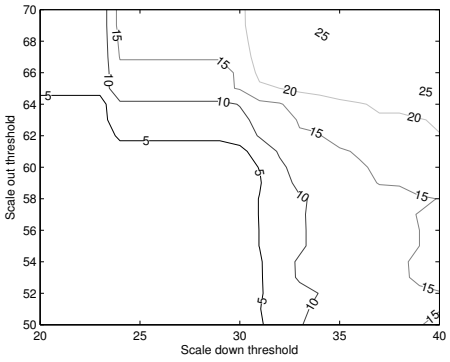
(a) Cost ($) - First day



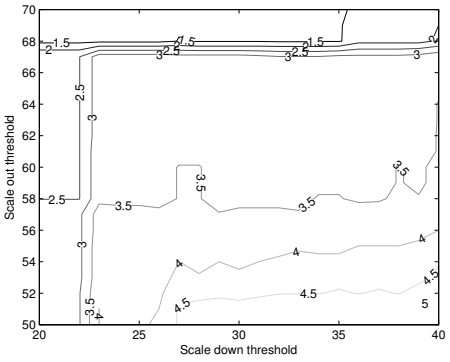
(b) SLO violations (%) - First day



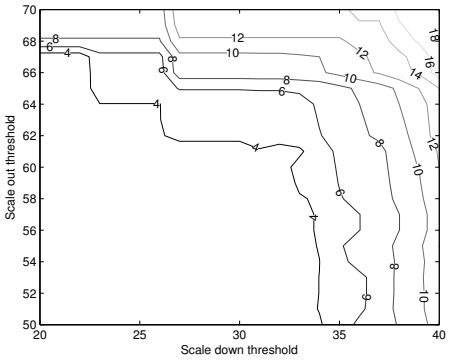
(c) Cost ($) - Second day



(d) SLO violation (%) - Second day



(e) Cost ($) - Third day



(f) SLO violation (%) - Third day

Figure 7.3: The impact of the *scalability thresholds* on cost and SLO violations at the application tier. The *performance threshold* of the application instances is 62%

- A scale out threshold higher than the *performance threshold* decreases the cost slightly, but increases the SLO violation.
- A scale out threshold lower than the *performance threshold* increases the cost slightly but reduces the SLO violation strongly.
- A very low scale out threshold increases the probability of over-provisioning which increases the cost but without remarkable decrease in SLO violation. However, if a very low scale out threshold coincides with a high scale down threshold the probability of oscillating increases, as we will discuss in Section 7.1.2.

2. Scale down threshold

- A high scale down threshold results in a high violation of the SLO. However, it does reduce the cost.
- A very low scale down threshold does not reflect an increase in the cost as expected in Figure 7.1. However, it reduces the SLO violation.

According to our observations, we suggest the following *scalability thresholds* for our application:

- For scale out, picking a value slightly lower than the *performance threshold* reduces the probability of SLO violation. For example, we select 66% as a scale out threshold for the web tier, while the *performance threshold* is 70%. On the other hand, we select 58% as a scale out threshold for the application tier, while the *performance threshold* is 62%.
- For the scale down, any value less that 30% keeps the SLO violation to the lowest if the scale out threshold is set as in the previous observation.

The impact of the *scalability thresholds* tuning on both cost and performance will be evaluated empirically in Section 7.3.

7.1.2 The Impact of Selecting Bad Thresholds

Looking for less SLO violations, IaaS customers may select a very low scale out threshold value. However, this behavior may not lead to the expected improvement. Figure 7.4 shows our application run for three days. The aggregated CPU

is the summation of the CPU utilization of the VM instances in the application tier. The available CPU capacity = Number of VM instances * the *performance threshold*. The figure shows how a very low scale out value leads to an oscillating in the number of the provisioned VM instances, which leads to short but chargeable runs of VMs instances. This explains the early increase of the cost seen in Figures 7.3(a) to 7.3(e) for scale out threshold 51% and scale down threshold 30%.

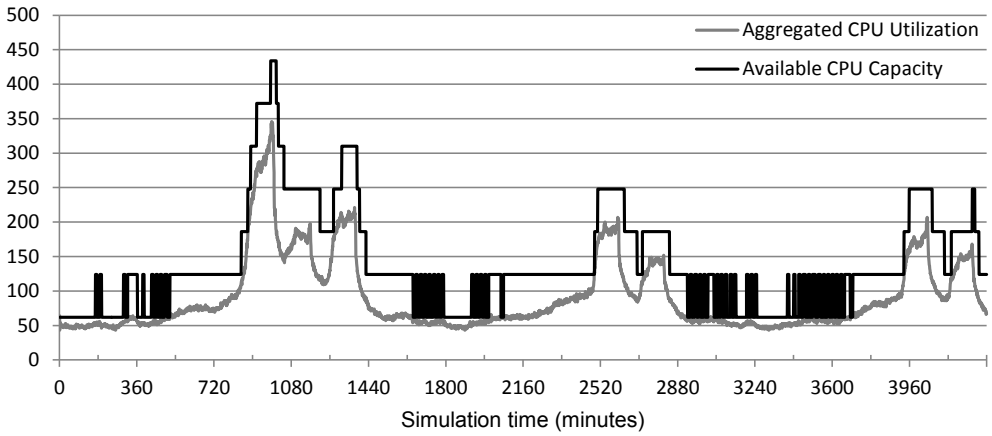


Figure 7.4: The bad *scalability thresholds'* impact on the performance of the application tier

What also increases the probability of the oscillation is the workload itself. For instance, we see in Figure 7.4 that most of the oscillation happens when the *aggregated CPU utilization* of VM instances in the application tier oscillates around 51%. The oscillation is due to the following: when the CPU utilization is a little higher than the scale out threshold (e.g., 53%); the system will scale out to two instances. For the next control period of time, the CPU utilization per an instance will be (53/2 = 26.5, which is less than 30%). According to the scale down policy, the system should start a scale down process causing the system under-provisioning and starting another scale out process. Similarly, for web tier, we observe many periods of *aggregated CPU utilization* oscillate around 66%, which makes selecting close values of scale out and scale down thresholds (e.g., 66%, as scale out threshold, and 33%, as scale down threshold) a bad choice, as seen in Figures 7.2(a) to 7.2(e). One solution is to increase the *cooldown*

parameter described in Table 6.1. However, a big *cooldown* value delays the system response to the spikes in workload and may results in longer periods of under-provisioning (i.e., SLO violation). Nevertheless, a higher *cooldown* may reduce the oscillation but will not eliminate it totally.

According to our observations, we can emphasize considering the following rules to reduce the probability of resource provisioning oscillation:

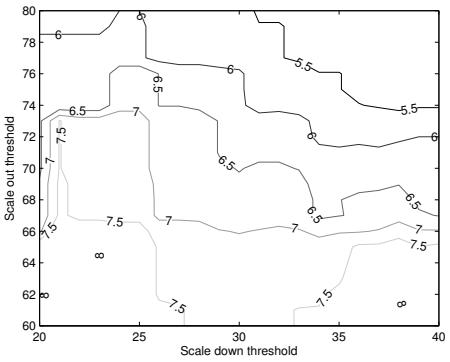
- Avoiding very low scale out thresholds that increase the cost dramatically without a remarkable reduction in SLO violation.
- Avoiding very high scale down threshold that increases both the cost and the SLO violation.
- Avoiding scale out or down thresholds (i.e., scale_threshold) that satisfy the following relation for long periods of the monitored metric (e.g., aggregated CPU utilization): **mean(aggregated CPU utilization) = n * scale_threshold**, where **n** is a positive integer.

7.1.3 The Impact of Scalability Step Size

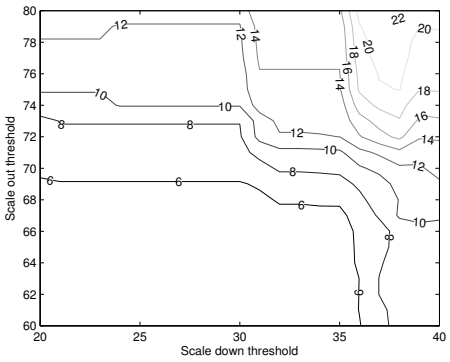
One of the proposed solutions to cope fast with the workload increase is to increase the scaling step size. For example, instead of scaling out gradually (i.e., one VM instance per the scale), the system can be configured to scale out with two or more instances per a scale. At first glance, a bigger scaling step size looks as an efficient solution for mitigating the impact of the resource initialization overhead on the application performance. Therefore, we dedicate this section to exploit using the scale out step size for improving the system performance, and also analyze its impact on cost. Towards this goal, we repeat the last simulation of web tier but with different *adjustment* values. The *adjustment* parameter determines the scaling step size, a described in Table 5.2.

By experiments, we noticed that a fast scale down has a severe impact on the SLO violation. So, we only examined different values for the scale out step size (i.e., only positive *adjustment*).

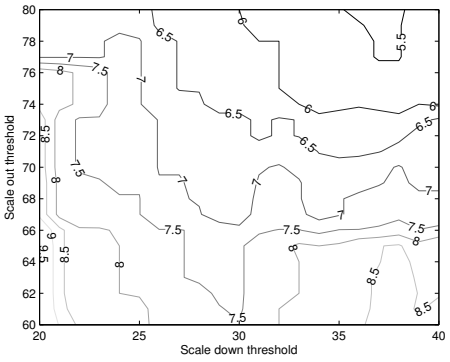
Because Figures 7.2(a) and 7.2(b) already present web tier scale out, we only repeat the web tier simulation with *adjustment* values one, two, three, and four



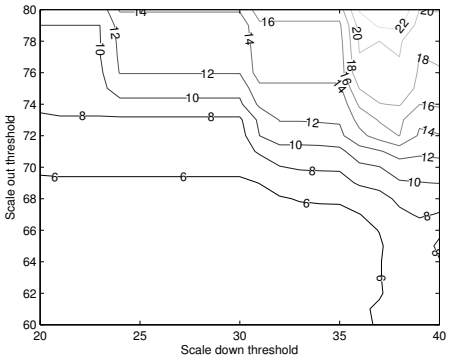
(a) Cost ($) - Two instances



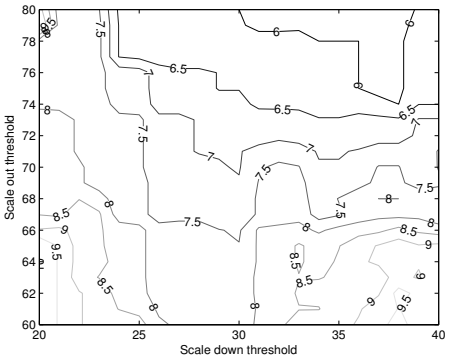
(b) SLO violation (%) - Two instances



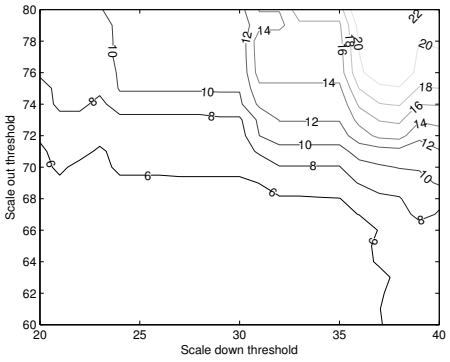
(c) Cost ($) - Three instances



(d) SLO violation (%) - Three instances



(e) Cost ($) - Four instances



(f) SLO violation (%) - Four instances

Figure 7.5: The impact of the scale out step size on the cost and the SLO violations at the web tier

for the first day, with one VM per scale out. The cost and SLO violation of each case are depicted in Figure 7.5. The results show no reduction in SLO violation, but an increase in the cost for the *adjustment* values two, three, and four. The scale out threshold for web tier is 66% and the scale down is 30%, in all experiments. In Table 7.1.3, we present the scale out step size, the cost, and the SLO violations for the first day's simulation.

