VRIJE UNIVERSITEIT

PERFORMANCE GUARANTEES FOR WEB APPLICATIONS

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op donderdag 29 maart 2012 om 15.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Dejun Jiang

geboren te Zhenjiang, China

promotor:	prof.dr.ir. M.R. van Steen
copromotoren:	dr.ir. G.E.O. Pierre
	prof.dr. C.H. Chi

PERFORMANCE GUARANTEES FOR WEB APPLICATIONS

DEJUN JIANG

Parts of Chapter 3 have been published in *Proceedings of the 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing*.
Parts of Chapter 4 have been published in *Proceedings of the 17th International World Wide Web Conference*.
Parts of Chapter 5 have been published in *Proceedings of the 19th International World Wide Web Conference*.
Parts of Chapter 6 have been published in *Proceedings of the 2nd USENIX Conference on Web Application Development*.

VU University Press
De Boelelaan 1105
1081 HV Amsterdam
The Netherlands

E-mail: info@vu-uitgeverij.nl
Website: www.vu-uitgeverij.nl

ISBN 978 90 8659 601 0
NUR 980

ACKNOWLEDGEMENTS

I still remembered the day when I first arrived in Amsterdam: December 2nd, 2006. Starting from that day, I began the exploring journey to pursue a Ph.D. degree. Now it is time to thank the people that have helped me during this journey.

First of all, I would like to thank my advisors: Maarten van Steen, Guillaume Pierre, and Chi-Hung Chi. Maarten has always been an energetic advisor to me. Each time I discussed my work with Maarten, he would quickly grasp the key point of the problem and provide positive suggestions to me. I have to thank Maarten to teach me to think problems within a full picture.

I owe the most appreciation to Guillaume Pierre. Guillaume is a great advisor. He not only supervises me the detailed research questions, but also teaches me the methods to do research independently. The most important thing I learned from him is the manner to employ experimental data to support any argument. I kept this in my mind through the whole PhD study and benefited a lot. In the last five years, Guillaume has kept asking me "what is next" to encourage me to consider problems thoroughly and comprehensively. I owe a great deal to this question he taught me as it conducts a constantly progressive research work to me. Guillaume has remarkably been patient with my paper writing. For each of my paper drafts, he has done a lot to make them more readable. I would like to thank Guillaume again for making me understand how to do good research.

In addition to being a great advisor, Guillaume has also been a great friend to me. During the initial days I arrived in Amsterdam, he helped me to get through the culture shock between the western and the eastern. Of course, thank Guillaume and Caroline to treat me to delicious French food each time I was in Amsterdam.

Also I would like to thank Chi-Hung Chi. Prof. Chi first served as my master advisor in Tsinghua University and then continued to supervise me during my PhD study. He has presented a hard-working example to me during the last five years. I learned from Prof. Chi several good characters to be a qualified researcher.

Now, to my friends and colleagues at the Vrije Universiteit. First, I owe a great deal to Zhou Wei and Chen Shuo. The three of us worked together and made the five years stay more fun with a number of interesting things, like traveling and

sporting. Also, the discussions with the two guys usually made me find explorable aspects of my research work. Then I would like to thank Guido Urdaneta. I still remembered that the first time I went to the ASCI conference, Guido was so nice to take care of my trip to the conference venue such that I would not get lost. Especially, in the final stage of my thesis writing, Guido helped me with the thesis formatting! Also, I would like to thank Ana-Maria Oprescu, who helped me to deal with a number of administrative procedures for printing my thesis and contacting the Pedel. Of course, many thanks to Timo van Kessel for writing the Dutch summary and the Dutch brief introduction for the thesis publicity.

Finally, I owe this thesis to my family. My father and mother have constantly given me a lot support to finish my PhD study. During the last five years, they worked hard to take care of the family financial aspects. Thanks to their strong support, I could focus on my study and get through this journey. Thank you!

Dejun Jiang
Beijing, China. January 2012.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Users are increasingly demanding for responsive Web applications. A survey from 2006 revealed that 62% Web users were willing to wait only 6 seconds or less for a single page load before they leave the web site [Jupiter Research, 2006]. A more recent research (2009) indicated that this performance expectation has become more demanding as 83% Web users expected a Web page to load in 3 seconds or less [Forrester Research, 2009]. In addition, this research also found that 79% of online shoppers who visited an underperforming Web site were likely not to buy from that site. Obviously, performance guarantees for Web applications are business-critical.