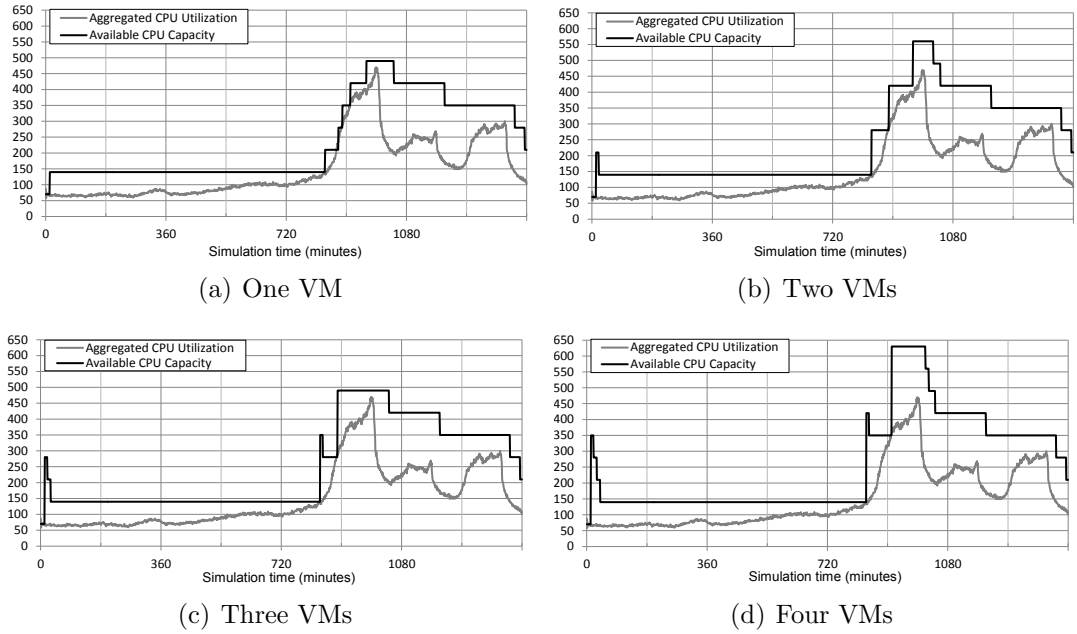
Step size	Cost ($)	SLO violation (%)	Cost increase (%)	SLO violation decrease (%)
One	5.36	1.67	N/A	N/A
Two	6.96	1.18	29.85	29.34
Three	7.12	1.18	32.84	29.34
Four	7.43	1.11	38.62	33.52

Table 7.1: For the scaling thresholds 66% and 30%, we calculate the cost and the SLO (%) violation for different scale out step sizes (i.e., *adjustment*)

The results in Table 7.1.3 show 29% reduction in the SLO violation when using 2 VM for each scale out step, and almost the same percentage (i.e. 30%) increment in the cost. For the scale out step values three and four, the reduction in the SLO is tiny compared to the increase in the cost. To analyze the results, we plot the system scalability for each scale step size in Figure 7.5.

In Figure 7.6(a), when *adjustment* = 1, we recognize four scales out, in addition to the first scale out at simulation start up time. Due to the overhead of initializing new instance, the SLO is violated until initiating adequate number of VMs. In Figure 7.5(c) the scale step size is increased to two VM instances per the scale out, which reduces the total SLO violation periods to two. We can recognize periods of over provisioning that explain the increase in the cost seen in Figure 7.5(a). In Figures 7.6(c) and 7.6(d), the step size is increased to three and four, respectively. However, we notice the same number of SLO violations, but extra periods of over-provisioning, which explains why we cannot recognize a significant decrease in the SLO violation in Figure 7.5(d) and Figure 7.5(f), but an increase in the total cost due to over-provisioning.

Our results express that increasing the step size is not an efficient way to mitigate the overhead of resources initialization on the application performance.



(a) One VM

(b) Two VMs

(c) Three VMs

(d) Four VMs

Figure 7.6: Scale out step size impact on cost and SLO violations at the web tier for the first day

It increases the cost without a significant decrease in the SLO violation.

7.2 Proactive Scalability

In the previous section, we studied tuning *scalability thresholds* of the reactive controllers to reduce SLO violation without modifying the current implementation of the cloud infrastructure. Tuning scalability parameters improved the reactive scalability performance. Nevertheless, in this section, we employ the proactive scalability looking better performance of Internet applications.

7.2.1 Time Series Forecasting Algorithms

Implementing a proactive scalability requires a forecasting algorithm predicting coming workload and provisioning adequate resources in advance. In this section, we study several forecasting algorithms looking for the fit algorithm to our need. Practically, we need an algorithm scaling up fast with the workload in-

crease, but in the other hand scaling down slowly to avoid terminating instances quickly, which can lead to resources under-provisioning. The studied time-series forecasting algorithms are as the following:

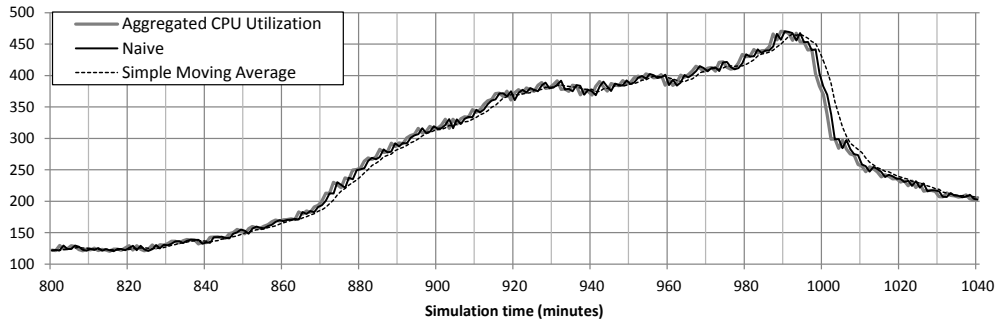
Naive algorithm: It predicts a next value equals to the previous value: $u_{t+1} = u_t$. The algorithm results in a lag between the current and the predicted measurment. Accordingly, it does not help coping fast with the workload increase. Nevertheless, the lag can be useful in the case of the scale down to reduce the probability of under-provisioning, as we explain in Section 7.2.2.

Simple Moving Average algorithm: It calculates the next value u_{t+1} as the average of the recent n values: $u_{t+1} = \frac{1}{n}\sum_{k=0}^{n-1} u_{t-n}$. The choice of the integer n has a big impact on the algorithm responsiveness. A small value for n causes the algorithm to behave as the naive algorithm. On the other hand, a big value for n causes the algorithm to respond slowly to the recent changes of the measured values, which is not useful in case of an unexpected increase in the workload. Choosing the best window size of the recent measured data, is one of the challenges that we had approached during designing our algorithm in Section 7.2.2.

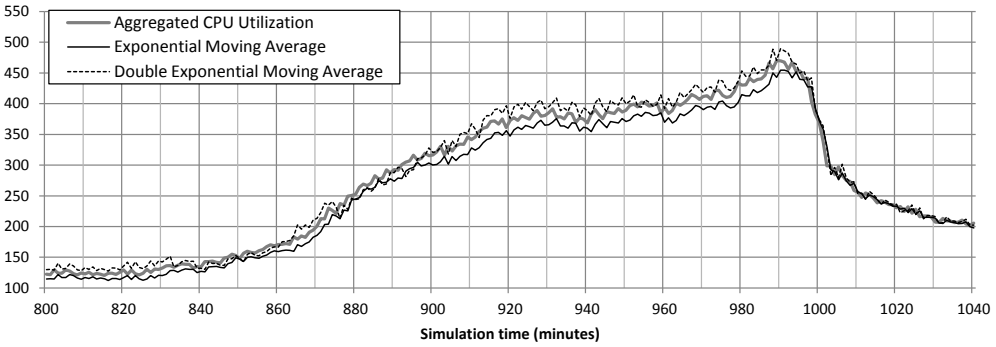
Exponential Moving Average algorithm: It gives a weight α ($0 < \alpha < 1$) to the last measured value Y_t, and lower weights to the former values. The weight for the former values decays with the time, which makes the algorithm responsive to the recent changes in the measured values. Nevertheless, both exponential moving average and the simple moving average algorithms assume stationary and not trending time series, therefore, both algorithms lagging behind the trends causing a slow response to the unexpected increase in the workload.

Double Exponential Moving Average algorithm: It avoids the limitation of the exponential moving average algorithm when there is a trend in the data. In addition to α, the algorithm considers β ($0 < \beta < 1$) as a smoothing factor for the trend in the data. For more details about exponential and double exponential moving average, we refer the reader to [26].

For comparison purposes, the studied algorithms are implemented into a module called *ProactiveAutoScaleManager*, which substitutes the reactive controller *AutoScaleManager* seen in Figure 6.4. Actually, there is a huge literature on the topic of time-series forecasting including the artificial intelligence concepts



(a) Comparison of Naive and Simple Moving Average Algorithms



(b) Comparison of Exponential and Double Exponential Moving Average Algorithms

Figure 7.7: Web tier - first day (the minutes 800 to 1040), reactive v.s. proactive scalability

such as neural networks, genetic algorithms, and fuzzy logic. However, in the production environments, the light-weight algorithms that do not imply a high computation or an intensive training are more suitable.

7.2.2 The Proposed Prediction Algorithm

The lag in the prediction algorithms: Naive, simple moving average and exponential moving average does not help mitigating the SLO violation. On the other hand, finding optimal values for α and β, for the double exponential algorithm is dependent on the workload.

Our proposed algorithm is designed to cope fast with the workload increase. It finds the next value u_{t+1} by calculation the simple regression of the last measured values n. However, a very large window results in slow reaction to workload variations. On the other hand, a very small window results in unreliable predic-

tions. Our solution is to calculate the window size depending on the statistical analysis of the historical measurements. For example, we start with a small window of the historical measurements (e.g., five values: $[u_t, u_{(t-1)}, \ldots, u_{(t-4)}]$), then we iterate through the previous values to include measured values that have the same statistical characteristics. The first iteration compares $[u_t, u_{(t-1)}, \ldots, u_{(t-4)}]$ with $[u_t, u_{(t-1)}, \ldots, u_{(t-5)}]$ to examine if the two groups have the same statistical attributes. If the compared groups showed high similarity, the test continues to compare $[u_t, u_{(t-1)}, \ldots, u_{(t-4)}]$ with $[u_t, u_{(t-1)}, \ldots, u_{(t-6)}]$.

We use t-test to compare the results. It is a test used to compare two groups of samples, even if they have different numbers of elements. Our null hypothesis is that there is no difference in measured values between the two groups. For each compared groups, we calculate $pValue$. We stop appending new elements to the second group when the $pValue$ is less than 0.05 (i.e., hypothesis is rejected). The algorithm is represented in Algorithm 1.

Algorithm 1 The Proposed Prediction Algorithm

Algorithm: Prediction of the next utilization value
Input: A list of the last n measured utilizations, $[u_t, u_{(t-1)}, \ldots, u_{(t-n-1)}]$
Output: The next utilization value u_{t+1}
Initialization: $pValue = 1$, $n = 5$, $j = 1$, $limit = 100$
while $pValue > 0.05$ && $j < limit$ **do**
  $pValue = tTest([u_t, u_{(t-1)}, \ldots, u_{(t-n-1)}], [u_t, u_{(t-1)}, \ldots, u_{(t-n-1-j)}])$
  $j++$
end while
// Calculate the regression of the new window $[u_t, u_{(t-1)}, \ldots, u_{(t-n-1-j)}]$
$slope = regression.getSlope()$
if $slope > 0$ **then**
  $u_{t+1} = regression.predict(t+1)$
else
  $u_{t+1} = u_t$ //Naive algorithm
end if

During the evaluation of our algorithm, we noticed some periods of unchangeable workload, e.g., the period from 0 to 360, as seen in Figure 7.4. For such periods, considering more historical measurements will not improve the prediction accuracy. Therefore, we determined a limit for the window size to reduce the computation (i.e., limit = 100). After determining the best historical win-

dow size, we calculate the simple regression and find both the slop and the next predicted value.

In spite of the fact that the fast scale out is crucial for coping with the sudden increase in the workload, the fast scale down is harmful for the system performance. Therefore, at scale down (i.e., negative slopes) time, we use naive algorithm (i.e., $u_{t+1} = u_t$) to delay the scale down and increase its consistency.

7.3 Evaluation

The first part of this section evaluates the *scalability thresholds'* tuning impact on the system performance. The second part evaluates substituting the reactive scalability with proactive scalability on both the system performance and the total cost.

For evaluating the *scalability thresholds'* tuning impact on the system performance, we repeat the experiment in Section 6.2.2, but with tuned scalability parameters. The first row in Table 7.3 is an evaluation of one week simulation with scale out thresholds 70% and 62% for the web tier and application tier, respectively. For scale down, we set 30% as a threshold for both tiers. The second row of the Table 7.3 is an evaluation for the same simulation period but with tuned thresholds 66% and 58% for the web tier and application tier, respectively. The scale down threshold is 30% for both tiers.

As we explained in Section 7.1.1, the reduction of the SLO violation is a trade of the total cost. However, the results in the second row show a high reduction in the SLO violation (i.e., 72.29%) with a small increase in the total running cost (i.e., 3.81%). The reduction in SLO violation is appreciated when we recall that 1% SLO violation means that for each 100 running hours there is an accumulated one hour where the response time of the system is higher than 100 milliseconds.

In the second part of the experiment, we substituted the reactive scalability by proactive scalability. The naive (N) and the moving average (MA) algorithms show no change to the result archived by reactive scalability with tuning (RT). The exponential moving average (E), with $\alpha = 0.9$, results in higher SLO violation compared to RT, due to longer periods of the under-provisioning, as seen in Figure 7.7(b). On the other hand, both double exponential moving average (DE) and

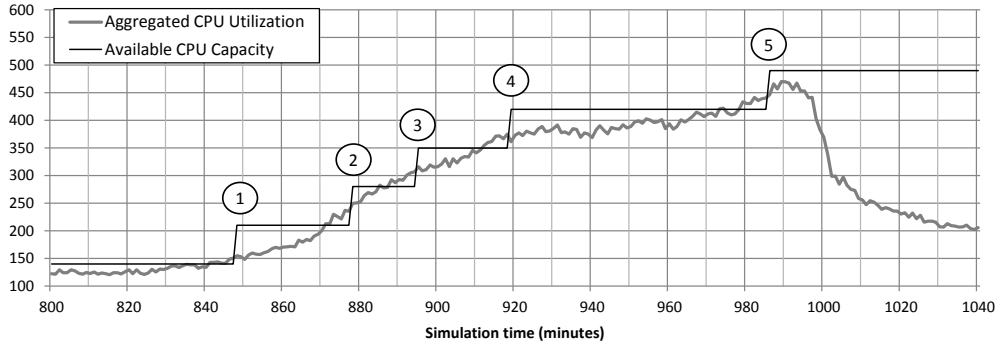
Algorithm	Cost ($)	SLO violation (%)	Cost increase (%)	SLO violation decrease (%)
R	49.84	4.72	N/A	N/A
RT	53.36	1.30	7.06	72.46
N	53.36	1.30	7.06	72.46
MA	53.44	1.30	7.22	72.46
E	53.43	2.82	7.20	40.25
DE	55.32	0.56	11.00	88.14
O	**54.24**	**0.55**	**8.83**	**88.35**

Table 7.2: Results of the simulation described in Section 6.2.2 for one week. The algorithms are: R-Reactive (without tuning). RT-Reactive with tuning. N-Naive algorithm. MA-Moving average (n=5). E-Exponential moving average ($\alpha = 0.9$). DE-Double exponential moving average ($\alpha = 0.9$ and $\beta = 0.3$). O-Our algorithm.

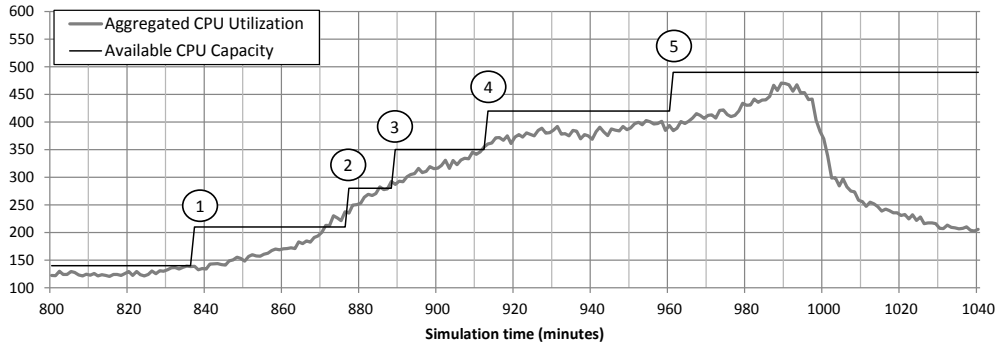
our proposed algorithm (O) outperform the other algorithms. Nevertheless, our algorithm shows a lower cost compared with DE algorithm.

To understand the improvement achieved by tuning the scaling thresholds, we plot the period of time between the minutes 800 and 1040 of the system run time. This period is selected while it shows a fast increase in the workload. Figure 7.8(a) shows the system scale out in five steps. The system scale only when the CPU utilization exceeds the *scalability thresholds*. Therefore, with each scale out, we notice a period of under-provisioning (i.e., SLO violation), where the aggregated CPU utilization is higher than the available CPU capacity. On the other hand, Figure 7.8(b) shows that tuning the scalability thresholds causes an early scale out for the system. For example, the first scale out step in Figure 7.8(b) toke place at the minute 837 instead of the minute 848 in Figure 7.8(a). Similarly, the third, forth, and fifth scales out also occurred earlier in the case of tuning the scalability thresholds. Nevertheless, tuning the thresholds did not help with the second scale out step where the workload increases very fast.

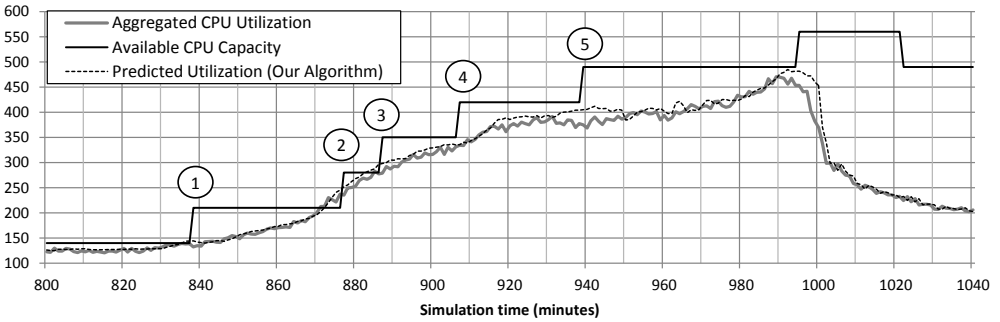
To explain the achieved improvement by our algorithm, we draw the system scaling in Figure 7.8(c), for the same period of time (i.e., 800 to 1040). The figure shows an early occurrence of the third, fourth, and fifth scale out of the system. On one hand the early scaling-out reduces the SLO violation, on the other hand it leads to a longer run of the VMs and a higher total cost. Nevertheless, with our



(a) Reactive scalability - without tuning



(b) Reactive scalability - with tuning



(c) Proactive scalability - with our algorithm

Figure 7.8: Web tier - first day (the minutes 800 to 1040), reactive (without tuning) v.s. reactive (with tuning) v.s. proactive scalability

approache, the improvement in the performance is much bigger than the increase in the cost.

7.4 Summary

In this chapter, we studied reducing the impact of the resource provisioning overhead on the applications' performance in the IaaS layer. We developed two approaches. Both of them exploit the trade-off between the performance and the cost. However, the improvement in performance was significant compared to the increase in the cost.

The first approach extracts the optimal *scalability thresholds* for an application considering the workload statistics, the application's *performance thresholds*, and the initialization time of resources. The results showed that tuning the *scalability thresholds* can eliminate 72% of the SLO violation with only 7% increase in the cost. The novelty of this approach lies in the fact that it does not imply any modification to IaaS provider infrastructure.

The second approach employs the proactive scalability to provision resources in advance and avoid the overloading periods. The results showed that the proactive scalability reduces the SLO violation 88% with only 9% increase in the total cost.

Our approaches are evaluated using models extracted from RUBiS benchmark at a physical environment. The models are fed into our developed simulator (ScaleSim), which is described in Section 6.2. Nevertheless, our approaches can be applied to any other application. Our approaches help IaaS customers, as well as SaaS and PaaS providers, to optimize the cost and performance of their applications, and consequently maximize the profit.

Chapter 8

Performance Interference in Public IaaS Environments

The advance of cloud computing services has attracted many customers to host their applications in the cloud. The Infrastructure as a Service (IaaS) model provides the customers a higher control over the provisioned resources compared to other models (i.e., PaaS and SaaS). Moreover, workload consolidation in cloud environments is an opportunity for service providers to increase resource utilization in their data centers. Nevertheless, workload consolidation exposes the VMs' performance to interference due to a possible contention on resources by the co-located VMs in the same physical host, as shown in Figure 8.1.

Typically, the IaaS providers offer high-level monitoring for several metrics (e.g., CPU, Memory and Network). The high-level monitoring is necessary for customers to manage their applications' scalability and performance. Nevertheless, it does not help detecting performance interference due to the following facts: First, contention on resources is typically hidden and cannot be measured by the high-level metrics. Second, according to the multi-tier architecture, the influence of a contention in one tier migrates to other tiers [119], that makes it difficult to determine and manipulate the main source of the contention. Finally, a contention on some resources results in misleading monitoring values for other metrics of the VM instance. For example, co-locating two VM instances competing on the memory bus, on the same physical host, limits their consumption of

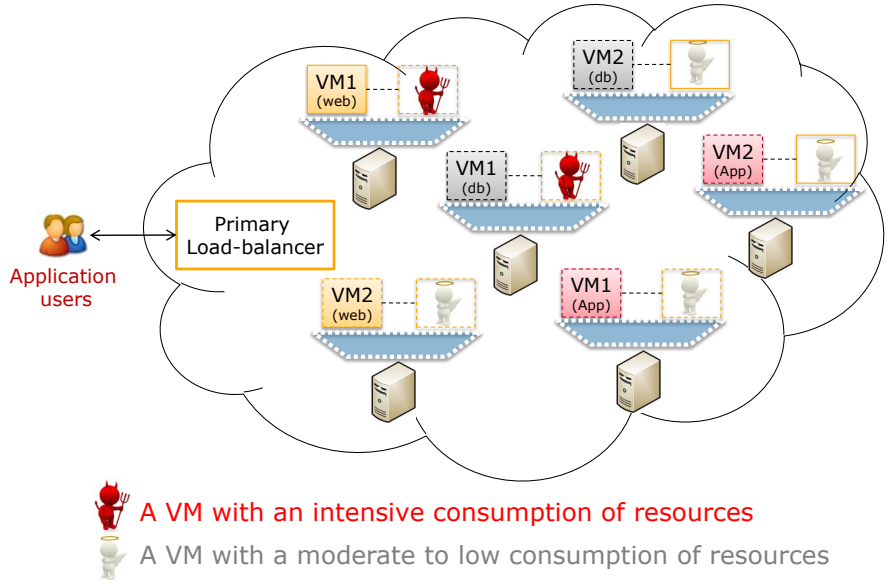


Figure 8.1: Public cloud infrastructure exposes a VM's performance to the influence by other VMs

the CPU, that is considered as a reduction in the CPU utilization, while in fact the system approaches performance degradation, as we explain by our experiment in Section 8.3.3.

On the light of above facts, we investigate maintaining the application performance in public IaaS environment by abandoning the contended VM instances automatically. The first step towards this goal is to determine the contended VM instances. To do that, we built models describing a VM's performance at normal case. At run time, we measure VMs' deviation from extracted models. The level and the period of the deviation are used to recognize the permanent contentions. Whenever a VM is marked as a contended VM, another VM in the same tier is initialized to take the place of the contended VM. Practically, customer has no control over the VM location in the IaaS provider's data center. Nevertheless, initializing a new VM instance exploits the possibility of locating it into another free-of-contention host. Our approach is a customer-oriented solution that does not imply provider involvement, and therefore it is suitable for production environments.

In the next section, we present the motivation and the real need for this study.

In Section 8.2, we explain our proposed system. Next, in Section 8.3, we evaluate our system using RUBiS benchmark. In Section 8.4 we describe the limitations and challenges in our proposed system. Finally, in Section 8.5, we conclude our work and present the intended future work.

8.1 Motivation

A VM running in IaaS is treated as a black-box. Neither the provider nor the co-located customers are aware or have control over the running software within other customers VMs. In fact, providers can use low level metrics to predict the software pattern (e.g., memory intensive or CPU intensive). Nevertheless, the enormous number of hosted VMs and the huge possible combinations of the hosted software within these VMs make it hard or even impossible for a provider to maintain individual customers' applications performance in IaaS model. As a result, the typical SLA of an IaaS provider describes only the annual up time of the instances, but it does not discuss potential performance degradation caused by the performance interference.