One important performance metric is the response time of a Web application. The response time can be split into three parts: client-side latency, network delay, and server-side latency. Recently, Web applications started employing client-side codes, such as JavaScript, to enrich application features [Paulson, 2005]. Client-side latency refers to the time used to execute client-side code. The research community has made efforts to address several client-side performance issues, such as JavaScript runtime behavior investigations for improving the representativeness of performance benchmark suites [Ratanaworabhan et al., 2010], remote monitoring for client-side performance diagnosing [Kıcıman and Livshits, 2007], and JavaScript performance optimizations by trace-based just-in-time compiler [Chang et al., 2009]. In the ICT industry, the browser war among various vendors also targets JavaScript performance improvement for a major part [Shankland, 2009]. Client-side latency mainly depends on two factors: application client-side code and specific mechanisms built in each Web browser. From the perspective of Web hosting providers, these two factors are beyond their controllable scope.

Network delay refers to the transmission time of a request's response from the server to the client over a network such as the Internet. Various techniques, such as edge computing [Davis et al., 2004], data caching [Sivasubramanian et al., 2006],

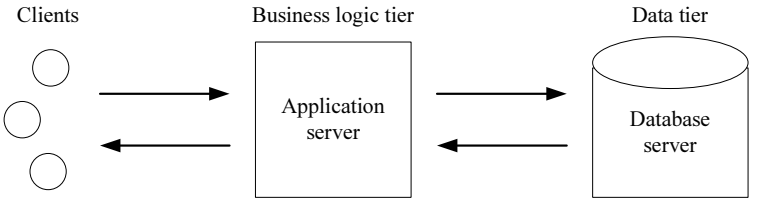


Figure 1.1: A simple 2-tier Web application model

and data replication [Sivasubramanian et al., 2005] have been proposed to reduce these delays. Commercial products such as Akamai CDN and Amazon CloudFront are also available for guaranteeing the best possible access performance [Akamai, 2006; Amazon CloudFront, 2011].These academic and industrial efforts work together to significantly reduce the network delay incurred by Web applications, and have been quite successful.

Though optimizing client-side latency and network delay is important, we cannot guarantee performance of a Web application unless its server-side latency is also under control. For instance, previous experiments showed that service-side latency could account for nearly 50% of the end-to-end latency of a Web application [Andreolini et al., 2004]. As Web applications continue to become more complex, we can only expect server-side latency to increase. Server-side latency refers to the residence time of an incoming request waiting for its response at the server. For instance, a typical Web application consists of a business-logic tier and a data tier as shown in Figure 1.1. The business-logic tier may be deployed on an application server while the data tier is often deployed on a database server. The server-side latency includes the time of executing application code at the application server and the time of fetching data from the database server.

Guaranteeing server-side Web application performance is made difficult by the fact that Web application workloads are fluctuating and highly unpredictable [Arlitt and Jin, 2000; Gill et al., 2007]. The unpredictability and fluctuations introduce two important demands for the hosting system. First, a Web-application architecture must be ready to accommodate arbitrary levels of load. Second, it must be capable of adjusting its own capacity to support fluctuating Web traffic.

On the one hand, given that Web traffic is unpredictable, one cannot predict the maximum workload a Web application will receive in advance. Meanwhile, Web application providers aim at attracting as many users as possible for potentially growing business benefits. Therefore, a Web application needs to be scalable. A scalable Web application is capable of handling arbitrary levels of traffic by adding resources while sustaining reasonable performance. However, constructing a scalable Web application is nontrivial in that it requires careful partitioning of

both business-logic tier and data tier [Shoup and Pritchett, 2006]. We return to this issue later in this thesis.

On the other hand, the fluctuations of Web application workload make it impossible to plan "proper" fixed hosting resource capacity at minimal resource cost. Cost-sensitive Web application providers, for example small and medium-sized providers, expect a cost-effective manner to host their applications. By applying the utility computing model to Web-application hosting and varying the number of resources Web applications use according to the current load, application providers can expect reducing their costs.

Utility computing provides a model of packaging computing resources as a metered service [Wikipedia, 2011c]. Since year 2000, IT providers have been making efforts to develop products and services to implement utility computing model in computer clusters and data centers [Sun Cloud, 2000; Kallahalla et al., 2004]. Recently, cloud computing started applying utility computing by provisioning resources in a pay-as-you-go manner. In clouds, resources such as computation, storage, and network are rented as services and charged by usage [Amazon EC2, 2011; Rackspace, 2011]. The utility computing model facilitates dynamic resource provisioning for Web applications to handle varying resource demands. However, efficient dynamic resource provisioning faces challenges from both Web applications and hosting environments [Dejun et al., 2009, 2010]. This brings us to the central research question of this thesis: *how to guarantee the server-side performance for Web applications in a cost-effective manner*.

This thesis uses the server-side average response time as the Web-application performance measurement. Other performance metrics, such as percentiles of the response time, are also useful for performance guarantees [Menascé, 2002]. We believe that our techniques can be extended to support such metrics. The issue of guaranteeing server-side performance can be translated into sustaining reasonable average response time for Web applications under fluctuating traffic. A reasonable response time is defined to be under the maximum average response time within which an application should finish processing an incoming request. Web application providers usually set this maximum response time in their Service Level Objectives (SLOs) [Jin et al., 2002].

Besides choosing performance metrics, this thesis uses the number of utilized machines as the cost measurement. A utilized machine can be either a dedicated physical machine in a cluster or a virtual machine in a cloud. The number of machines can be further translated into the monetary cost if given the charging price.

This thesis mainly involves two aspects of research efforts to address our central research question: i) constructing a scalable Web application architecture; and ii) designing dynamic resource provisioning systems.

Constructing a scalable Web application can be done in two main ways: scale-up and scale-out [Wikipedia, 2011b]. As for scaling the Web application shown in Figure 1.1, scale-up means adding more capacity, such as CPU speed and memory size, to individual application servers and database servers. In contrast, scale-out means adding more servers to the two tiers. Scale-out outperforms scale-up when the performance/cost ratio is concerned for Web applications [Michael et al., 2007]. Scale-up also has a hard limit by the scale of the hardware while scale-out allows to continuously add resources. Therefore, we construct a scalable Web application architecture by using scale-out techniques in this thesis.

Adding more servers to the business-logic tier of a Web application can improve the performance by alleviating the workload addressed to each individual server at that tier [Ranjan et al., 2002]. However, adding more servers to the data tier cannot always improve the performance of that particular tier under arbitrary levels of load [Zhou et al., 2008]. Partial database replication [Groothuyse et al., 2007], careful data partition and placement [Gao et al., 2003], allow improved scalability of the data tier through adding more resources. However, the coarse partition granularity limits the scalability extent of current scaling techniques. In this thesis, we show the potential scalability of Web applications resulted from finer data-partition granularity.

Though a scalable Web-application architecture provides promising mechanisms for guaranteeing the performance of Web applications, Web applications still face the issue of fluctuating traffic. Over-provisioning Web applications according to the peak workload can result in inefficient resource usage while under-provisioning creates a risk of violating SLO [Chandra et al., 2003]. The most straightforward technology used to guarantee performance for Web applications under fluctuating traffic is dynamic resource provisioning [Urgaonkar et al., 2008]. This technology consists of adding extra resources to a Web application when its response time is close to violating its SLO, and removing underutilized resources from a Web application with retaining its SLO.

Unfortunately complex Web applications and heterogeneous hosting environments challenge current dynamic resource provisioning techniques. On the one hand, current Web applications are not designed as a monolithic 3-tier application. For instance, the Web application used to generate Web pages of Amazon.com consists of hundreds of services [Vogels and Gray, 2006]. It is hard to figure out the performance bottleneck within such applications that consist of multiple interacting services. It is even harder to handle this issue by dynamically and efficiently provisioning resources. On the other hand, heterogeneous physical machines and virtual machines in data centers and clouds result in performance heterogeneity of virtual hosting resources [Dejun et al., 2009; Schad et al., 2010]. This feature also limits the applicability of current resource provisioning

techniques that assume the existence of homogeneous underlying resources.