Virtualization technology is the main source of the interference in public clouds. A straight forward solution for eliminating the interference is to strengthen the isolation in virtualized environment. Memory and CPU isolation is relatively simple [89] [91] [112] compared to other shared resources [33] (e.g., memory bus and I/O devices), which exposes system performance to interference. For example, all virtualization technologies are able to dedicate specific pages for a VM. However, when two VMs in the same physical host are competing on memory bandwidth or hard drive access, there will be an interference [66] degrading both VMs performance. An experiment by Armbrust et al. [19] measured the average disk write rate of 75 instances in Amazon EC2 as 52 Mbyte/s with 16% standard deviation, which means that the I/O performance is very exposed to influence by the other customers in the cloud. Venkatanathan et al. [108] showed empirically that interference can degrade the performance of a cache-sensitive benchmark by more than 80%. Moreover, they introduced the concept "resource freeing attack" (RFAs). The attack depends on understanding and modifying the workload on a victim's VM to free up resources for the attacker's VM. Their attack was also

possible on public environments, such as Amazon EC2. Malkowski et al. [74] showed empirically that increasing the workload of an n-tier application may unexpectedly spike the overall response time of another co-located system by 300% despite stable throughput. A statistical study by Benson et al. [22] showed that several performance and virtualization issues were intensively discussed in a message forum of one of large IaaS providers. The study classified the threads for three years period started from 2007.

8.2 The Proposed System

Our proposed system integrates the proactive scalability architecture, designed in Section 7.2, with the models extracted in Section 6.1.4 to have a system coping rapidly with the workload variation and automatically replacing the VMs that can degrade the performance. The proposed system architecture is presented in Section 8.2.1, while the operational details are explained in Section 8.2.2.

8.2.1 System Architecture

Our system assumes that the application is hosted in a public IaaS environment and deployed as a multi-tier architecture. A typical multi-tier architecture is illustrated at the upper part of Figure 8.2. The rounded rectangles show running instances in each tier. Whenever the customer submits a request for an instance, the provider finds the best host according to the instance type and workload on physical hosts. The same host can run instances of different customers with different demands. The number of instances at each tier should be determined by the IaaS's customer according to the workload variation.

The services of the IaaS provider can be consumed through APIs. The customers can develop clients with different programming languages to interact with the IaaS provider. Figure 8.2(b) illustrates our proposed system. To reduce the overhead, the proposed control system can be hosted in a VM running in the IaaS environment. The main modules of our system are the *Sampler* module, the *Predictor* module, and the *Provisioner* module. The *Sampler* module periodically makes queries to the load balancer VM ("nginx" in our case) to get the last web

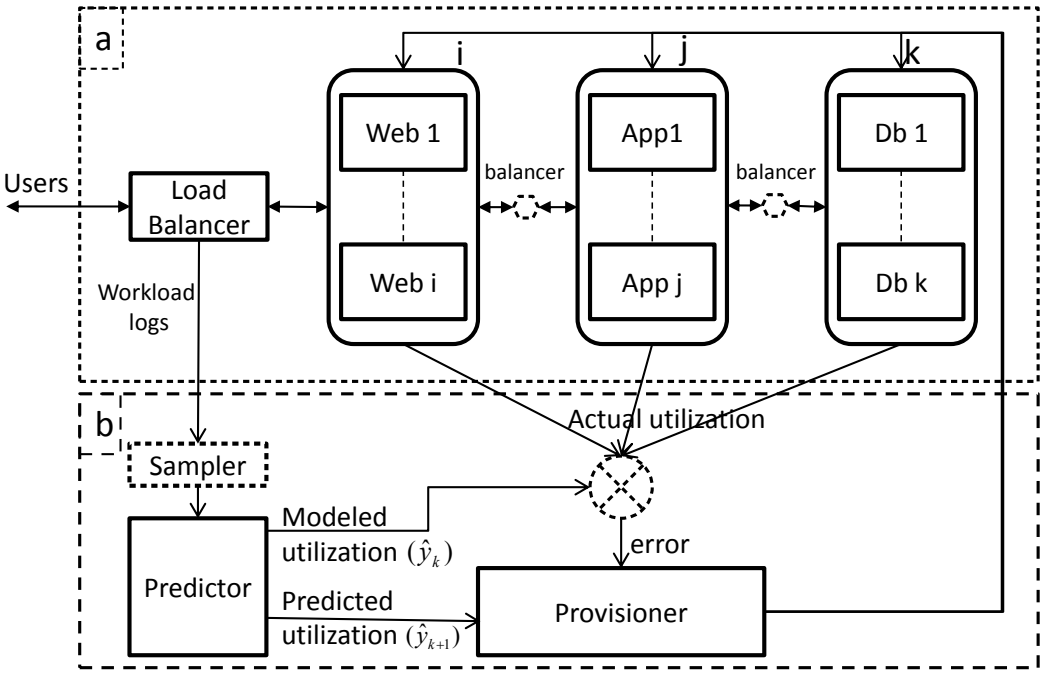


Figure 8.2: Our dynamic scalability and contention detection architecture

traces. To make the traces accessible remotely, we modified the load-balancer to log into a MySQL database. The *Predictor* module predicts the utilization of each resource according to the traces from the load-balancer. Moreover, it has a prediction component that help the system to scale proactively using our algorithm designed in Section 7.2. The only module that consumes IaaS services is the *Provisioner* module. The interfaces allow the *Provisioner* module to get on-line measurement of resource utilization, and decide accordingly about the number of instances in each tier and the VM instances that should be replaced. The details and the interaction between these modules are explained in Section 8.2.2.4.

8.2.2 Operational Phases

The architecture of our proposed solution is shown in Figure 8.2(b) and explained in details in Figure 8.3. It integrates the monitoring of resource utilization with requests logging. Our proposed system incorporates the following phases: monitoring, sampling, models building, and running.

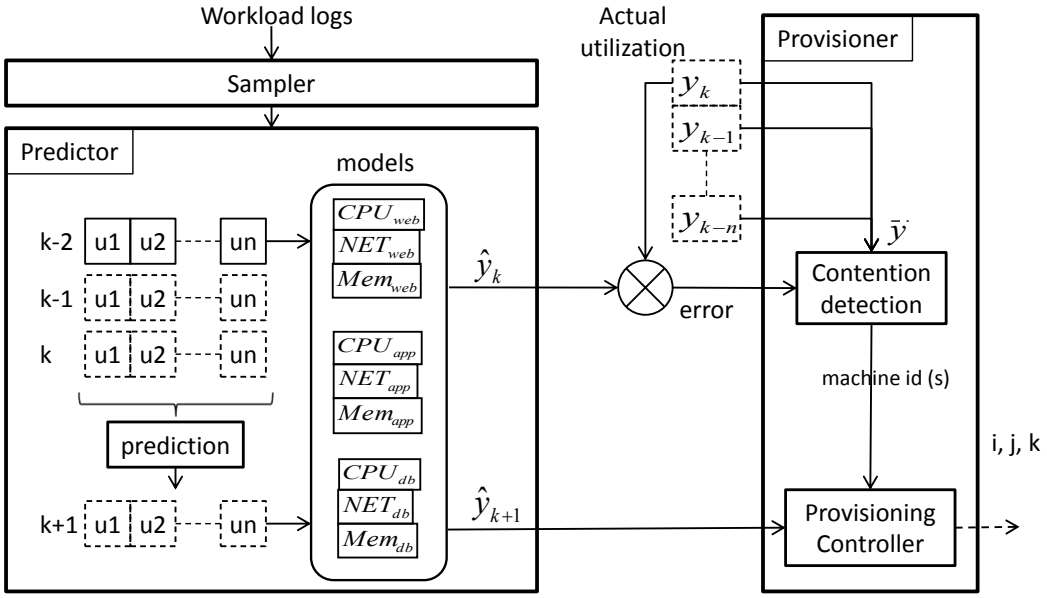


Figure 8.3: Details of modules composing our proposed architecture for dynamic scalability and contention detection

8.2.2.1 Monitoring Phase

Beside the monitored resources that are provided by current IaaS providers, we consider the log files that describe the transactions handled by the application. Typically, these logs locate at a single point where all traffic passes (e.g., load balancer). However, to avoid single point of failure, some providers offer hardware load balancers or Elastic Load balancers [12] as we will discuss in Section 8.3.4. Even in such cases, the logs collected by different web servers can be synchronized to one log file that describes the total workload. The access log file can be configured to log access time, response time, and URLs of requests.

8.2.2.2 Sampling Phase

As the transactions continuously arrive to the system, we consider a sampling window that describes the transactions' behavior at specific period of time. Within the sampling window, we classify the transactions into categories, and then determine the rate of each category. The categories are determined depending on the URL. In fact, types of requests depending on URL can be very large in a

real application. However, classifying these requests to coarse-grained categories: cacheable, non-cacheable, and demanding [18] reduces the number of categories that really have an impact on the system utilization. For example, Zhang et al. [119] showed that using only 20 types of a requests with a system of total 756 types of requests leads to accurate prediction of resource utilization. In our experiments, we considered 18 types of requests using RUBiS benchmark and ignored the cacheable requests (i.e., static pages and images), since they show a negligible impact on web tier's performance. The output of the sampling phase for the sampling period k is a vector describes the number of requests of each category ($k[u^1, u^2, \ldots, u^n]$). These sampled vectors are used for both training and running the system.

8.2.2.3 Modeling Phase

In the modeling phase, to avoid any influence from the other tiers, we run several replicas in all tiers except the modeled one. For instance, to model a web tier, we run a single VM instance at the web tier but many replicas at both application and the database tiers. The collected traces are CPU utilization, Memory allocation, Network IN/OUT rate, and Disk read/write rate. Afterwards, we synchronize the collected traces with the access logs that are collected in the sampling phase. A vector of requests' rate is the input of our models, while the output is one performance metric at a time (e.g., CPU utilization). More details about the ARX models are presented in Section 6.1.4.

8.2.2.4 Running Phase

At running phase, the modules *Sampler*, *Predictor*, and *Provisioner* are in continues interaction. The *Sampler* module, seen in Figure 8.3, continuously extracts a raw vector of requests' rate for each sampling period k. These rates are passed to the *Predictor* module as a vector $k[u^1, u^2, \ldots, u^n]$. The *Predictor* module has two outputs: First, the calculated utilization $\hat{y_k}$ depending on the models extracted at modeling phase. Second, the prediction of the resources utilization $\hat{y_k}$ for the next period of time (i.e., one step ahead). The two outputs of *Predictor* as well as the actual utilization are passed as inputs to the *Provisioner* module. The

error $|y_k - \hat{y}_k|$ (e.g., the absolute difference between the actual measure CPU utilization and the CPU utilization calculated by the CPU model) is calculated for each monitored resource of each VM instance. The on-line measured utilization y_k, the average of the measured utilization $\overline{y}$, as well as the modeled utilization $\hat{y_k}$ are used in Equation 8.1 to calculate F, which describes the fitness of the measured to the modeled values of resources utilization of VMs instances at each tier.

$$F = 1 - \sqrt{\frac{\sum_{k=1}^{N} |y_k - \hat{y}_k|^2}{\sum_{k=1}^{N} |y_k - \overline{y}_k|^2}} \tag{8.1}$$

Actually, Equation 8.1 is applied for each resource r of the VM instance n, that runs at tier m. To formalize this, we consider M tiers. Each tier runs N VM instances, while each VM instance has R resources. Practically, we implemented $\hat{y}_k$, $\hat{y}_{k+1}$, $\overline{y}_k$, and y_k using ArrayList object, while each tier can have a different number of instances. Nevertheless, we represent ArrayLists as three dimensional arrays for simplicity.

Whenever the error $|y_k - \hat{y}_k|$ is very high, the fitness value F will be very low, that is an induction of a degradation in VM performance due to a contention for a resource.

The *Provisioner* module periodically calculates F value (e.g., every 20 seconds in our setup). Each time a resource of a VM shows a very low fitness to the modeled values, the VM is marked as a contended VM. However, to maintain the system stability, no action is triggered until the number of low-fitness states registered by a VM in $temp\_VMs[\,]$ is higher than $ConfWindow$ value, which is 48 in our setup. Whenever a VM instance records deviations from the model higher than the determined $Confwindow$, the algorithm add the id of that VM instance to $VMs[\,]$ vector for replacement and remove its records from $temp\_VMs[\,]$ vector.

In parallel to the contention prediction algorithm, the provisioning algorithm (i.e., Algorithm 3) continuously checks the contented VMs list $VMs[\,]$. The first step to eliminate a contented VM is to run a new VM instance in the same tier. Afterwards, the algorithm continues its run looking for any overloaded tier m. The minimum and maximum thresholds of each resource r at tier m are described

Algorithm 2 Contention Prediction Algorithm

Input: $y_k[M][N][R]$, $\hat{y}_k[M][N][R]$, $\overline{y}_k[M][N][R]$, and $ConfWindow$
Output: $VMs[\,]$
Initialization: $temp\_VMs[\,] \leftarrow null$ (Local vector contains candidate VMs for replacement)
loop
  // Find VMs with potential contention
  for $m = 1$ to M **do**
    for $n = 1$ to N **do**
      for $r = 1$ to R **do**
        //Calculate F at Equation 8.1 for each resource r
        if $F < 0$ **then**
          insert VM_id into $temp\_VMs[\,]$
        end if
      end for
    end for
  end for
  if count(VM_id in $temp\_VMs[\,]$) $> ConfWindow$ **then**
    insert VM_id into $VMs[\,]$
    remove VM_id from $temp\_VMs[\,]$
    pass $VMs[\,]$ to Provisioning controller
  end if
end loop

by system administrator as $min[m][r]$ and $max[m][r]$, respectively. Determining these thresholds is studied intensively in Section 7.1.1.

A tier is scaled out when the average predicted utilization $\hat{y}_{k-1}[m][n][r]$ is higher than the predetermined maximum thresholds $max[m][r]$. To be sure that each tier is scaled up once per a control interval, we keep a tag $scale\_up[m]$ for each tier m. Similarly we have $scale\_down[m]$ tag for scaling down. These tags are reset for next control period (i.e., after 5 minutes in our setup). At scaling down, the algorithm gives the priority for terminating contended instances.

In our algorithms we assume that each load-balancer, seen in Figure 8.2, routes the same amount of traffic for each replica. In other words, to calculate $\hat{y_k}$ and $\hat{y_{k-1}}$, the model should divide the input vector $[u^1, u^2, \ldots, u^n]$ by number of replicas. We will validate this assumption in Section 8.3.2.

Algorithm 3 Dynamic Provisioning Algorithm

Input: $\hat{y}_{k-1}[M][N][R]$, $max[M][R]$, $min[M][R]$, $Current[M]$, and $VMs[\,]$
Output: $Next[M]$
Initialization: $scale\_up[M] = scale\_down[M] =$ false
loop
  for $m = 1$ to M **do**
    if $VMs[\,] \neq null$ **then**
      // Add new VM instances sibling to contented VMs
    end if
    for $n = 1$ to N **do**
      for $r = 1$ to R **do**
        // Check if scaling up is required
        if $\hat{y}_{k+1}[m][n][r] >= max[m][r]$ **and** $scale\_up[m] \neq true$ **then**
          // Add new VM instance to tier m
          $Next[m] \leftarrow Current[m] + 1$
          $scale\_up[m] \leftarrow true$
        end if
        // Check if scaling down is possible
        if $\hat{y}_{k+1}[m][n][r] < min[m][r]$ **and** $scale\_up[m] \neq true$ **and** $scale\_down[m] \neq true$ **then**
          // Turn off a VM instance at tier m
          if $VMs[\,] \neq null$ **then**
            // Schedule terminating the contented VM
          else
            // pickup any VM for scheduled termination
          end if
          $Next[m] \leftarrow Current[m] - 1$
          $scale\_down[m] \leftarrow true$
        end if
      end for
    end for
  end for
end loop

8.3 Evaluation

Due to infeasibility of controlling the VMs location in the cloud, we hosted the system in our infrastructure. Co-locating VMs in the same physical host is necessary to imitate a contention for resources. However, no big modifications are

required to run our system in real IaaS while we implemented clients for both the vShpere and Amazon EC2 environments.

8.3.1 Physical Setup

Our physical infrastructure consists of three Dell OptiPlex 980 servers with Intel Core i5-2400 CPU @ 3.10GHz. Memory size was 8 GB on all machines. These machines are connected using a Gigabit-Ethernet switch. The installed hypervisor is VMware ESX 4.1. To monitor VMs utilization, we implemented a java-based client that consumes the web services of each VMware server to get online measurements of VMs resources utilization. We implemented each tier into a different physical host. Moreover, we depend on the vCPU affinity to isolate performance. The running application is RUBiS benchmark [31] implemented as multi-tier system runs Apache as a web server, Tomcat as an application server, and MySQL as a database.

8.3.2 Models Extraction and Validation

To cover a variety of workload intensity, we ran RUBiS client with different values of *number of clients* that range from 100 to 1200. Moreover, we ran RUBiS with both browsing and updating requests. The online measurements are merged and synchronized with the logs from load balancer to build a model for each monitored resource. These models are used for contention prediction and system scalability.

To validate the extracted models, we run the experiment again but with different steps ranging from 150 to 1250 to have a different dataset for validation. In this experiment we intended to validate extracted models and at the same time measure the scalability impact on models fitness. Therefore, we ran two replicas in both web and application tiers. Figure 8.4 shows the cumulative distribution function (CDF) of absolute error of CPU utilization of web1, app1, and database. We do not show CDF of web2 and app2 since they are almost identical to CDF of web1 and app1, respectively. The figure shows that 90% of the measured absolute errors are less than or equal 2 for web and application replicas. On the other hand, 90% of the measured absolute errors are less than or equal 4.5 for

database. In addition to CDF, we calculate the fitness F, using Equation 8.1 for each VM instance.

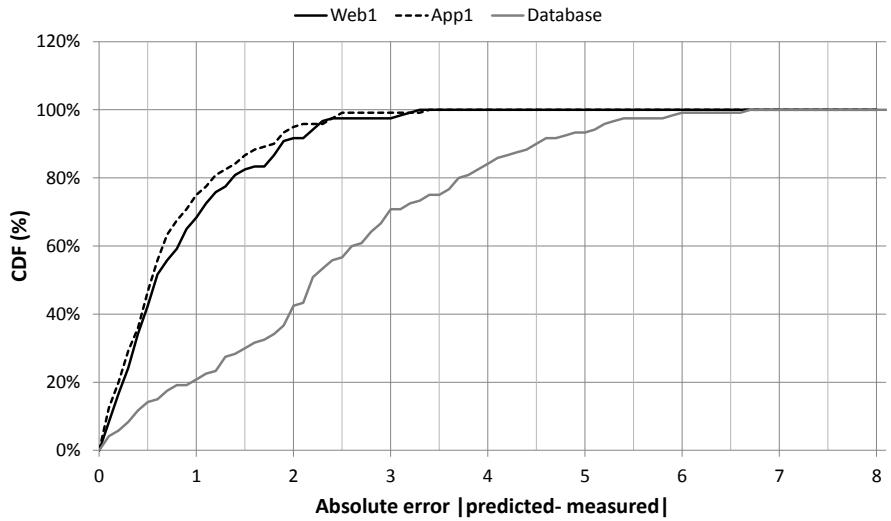


Figure 8.4: CDF of absolute error of CPU utilization of web1, app1, and database instances

Instance:	web1	web2	app1	app2	db
Fitness (%):	95.60	95.75	94.67	94.09	86.85

Table 8.1: Fitness of CPU utilization model for two web replicas, two application replicas, and a database calculated by Equation 8.1

The fitness values shown at table 8.1 validate the assumption that each replica behind a load-balancer running round robin balancing algorithm has similar amount of the workload. Practically, session-based load balancing can result in non-equally distributed traffic to the replicas. However, in all our experiments, even though sticky sessions are enabled at load balancers, the fitness of the models is still high. This confirms the statement of Zhang et al. [119]: "a multi-tier system with a complex session-based workload can be modeled with a transaction-based mix".

Another interesting observation was that among several experiments on different system structures, we noticed that the fitness of the CPU model for web and application instances is always higher than it for the database. For instance, the fitness of web and application tiers ranges from 90% to 95%, while it ranges

from 85% to 90% for the database tier. Zhang et al. [119] and Ahmad et al. [7] have the same observation. Therefore, [7] applied a heuristic approach to increase the model's accuracy. The heuristic approach ignores the difference between the predicted and measured values if it does not have an impact on the system performance. According to [7], their approach decreases the mean error of CPU model from 12.83 to 11.10. By experiments, we observed that stopping the binary logging in MySQL database can raise the fitness of database tier from 88% to 93%. Nevertheless, binary logging is necessary for database replication, which makes turning it off impractical.

To increase the database's model accuracy, we investigate more measured metrics that are related to database performance. The query cache hit-ratio is one of the metrics that influences the CPU utilization of the database. The query cache stores the text of a SELECT statement together with the corresponding result of the query. Any following identical SELECT statements will get the same results, which reduces the need for parsing and executing the same query for each SELECT statement. Moreover, the query cache can be shared among sessions, so many users can benefit from caching a frequently used SELECT statements[1]. Nevertheless, whenever a table is updated, all the cached queries from that table are flushed for consistency.

To measure the impact of the query cache hit-ratio on the CPU model, we evaluated the models again, but with disabling the query caching in our MySQL database. The fitness of the models showed no change, which means that the query caching has no significant impact on the CPU models. To investigate our observation, we calculated the cache hit-ratio as the following: Qcache_hits/(Com_select+Qcache_hits). It is measured for RUBiS benchmark as 10% to 12% for a mix of read/write workload, which is relatively low. This low hit ration explains why the caching of the queries does not show a significant impact on the CPU model. In other words, considering it as an input for our model will not improve the model accuracy.

[1] http://dev.mysql.com/doc/refman/5.1/en/query-cache.html

8.3.3 Contention Detection

In this section, we demonstrate the system ability to predict and eliminate contention of resources. First we consider running RUBiS with 1000 simultaneous clients. The experiment run time is divided into four epochs. Each epoch describes a different performance state: epoch1 (0 to 140), epoch2 (140 to 1100), epoch3 (1100 to 1500), and epoch4 (1500 to the end of experiment). Figure 8.5 shows that both *web1* and *web2* VMs were able to utilize around 48% of the physical core capacity until the end of epoch1. At that moment, a VM called *stress* started competing on the physical core with *web2*. *Stress* is a VM mapped to the same physical core with *web2*. It runs the command: "stress –cpu 10 –io 8 –vm 2 –vm-bytes 256M". The command implies intensive access to the hard drive and memory. As seen in Figure 8.5, *stress* VM has an impact on the entire system performance explained as the following: First, the response time jumped to 1400 ms second at the moment of starting *stress* VM and stabilized along the time interval (380 to 1100 seconds) to be 369 ms, which is 230% higher than the expected response time (i.e., 160 ms). Second, the contention caused by *stress* VM also decreased slightly the CPU utilization of the *balancer* instance, even though they are mapped to different physical CPUs. This decrease is due to the high overhead at the web tier, which consequently decreased the ability of the entire multi-tier system to accept more requests. Third, the contention had the most impact on *web2*, where the average CPU utilization dropped down to 35% instead of 48%. This is a practical example shows how a contention on resources can lead to misleading monitoring values.

Using Equation 8.1, the controller predicts the degradation in *web2* performance and starts resolving it at epoch3. As seen in Algorithm 3, the first step is to run another replica *web3* to take the place of *web2*. At that moment, the model of the web tier instances is updated to predict 3 replicas at web tier, as seen in Figure 8.6. Also, the fitness is calculated using Equation 8.1 to insure that *web3* has no contention with the other VMs on the physical host. Afterwards, the provisioning controller schedules the contended VM (i.e., *web2*) termination and keeps running *web1* and *web3*. This brings down the response time to 160 ms as it was at the beginning of experiment. Finally, we should notice that at

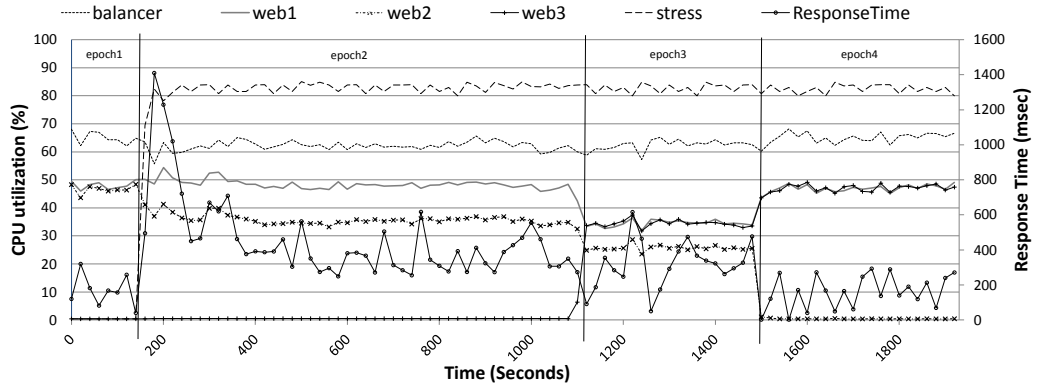


Figure 8.5: Detecting and eliminating contention that affected web2 VM instance

epoch3, even though the system has three web replicas, the VM with contention has influence on the response time, while a portion of the traffic is still routed to *web2*.