THESIS CONTRIBUTION AND OUTLINE

In this thesis, we contribute new ideas to the two aspects of efforts: i) constructing a scalable Web application architecture; and ii) designing dynamic resource provisioning systems. Our contributions map to the chapters as follows.

Related work (Chapter 2)

Performance guarantees for Web applications are well studied in the research community. This chapter presents a comprehensive survey to introduce research efforts in this area.

Challenges (Chapter 3)

Guaranteeing Web application performance is difficult for several reasons. First, a monolithic 3-tier Web application is not scalable if we need to guarantee performance under arbitrary levels of load [Zhou et al., 2008]. This chapter shows the negative impact of a nonscalable architecture on performance guarantees of Web applications. Second, today's complex Web applications and heterogeneous hosting environments challenge current dynamic resource provisioning techniques. This chapter discusses the limitations of current resource provisioning techniques to complex multi-service Web applications. In addition, this chapter presents an investigation on the performance heterogeneity of cloud hosting environments. The challenges and implications on resource provisioning introduced by heterogeneous clouds are also discussed in this chapter.

Making Web applications scalable (Chapter 4)

Constructing a scalable Web-application architecture is not a novel topic in the research community. Techniques used to scale multi-tier Web applications are well understood. A typical multi-tier Web application consists of a presentation tier, a business-logic tier and data tier. Many effective techniques have been proposed to scale the presentation tier and business-logic tier. However, previous studies show that scaling the data tier remains a challenge. This chapter proposes general guidelines and techniques to construct a Web application in a multi-service architecture such that one can scale each service more easily. We demonstrate that by reconstructing the data tier of a Web application into multiple data services, one can achieve a significant scalability improvement.

Resource provisioning for multi-service Web applications (Chapter 5)

Most real-world scalable Web applications consist of multiple interacting services. Although dynamic resource provisioning of multi-tier Web application is well studied, few research efforts contribute to provisioning multi-service Web applications. Unlike multi-tier Web applications, services in a multi-service Web application usually interact with each other, which results in the application ar-

chitecture following a directed acyclic graph. One cannot simply apply current model-based techniques to provision multi-service Web applications. This chapter proposes a decentralized approach to let each service propagate its performance objective in case of adding or removing resources within the application. The front-end service is responsible for aggregating these performance objectives and to decide on resource assignment. We show that our proposed mechanism effectively guarantees performance of multi-service Web applications with efficient resource usage.

Resource provisioning in Cloud environments (Chapter 6)

Utility computing helps to reduce resource costs when one applies dynamic resource provisioning techniques to guarantee performance of Web applications. When moving to real-world hosting environments that implement the utility computing model, one has to face the reality of a heterogeneous resource environment. A cloud is a typical hosting environment that exhibits the heterogeneity of underlying resources. In addition to providing heterogeneous virtual instances possessing different types of capacities, the same type of virtual instances also behave differently in a cloud. This chapter proposes performance correlation techniques to derive the performance of a Web application based on a pre-profiled calibration machine. Our technique is capable of predicting the performance of newly acquired virtual instances without actually running an application-specific load on it. We demonstrate that one can guarantee performance of Web applications in real-world heterogeneous clouds in a cost-effective manner by adopting our techniques.

Conclusion (Chapter 7)

We conclude the main contributions of our thesis work in this chapter. We further discuss possible future work topics.

Chapter 2

Related work

When a user sends a request to a Web application, a number of factors affect the end-to-end response time she observes. Figure 2.1 shows the component view of the end-to-end response time of a single request-response pair. DNS lookup is necessary to find the server's network address from the DNS server or the DNS cache. This time needed for lookup can vary tremendously due to a DNS cache miss, however in most cases it remains below 500 ms [Cohen and Kaplan, 2001]. The initial connection time is the time used to establish a TCP connection. Previous work showed that this time would often be below 200 ms [Zari et al., 2001]. These two time components are one-time costs at the beginning of each connection. The requests transferred over an established connection do not incur such latency again. The time used to send requests can vary depending on the request size. However, requests to Web applications such as e-commerce usually have small sizes. Therefore, the time to send requests is rarely evaluated in the end-to-end response time. The time to the first byte refers to the time needed to receive the first byte of the response after one sends a request. This component includes