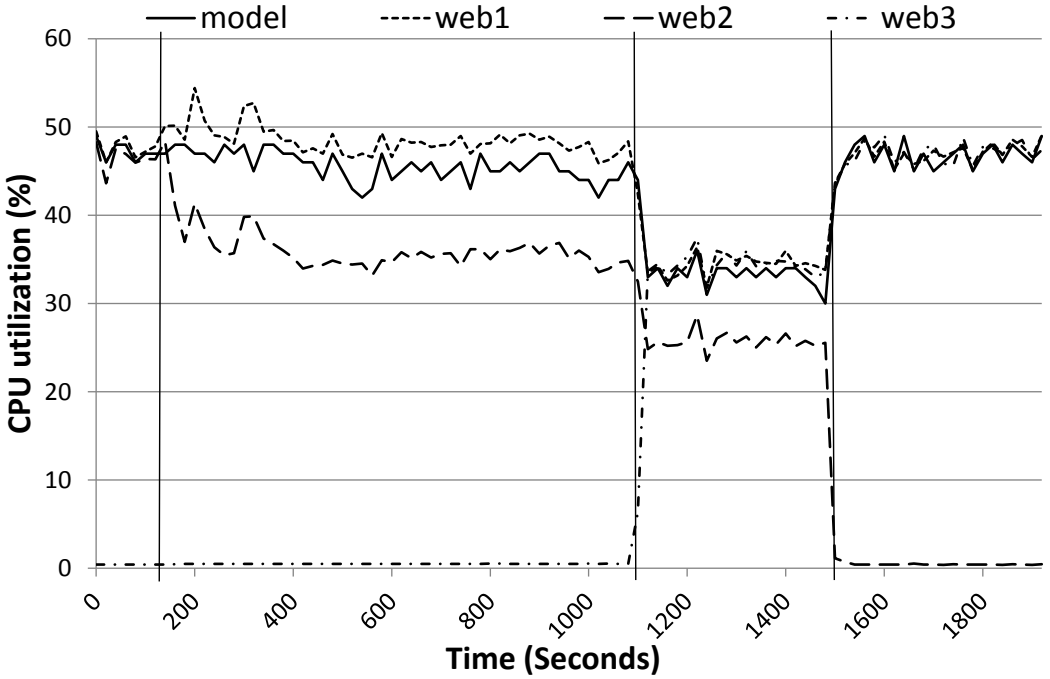


Figure 8.6: CPU utilization of web tier replicas

Figure 8.6 shows both the measured and the predicted CPU utilization of each web instance. Epoch 1 and 4 show that the measured CPU utilization closely

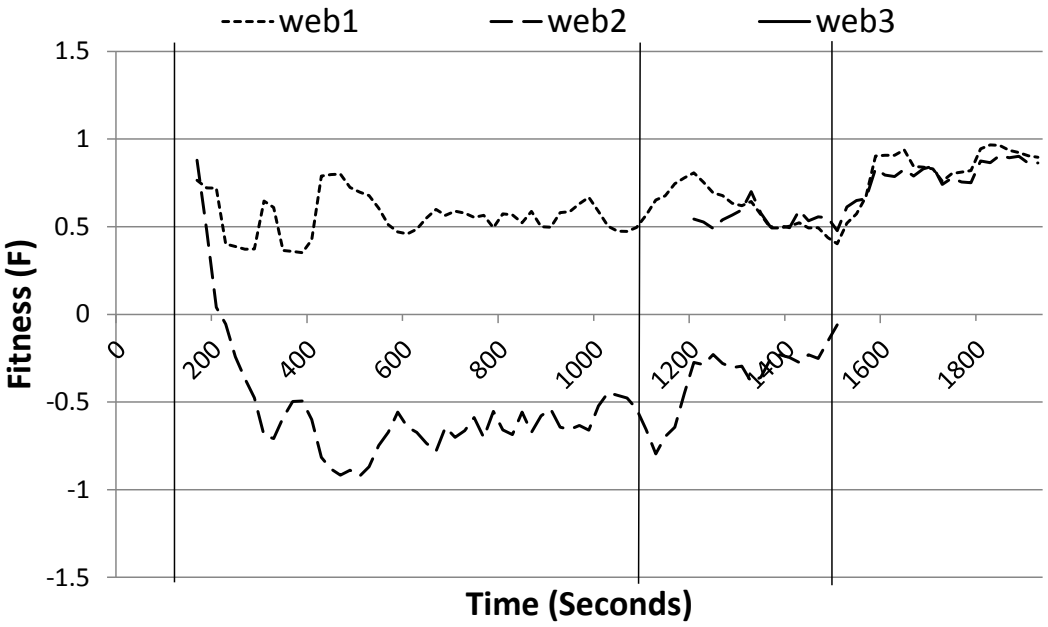


Figure 8.7: Fitness of web tier replicas calculated by Equation 8.1

fits the modeled utilization. However, at Epoch 2, according to contention at *web2*, the balancer routes more traffic to *web1*, which results in an increase in measured CPU utilization compared with the modeled one. Figure 8.7 shows the fitness of each web instance model. Epoch 4 shows that after replacing the contended instance (i.e., *web2*), the fitness of *web1* and *web3* models comes close to "one", where "one" is the highest fitness value. The value of $\overline{y}_k$ in Equation 8.1 is calculated from the last five measurements of a VM lifetime; therefore, there is discontinuity in fitness charts in Figure 8.7.

8.3.4 Technical Discussion

In this section, we discuss some of the technical details that were confronted during our system implementation.

First, *Nginx* showed much better performance compared with Apache as a load balancer. During our experiments, we noticed that Apache performance degrades drastically at a high rate of traffic. Moreover, either in process-driven or thread-driven modes, Apache consumes much memory to spawn more processes or threads. On the other hand, as an event-driven application, *Nginx* was able

to outperform Apache performance even with fewer processes.

Second, in an IaaS environment, running a new VM instance implies assigning a new IP address to the new instance, which is unknown to load balancer. This requires updating the load balancer with the new IP addresses online. In fact, both Apache and *Nginx* enable online reloading of the configuration file, which contains replications details. However, *Nginx* showed no degradation in performance compared to Apache, that interrupts the service temporarily by killing current processes and creating them again.

Third, we implemented our prototype into a local infrastructure to have more control over resources during experiments. However, Amazon EC2 [12] has all the tools that support implementing our approach. For example, using Amazon AWS client, a user can monitor, provision, and terminate instances remotely. For load balancing, a customer can either use static load balancers, as seen in our experiments, or an *Elastic Load Balancer* [101]. *Elastic Load Balancer* is a service available at Amazon EC2 to increase the applications reliability. It has many advantages while it allows distributing load among different zones. Moreover, the provider is responsible for its reliability and dynamic scalability to cope with workload demand. Currently, using Amazon CloudWatch, clients can get the number of requests that are manipulated by *Elastic Load Balancer* and the response time of each request. However, there is no information about the request's URL. We hope that Amazon considers such a metric in their development. Until then, our approach remains valid using static load balancer, or by collecting and synchronizing access logs from web tier instances.

8.4 Challenges

Adapting the models dynamically is one of the challenges that confront our system. For example, the database size growths gradually with the usage causing a deviation from the models. Moreover, modifying the application or adding new contents requires building new models considering the recent updates. One solution is to have a copy of the current system and repeat the steps in sections 8.3.1 and 8.3.2 with each modification or a significant increase in the database tables size. Another solution is to allow the system to adapt automatically (i.e.,

rebuild new models) from the traces collected from production environment. In other words, allow the system to discover the appearance, the absence, or the modification of services, then to categorize the URLs according to their influence on the system utilization.

Towards automatic categorization and more accurate models, Zhang et al. [120] suggested improving the models' accuracy by iteratively splitting and merging the categories depending on estimated resource usage. Their approach is inspired by [93] approach. The Sharma et al. [93] approach successfully discovered two workload categories of PetShop benchmark using a machine learning technique called Independent Component Analysis (ICA). The approach does not require any information about requests URL. Nevertheless, the ICA approach limits the possible number of categories to the number of measurable resources. Zhang et al. [120] overcome this restriction by proposing transaction ICA. Instead of using ICA solely, they start initial categorization of requests using URL, then improve it iteratively by merging homogeneous categories and splitting heterogeneous categories. The proposed approach "Transaction ICA" by Zhang et al. [120] is a promising solution. However, ICA assumes that the sources (i.e., inputs) must be non-Gaussian and independent. In our case, there is a correlation between requests arrival that violates the ICA assumption of the inputs independence. Ghanbari et al. [49] suggest classifying requests depending on the response time. They validated their approach on two tier system (i.e., web and database server). During our experiments, we noticed that the response time is not a reliable metric for categorization, since it is very sensitive to the performance influence in the public infrastructures.

The second challenge for our system is the noise in public environments. The noise caused by performance interference can influence the accuracy of the built models. To mitigate the impact of the noise on the models accuracy, we extracted the models depending on big datasets of monitoring values. Moreover, we filtered these datasets manually. For example, we remove any records showing resource saturation. Also, we remove records showing a transient states such as the start or the end of the benchmark run; or the moments of increasing or decreasing the intensity of the workload.

8.5 Summary

The IaaS providers offer their customers the control and the tools to manage their application performance. On the other hand, IaaS environment leaves the customers' VM instances exposed to the influence by the other VMs' workload. Virtualization technology struggles to improve the performance isolation especially for resources such as Memory bandwidth and I/O resources. In the same time, provider is unable to monitor every customer application performance due to the enormous number of the applications and the different requirements for each one.

Until improving the shared resources isolation in virtualized infrastructures, IaaS customers are in need for a customer-oriented solution to mitigate the degradation in performance caused by other customers' workloads. At this point, our solution stands as a practical and light-weight solution to escape the highly potential contention in public infrastructures. Even it does not handle the isolation problem directly, it exploit the probability that not all physical hosts in the cloud provider data centers hosting VMs with intensive workload.

In our approach, we categorized the requests depending on the URLs and employed prior coarse-grain classification (i.e., cacheable, non-cacheable URLs). Our observations show that prior coarse-grain classification and a careful selection of the queries that are considered in the models extraction have much impact on the model's accuracy. We used RUBiS benchmark to build and validate our approach. The extracted models of the CPU utilization showed fitness ranges from 90% to 95% for web and application tier, and from 85% to 90% for the database tier. However, we intend to evaluate iterative categorizing techniques suggested by [120] and [49] to exploit the possibility of automatic categorization of the models' inputs.

Chapter 9

Conclusions and Future Work

Techniques of Hosting Internet applications have been developed rapidly during the past few years. This development was driven by the need for scalable infrastructures coping with the requirements of Internet applications and the workload variation. Before the emergence of the virtualization technology, data centers were providing either shared or dedicated hosting platforms for Internet applications. Virtualization technology added many features to data centers including workload consolidation, live migration, and dynamic management of resources. Cloud computing is a business model exploiting virtualization technology to enable on-demand access to a shared pool of resources that can be easily provisioned and released without provider intervention.

Nevertheless, the performance of the hosted applications in cloud environments is exposed to two main issues. The first issue is the resource provisioning overhead, which exposes the application to periods of under-provisioning. The second issue is the performance interference in public cloud infrastructures. In this chapter, we summarize our contributions towards solving these issues, highlight the limitations, and discuss directions for future work.

9.1 Summary of Research Contributions

In this dissertation, we made the following contributions:

1. *Scaling resources vertically to maintain performance:* The vertical scalability provides a rapid scaling for Internet applications over the horizon-

tal scalability. We exploit this advantage to scale an Internet application rapidly and avoid performance degradation. The scalability is driven by CPU and Memory loop-back controllers running in parallel with application controller. The main difference with respect to the pre-existing solutions is the Application controller, which helped maintaining a better performance even during the presence of competition on resources.

2. *Efficient use-cases for vertical scalability:* Scaling relational database tier horizontally is complex compared to web and application tiers. A part of our research was to scale the database tier with policies similar to horizontal scaling polices on the level of virtual cores rather than VMs. The results showed an efficient coping with the workload variation.

   The second use-case was improving the business model for selling the spare capacity in the provider's data centers, namely spot instances. The proposed idea was to scale the VM instance resources vertically with the price which is calculated dynamically according to the free capacity in the provider's data center. The proposed solution showed benefits for both providers and customers.

3. *Taking the best of reactive scalability in public IaaS environments:* Nowadays, IaaS environments provide reactive scalability for a huge number of applications. Reactive scalability depends on static thresholds to provision or terminate VM instances. Employing static thresholds is not the optimal solution to handle the dynamic provisioning process. Nevertheless, it is commonly used in large production environment due to its simplicity. Considering this fact, we investigated tuning scalability parameters to achieve better performance. The results showed 72% reduction in SLO violation and only 7% increase in the cost by optimizing scalability thresholds.

4. *Proactive scalability to improve the performance:* To reduce the impact of resource provisioning overhead on performance, we developed a prediction algorithm that proactively calculates the inbound workload depending on a dynamic window-size of historical measurements. The results showed 88% reduction in SLO violation and only 9% increase in the cost.

5. *Avoiding performance interference in public IaaS environments:* Although the economic benefits of workload consolidation, it exposes the performance of VMs to interference. Virtualization technology efficiently isolates some resources (e.g., CPU allocation) but struggles to isolate others, such as memory bus and I/O devices. Accordingly, many studies showed VMs' vulnerability to performance interference in public cloud environments [19] [108] [74]. In the course of this dissertation, we developed a system to detect the VMs that suffer from performance interference and replace them with new VMs. The main advantage of this solution is being a customer-oriented solution that does not require providers' involvement.

6. *A platform for extended research:* Running large-scale experiments in real environments is costly. On the other hand, testing a system on small testbeds does not guarantee its feasibility at large-scale level. To overcome this, we developed the ScaleSim simulator. It simulates the scalability components of Amazon EC2. ScaleSim is implemented into modules to allow other researchers to implement, run, and compare their algorithms at a large-scale level. To achieve realistic results, we imparted the simulator with models extracted from a real cloud environment.

9.2 Future Research Directions

This dissertation opens several opportunities for short and long-term research. We summarize future directions as follows:

1. *Improving the efficiency of vertical scalability:* The vertical scalability is an opportunity for fast scaling and optimal consumption of resources. However, it is exposed to several limitations. We plan to extend our research towards an efficient usage of the vertical scalability. The statistical analysis of the workload history can be the first step towards an efficient consolidation of the workload. For example, consolidating complementary workloads on one physical host allows an overloaded machine to scale up rapidly and maintain the performance without impacting the performance of other machines. Moreover, building a pricing model considering the vertical scala-

bility helps the customers to maintain their applications' performance and allows the providers to increase their revenue.

2. *Adapting models dynamically:* Currently, our approach implies much administrative work, therefore we investigate automating models extraction and adaption. Moreover, we plan to consider heterogeneous types of replicas as a technique to optimize resources provisioning.

3. *Fully automating the performance management of Internet applications:* As a future work, we plan to fully automate the optimization process for scalability parameters. Moreover, we are looking forward to mathematically modeling the relation between the scalability thresholds and the application performance. In this dissertation, we considered the CPU utilization as a scaling metric. However, we plan to consider other metrics such as memory and network.

4. *Investigating more applications:* In our research, we focused on RUBiS as a widely used benchmark to model multi-tier systems. We plan to consider other types of Internet applications to generalize our findings.

9.3 Summary

This dissertation solved several open questions in the field of cloud resource management to provide an efficient scalability and performance management for Internet applications. Our study advocates customer-oriented solutions because of their feasibility in public cloud environments. The results of our study provide a motivation for interesting future research.

Bibliography

[1] GoGrid. http://www.gogrid.com/, (Retrived: March 2, 2013).

[2] Microsoft Performance and Resource Optimization (PRO). http://technet.microsoft.com/enus/library/cc917965.aspx, (Retrived: March 2, 2013).

[3] Rackspace. http://www.rackspace.com/, (Retrived: March 2, 2013).

[4] RightScale. http://www.rightscale.com/, (Retrived: March 2, 2013).

[5] VMware Distributed Resource Scheduler - Dynamic Resource Balancing. http://vmware.com/products/drs, (Retrived: March 2, 2013).

[6] Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J. Elmore. Database scalability, elasticity, and autonomy in the cloud. pages 2–15, apr 2011.

[7] Mumtaz Ahmad and Ivan T Bowman. Predicting system performance for multi-tenant database workloads. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, DBTest '11, pages 6:1—-6:6, New York, NY, USA, 2011. ACM.

[8] Sherif Akoush, Ripduman Sohan, Andrew Rice, Andrew W. Moore, and Andy Hopper. Free lunch: exploiting renewable energy for computing. In *13th Workkshop on Hot Topics in Operating Systems*, page 17. USENIX Association, may 2011.

[9] Mohammed A. AlZain, Eric Pardede, Ben Soh, and James A. Thom. Cloud Computing Security: From Single to Multi-clouds. In *2012*

45th Hawaii International Conference on System Sciences, pages 5490–5499. IEEE, jan 2012.

[10] Amazon. Amazon CloudWatch. http://aws.amazon.com/cloudwatch/, (Retrived: March 2, 2013).

[11] Amazon. Amazon CloudWatch Command Line Tool. http://aws.amazon.com/developertools/2534, (Retrived: March 2, 2013).

[12] Amazon. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/, (Retrived: March 2, 2013).

[13] Amazon. Amazon Relational Database Service (Amazon RDS). http://aws.amazon.com/rds/, (Retrived: March 2, 2013).

[14] Amazon. Auto Scaling Command Line Tool. http://aws.amazon.com/developertools/2535, (Retrived: March 2, 2013).

[15] Amazon. Elastic Load Balancing API Tools. http://aws.amazon.com/developertools/2536, (Retrived: March 2, 2013).

[16] Amazon. Route 53. http://aws.amazon.com/route53/, (Retrived: March 2, 2013).

[17] Alex Amies, Harm Sluiman, Qiang Guo Tong, and Guo Ning Liu. *Developing and Hosting Applications on the Cloud*. IBM Press, 1st edition, 2012.

[18] Martin Arlitt, Diwakar Krishnamurthy, and Jerry Rolia. Characterizing the scalability of a large web-based shopping system. *ACM Trans. Internet Technol.*, **1**[1]:44–69, aug 2001.

[19] Michael Armbrust, Ion Stoica, Matei Zaharia, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, and Ariel Rabkin. A view of cloud computing. *Communications of the ACM*, **53**[4]:50, apr 2010.

[20] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. *Xen and the art of virtualization*, **37**. ACM Press, New York, New York, USA, oct 2003.

[21] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing. *Future Generation Computer Systems*, **28**[5]:755–768, may 2012.

[22] Theophilus Benson, Sambit Sahu, Aditya Akella, and Anees Shaikh. A first look at problems in the cloud. In *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, page 15, jun 2010.

[23] Jing Bi, Zhiliang Zhu, Ruixiong Tian, and Qingbo Wang. Dynamic Provisioning Modeling for Virtualized Multi-tier Applications in Cloud Data Center. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, CLOUD '10, pages 370–377, Washington, DC, USA, 2010. IEEE Computer Society.

[24] Peter Bodík, Moises Goldszmidt, and Armando Fox. HiLighter: automatically building robust signatures of performance behavior for small- and large-scale systems. In *SysML'08 Proceedings of the Third conference on Tackling computer systems problems with machine learning techniques*, page 3. USENIX Association Berkeley, CA, USA, dec 2008.

[25] M. Bramfitt and H. Coles. Modular/Container Data Centers Procurement Guide: Optimizing for Energy Efficiency and Quick Deployment. Technical report, Lawrence Berkeley National Laboratory on behalf of the General Services Administration, 2011.

[26] Robert Goodell Brown. *Smoothing, Forecasting and Prediction of Discrete Time Series*. Courier Dover Publications, 2004.

[27] Sven Bugiel, Stefan Nürnberger, Ahmad-Reza Sadeghi, and Thomas Schneider. Twin Clouds: Secure Cloud Computing with Low

Latency. In Bart Decker, Jorn Lapon, Vincent Naessens, and Andreas Uhl, editors, *Communications and Multimedia Security*, **7025** of *Lecture Notes in Computer Science*, pages 32–44. Springer Berlin Heidelberg, 2011.

[28] Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski. *Cloud Computing: Principles and Paradigms*. John Wiley & Sons, Inc., Hoboken, NJ, USA, feb 2011.

[29] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, **41**[1]:23–50, jan 2011.

[30] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot A technique for cheap recovery. In *USENIX osdi'04*, page 3, dec 2004.

[31] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of EJB applications. *ACM SIGPLAN Notices*, **37**[11]:246–261, nov 2002.

[32] Xi Chen, Haopeng Chen, Qing Zheng, Wenting Wang, and Guodong Liu. Characterizing web application performance for maximizing service providers' profits in clouds. In *2011 International Conference on Cloud and Service Computing*, pages 191–198. IEEE, dec 2011.

[33] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three CPU schedulers in Xen. *SIGMETRICS Perform. Eval. Rev.*, **35**[2]:42–51, sep 2007.

[34] Y. Diao Chess, J. L. Hellerstein, S. Parekh, and J. P. Bigus. Managing Web server performance with AutoTune agents. *IBM Systems Journal*, **42**[1]:136–149, jan 2003.

[35] Carlos A. Coello Coello, Gary B. Lamont, and David A. Van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. Springer, dec 2006.

[36] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, **3**[1-2]:48–57, sep 2010.

[37] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. In *Proceedings of the VLDB Endowment*, **4**, pages 494–505, may 2011.

[38] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. Elastic VM for Cloud Resources Provisioning Optimization. In Ajith Abraham, Jaime Lloret Mauri, John F. Buford, Junichi Suzuki, and Sabu M. Thampi, editors, *First International Conference, ACC 2011, Kochi, India, July 22-24, 2011. Proceedings*, **190** of *Communications in Computer and Information Science*, pages 431–445, Kotchi, India, 2011. Springer Berlin Heidelberg.

[39] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. Dynamic Scalability and Contention Prediction in Public Infrastructure using Internet Application Profiling. In *In Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science*, Taipei, Taiwan, 2012.

[40] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. Elastic Virtual Machine for Fine-Grained Cloud Resource Provisioning. In P.Venkata Krishna, M.Rajasekhara Babu, and Ezendu Ariwa, editors, *Global Trends in Computing and Communication Systems*, **269** of *Communications in Computer and Information Science*, pages 11–25. Springer Berlin Heidelberg, 2012.

[41] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. Increasing Spot instances reliability using dynamic scalability. In *2012 IEEE*

Fifth International Conference on Cloud Computing, pages 259–261, Honolulu, Hawaii, USA, 2012.

[42] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. Reliable Approach to Sell the Spare Capacity in the Cloud. In *CLOUD COMPUTING 2012: The Third International Conference on Cloud Computing, GRIDs, and Virtualization*, page 229 to 236, Nice, France, 2012.

[43] Wesam Dawoud, Ibrahim Takounah, and Christoph Meinel. Infrastructure as a Service Security: Challenges and Solutions. In *Proceedings of the 7th International Conference on Informatics and Systems (INFOS 2010)*. IEEE Press, 2010.

[44] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. UBL: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proceedings of the 9th international conference on Autonomic computing - ICAC '12*, page 191, New York, New York, USA, sep 2012. ACM Press.

[45] Paul Dowman. Backing up your MySQL database to S3. http://pauldowman.com/2009/02/08/mysql-s3-backup/, (Retrived: March 2, 2013).

[46] Matthias Ehrgott. *Multicriteria Optimization*. Springer, 2005.

[47] N. Gandhi, D.M. Tilbury, Y. Diao, J. Hellerstein, and S. Parekh. MIMO control of an Apache web server: modeling and controller design. In *Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301)*, pages 4922–4927. American Automatic Control Council, 2002.

[48] Yi Ge, Chen Wang, Xiaowei Shen, and Honesty Young. A database scale-out solution for emerging write-intensive commercial workloads. *ACM SIGOPS Operating Systems Review*, **42**[1]:102, jan 2008.

[49] Hamoun Ghanbari, Cornel Barna, Marin Litoiu, Murray Woodside, Tao Zheng, Johnny Wong, and Gabriel Iszlai. Track-

ing adaptive performance models using dynamic clustering of user classes. *SIGMETRICS Perform. Eval. Rev.*, **39**[3]:15, dec 2011.

[50] Michele Girola, Alessio M. Tarenzio, Mark Lewis, and Marian Friedman. *IBM Data Center Networking: Planning for virtualization and cloud computing.* IBM - redbooks, 2011.

[51] Jordi Guitart, Jordi Torres, and Eduard Ayguadé. A survey on performance management for internet applications. *Concurrency and Computation: Practice and Experience*, **22**[1]:68–106, 2010.

[52] Ajay Gulati, Ganesha Shanmuganathan, Anne Holler, and Irfan Ahmad. Cloud-scale resource management: challenges and techniques. In *HotCloud'11 Proceedings of the 3rd USENIX conference on Hot topics in cloud computing Pages 3-3*, page 3. USENIX Association Berkeley, CA, USA, jun 2011.

[53] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, pages 342–362, New York, NY, USA, 2006. Springer-Verlag New York, Inc.

[54] Rui Han, Li Guo, Moustafa Ghanem, and Yike Guo. Lightweight Resource Scaling for Cloud Applications. In *CCGRID*, pages 644–651. IEEE, 2012.

[55] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems.* John Wiley & Sons, 2004.

[56] Jin Heo, Xiaoyun Zhu, Pradeep Padala, and Zhikui Wang. Memory Overbooking and Dynamic Control of Xen Virtual Machines in Consolidated Environments. In *Proceedings of IFIPIEEE Symposium on Integrated Management IM09 miniconference*, pages 630–637. IEEE, 2009.

[57] Waheed Iqbal, Matthew N. Dailey, and David Carrera. SLA-Driven Dynamic Resource Management for Multi-tier Web Applications in a Cloud. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 832–837. IEEE, may 2010.

[58] David Jaeger, Konrad-Felix Krentz, Matthias Richly, Wesam Dawoud, Ibrahim Takouna, and Christian Willems. Xen Episode IV: The Guests still Strike Back. Technical report, Hasso Plattner Institute, 2011.

[59] Wayen A. Jansen. Cloud Hooks: Security and Privacy Issues in Cloud Computing. In *2011 44th Hawaii International Conference on System Sciences*, pages 1–10. IEEE, jan 2011.

[60] Gueyoung Jung, Kaustubh R. Joshi, Matti A. Hiltunen, Richard D. Schlichting, and Calton Pu. Generating Adaptation Policies for Multi-tier Applications in Consolidated Server Environments. In *2008 International Conference on Autonomic Computing*, pages 23–32. IEEE, jun 2008.

[61] Vikram Kanodia and Edward W. Knightly. Ensuring latency targets in multiclass web servers. *IEEE Transactions on Parallel and Distributed Systems*, **14**[1]:84–93, jan 2003.

[62] Emre Kcman. *Using Statistical Monitoring to Detect Failures in Internet Services*. PhD thesis, Stanford University, 2005.

[63] Gunjan Khanna, Kirk Beaty, Gautam Kar, and Andrzej Kochut. Application Performance Management in Virtualized Server Environments. In *2006 IEEE/IFIP Network Operations and Management Symposium NOMS 2006*, pages 373–381. IEEE, 2006.

[64] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. Task-aware virtual machine scheduling for I/O performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 101–110, New York, NY, USA, 2009. ACM.

[65] Stephan Kraft, Giuliano Casale, Diwakar Krishnamurthy, Des Greer, and Peter Kilpatrick. IO performance prediction in consolidated virtualized environments. *SIGSOFT Softw. Eng. Notes*, **36**[5]:295–306, sep 2011.

[66] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. CloudCmp: comparing public cloud providers. In *Proceedings of the 10th annual conference on Internet measurement - IMC '10*, page 1, New York, New York, USA, nov 2010. ACM Press.

[67] Jack Li, Qingyang Wang, Deepal Jayasinghe, Simon Malkowski, Pengcheng Xiong, Calton Pu, Yasuhiko Kanemasa, and Motoyuki Kawaba. Profit-Based Experimental Analysis of IaaS Cloud Performance: Impact of Software Resource Allocation. In *2012 IEEE Ninth International Conference on Services Computing*, pages 344–351. IEEE, jun 2012.

[68] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. *International Conference on Autonomic Computing*, pages 1–10, 2010.

[69] Harold C. Lim, Shivnath Babu, Jeffrey S. Chase, and Sujay S. Parekh. Automated control in cloud computing: challenges and opportunities. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds - ACDC '09*, page 13, New York, New York, USA, jun 2009. ACM Press.

[70] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K Panda. High performance VMM-bypass I/O in virtual machines. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, page 3, Berkeley, CA, USA, 2006. USENIX Association.

[71] Xue Liu, Jin Heo, and Lui Sha. Modeling 3-Tiered Web Applications. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*,

MASCOTS '05, pages 307–310, Washington, DC, USA, 2005. IEEE Computer Society.

[72] Xue Liu, Jin Heo, Lui Sha, and Xiaoyun Zhu. Queueing-Model-Based Adaptive Control of Multi-Tiered Web Applications. *IEEE Transactions on Network and Service Management*, **5**[3]:157–167, 2008.

[73] Xue Liu, Lui Sha, Yixin Diao, Steven Froehlich, Joseph L. Hellerstein, and Sujay Parekh. Online Response Time Optimization of Apache Web Server. In *IWQoS'03 Proceedings of the 11th international conference on Quality of service*, pages 461–478. In IWQoS, 2003.

[74] Simon Malkowski, Yasuhiko Kanemasa, Hanwei Chen, Masao Yamamoto, Qingyang Wang, Deepal Jayasinghe, Motoyuki Kawaba, and Calton Pu. Challenges and Opportunities in Consolidation at High Resource Utilization: Non-monotonic Response Time Variations in n-Tier Applications. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 162 – 169, Honolulu, Hawaii, USA, 2012.

[75] Simon J. Malkowski, Markus Hedwig, Jack Li, Calton Pu, and Dirk Neumann. Automated control for elastic n-tier workloads based on empirical modeling. In *Proceedings of the 8th ACM international conference on Autonomic computing - ICAC '11*, page 131, New York, New York, USA, jun 2011. ACM Press.

[76] Ming Mao and Marty Humphrey. A Performance Study on the VM Startup Time in the Cloud. In *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*, Honolulu, HI, USA, 2012. IEEE.

[77] Tai Jin Martin Arlitt. Workload Characterization of the 1998 World Cup Web Site. Technical report, IEEE Network, 1999.

[78] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. Quantifying

the performance isolation properties of virtualization systems. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[79] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical report, U.S Department of Commerce, 2011.

[80] Daniel A. Menascé. Load Testing of Web Sites. Technical report, IEEE Internet Computing, 2002.

[81] Microsoft. Windows Azure. http://www.windowsazure.com/, (Retrived: March 2, 2013).

[82] Rich Miller. A Look Inside Wikipedia's Infrastructure. http://www.datacenterknowledge.com/archives/2008/Jun/24/a_look_inside_wikipedias_infrastructure.html, (Retrived: March 2, 2013).

[83] David Mosberger and Tai Jin. httperf - A Tool for Measuring Web Server Performance. In *In First Workshop on Internet Server Performance*, pages 59–67, 1998.

[84] MySql. MySql Proxy. https://launchpad.net/mysql-proxy, (Retrived: March 2, 2013).

[85] Jakob Nielsen. *Usability Engineering*. Academic Press, Boston, 1993.

[86] J. Oberheide, E. Cooke, and F. Jahanian. Empirical exploitation of live virtual machine migration. *In Proc. of BlackHat DC convention*, 2008.

[87] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. *European Conference on Computer Systems*, pages 13–26, 2009.

[88] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth

Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys '07 Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 289–302. In Proceedings of the European Conference on Computer Systems, 2007.

[89] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. Understanding Performance Interference of I/O Workload in Virtualized Cloud Environments. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 51–58. IEEE, jul 2010.

[90] Jia Rao. *Autonomic management of virtualized resources in cloud computing*. PhD thesis, Wayne State University Dissertations, 2011.

[91] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment*, **3**[1-2]:460–471, sep 2010.

[92] Apache2 Server. Apache2 Server. http://httpd.apache.org/, (Retrived: March 2, 2013).

[93] Abhishek B Sharma, Ranjita Bhagwan, Monojit Choudhury, Leana Golubchik, Ramesh Govindan, and Geoffrey M Voelker. Automatic request categorization in internet services. *SIGMETRICS Perform. Eval. Rev.*, **36**[2]:16–25, aug 2008.

[94] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. Kingfisher: A system for elastic cost-aware provisioning in the cloud. Technical report, Univ. of Massachusetts Amherst IBM Research, 2010.

[95] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11*, pages 1–14, New York, New York, USA, oct 2011. ACM Press.

[96] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil,

Armando Fox, and David Patterson. Cloudstone: Multi-Platform, Multi-Language Benchmark and Measurement Tools for Web 2.0. In *Cloud Computing and Its Applications*, 2008.

[97] Sebastian Stadil, Igor Savchenko, Alex Kovalyov, and Michel Galibert. Scalr, 2012.

[98] Christopher Stewart, Terence Kelly, and Alex Zhang. Exploiting nonstationarity for performance prediction. *SIGOPS Oper. Syst. Rev.*, **41**[3]:31–44, mar 2007.

[99] Julian T J Midgley. Autobench. http://www.xenoclast.org/autobench/, (Retrived: March 2, 2013).

[100] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*, Macau, China, 2012.

[101] Matt Tavis. Web Application Hosting in the AWS Cloud: Best Practices, 2010.

[102] Bhuvan Urgaonkar. Dynamic resource management in Internet hosting platforms, 2005.

[103] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. *ACM SIGMETRICS Performance Evaluation Review*, **33**[1]:291–302, jun 2005.

[104] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. Analytic modeling of multitier Internet applications. *ACM Trans. Web*, **1**[1], may 2007.

[105] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of

multi-tier Internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, **3**[1], 2008.

[106] Marten Van Dijk and Ari Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. In *HotSec'10 Proceedings of the 5th USENIX conference on Hot topics in security*, pages 1–8, aug 2010.

[107] Luis M. Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. In *ACM SIGCOMM Computer Communication Review*, **41**, pages 45–52, jan 2011.

[108] Venkatanathan Varadarajan, Benjamin Farley, Thomas Ristenpart, and Michael M. Swift. Resource-Freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense). In *CCS 2012: The 19th ACM Conference on Computer and Communications Security*, Raleigh, NC, USA, 2012.

[109] Thiemo Voigt and Per Gunningberg. Adaptive Resource-based Web Server Admission Control. In *Proceedings of the Seventh International Symposium on Computers and Communications (ISCC'02)*, page 219. IEEE Computer Society Washington, DC, USA 2002, jul 2002.

[110] Thiemo Voigt, Renu Tewari, Douglas Freimuth, and Ashish Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *USENIX 2001*, pages 189–202, jun 2001.

[111] Carl A Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, **36**[SI]:181–194, dec 2002.

[112] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of amazon EC2 data center. In *2010 IEEE INFOCOM Conference*, pages 1163–1171, mar 2010.

[113] Zhikui Wang, Xue Liu, Alex Zhang, Christopher Stewart, Xiaoyun Zhu, Terence Kelly, and Sharad Singhal. AutoParam: Automated Control of Application-Level Performance in Virtualized Server Environments. In *In Proceedings of the 2nd IEEE International Workshop*

on Feedback Control Implementation in Computing Systems and Networks (FeBid, Munich, Germany, 2007.

[114] Zhikui Wang, Xiaoyun Zhu, Sharad Singhal, and Hewlett Packard. Utilization and SLO-based Control for Dynamic Sizing of Resource Partitions, 2005.

[115] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *4th USENIX Symposium on Networked Systems Design & Implementation*, page 17. NSDI, 2007.

[116] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. SLA-based admission control for a Software-as-a-Service provider in Cloud computing environments. *Journal of Computer and System Sciences*, **78**[5]:1280–1299, sep 2012.

[117] Lenar Yazdanov and Christof Fetzer. Vertical scaling for prioritized VMs provisioning. In *International Conference on Cloud and Green Computing*. IEEE Computer Society, 2012.

[118] Lamia Youseff, Maria Butrico, and Dilma Da Silva. Toward a Unified Ontology of Cloud Computing. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, nov 2008.

[119] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. In *Proceedings of the Fourth International Conference on Autonomic Computing*, pages 27—-, Washington, DC, USA, 2007. IEEE Computer Society.

[120] Zhen Zhang and Shanping Li. Automatic Fine-Grained Transaction Categorization for Multi-tier Applications. *Cyber-Enabled Distributed Computing and Knowledge Discovery, International Conference on*, **0**:130–138, 2011.

[121] Huican Zhu, Hong Tang, and Tao Yang. Demand-driven service differentiation in cluster-based network servers. In *Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2001*, **2**, pages 679–688, 2001.

[122] Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, Pradeep Padala, and Kang Shin. What does control theory bring to systems research? *SIGOPS Oper. Syst. Rev.*, **43**[1]:62–69, jan 2009.

[123] Xiaoyun Zhu, Zhikui Wang, and Sharad Singhal. Utility-Driven Workload Management using Nested Control Design. In *2006 American Control Conference*, pages 6033–6038. American Control Conference, 2006.