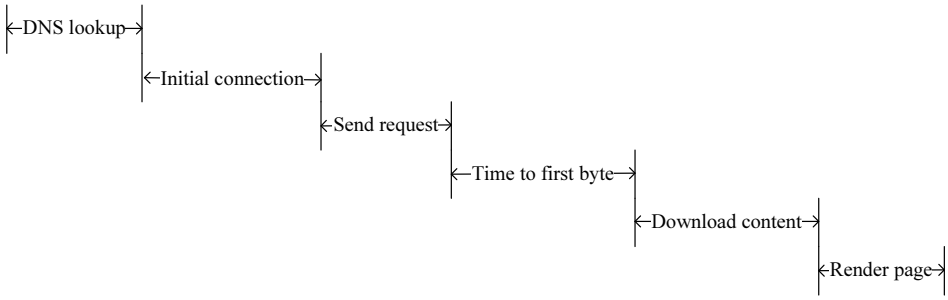


Figure 2.1: Component view of Web application performance

the time a server uses to handle the incoming request. The subsequent time for the client to receive the entire response is the content download time. This time component is affected by the response size and the network delay between the server and the client. Finally, the client needs time to render the received response to end users, which includes parsing response data stream, executing client-side code and displaying response results.

In the past few years, various research efforts have been made to improve these individual components. For instance, replication is widely used to reduce the time to download responses. Dynamic resource provisioning can also maintain reasonable server-side response times under fluctuating traffic. However, analyzing and comparing these efforts is difficult as these works address different aspects related to the performance of Web applications. To this end, we propose a framework that classifies these components into three main aspects: client-side latency, network-induced delays and server-side latency. This framework helps us to isolate the impact of different factors on the performance of a Web application. We then introduce important research results on each aspect of this framework. We particularly focus our attention on research works addressing server-side performance guarantees. We finally survey these works in detail from two different angles: available mechanisms for enabling performance guarantees on Web application server side, and dynamic controls for adjusting Web applications to sustain performance.

2.1 Framework

Web application performance can be split into three components: client-side latency, network delay and server-side latency. Within different Web applications, the three components have different weighted impacts on the performance of the whole application. For instance, e-commerce applications like Amazon.com include complex business logic (e.g. promoting new books according to a customer's historical order preference) and may therefore spend a large fraction of response time due to back-end server processing. Image-sharing applications like Flickr need to deliver image files of a potentially large size to end users across different geographical locations. Consequently, network delays can also play a significant role in the end-to-end response time. To offer performance guarantees, one therefore needs to understand the impact of each component on the end-to-end performance of a Web application.

2.1.1 Client-side latency

Web applications can use either Web browsers or customized clients to receive, process and render the responses. The behavior of customized clients varies

from vendor to vendor. Performance guarantees for customized clients are client-specific and out of reach for Web application providers. Therefore, we do not discuss the performance issues of customized clients in this thesis.

The compute-intensive tasks performed by Web browsers mainly include layout, painting, CSS parsing and JavaScripts execution. As today's Web applications include a large number of JavaScript components to enrich client-side experience, the performance of JavaScript plays an important role in the whole latency of Web browsers to load pages. Major Web browser vendors work hard to improve their browsers' performance in a so-called "browser war" [Wikipedia, 2011a]. In order to provide an objective comparison of the JavaScript performance of different browsers, the industry provides JavaScript benchmarks. For instance, WebKit SunSpider [SunSpider, 2011] and V8 from Google [Google V8, 2011] are widely used to evaluate JavaScript performance. However, these two most commonly used benchmarks were shown not to be able to represent real JavaScript workloads [Richards et al., 2010; Ratanaworabhan et al., 2010]. Ratanaworabhan et al. [2010] examined the representativeness of these two benchmarks by measuring the JavaScript behavior of real Web applications, such as Amazon, Facebook, Gmail and Yahoo. Their results show that these two benchmark suites fail to capture certain features of real workloads. For instance, the benchmarks are small while the JavaScript code of real Web applications is one to two orders of magnitude larger. In addition, the benchmarks are compute-intensive and batch-oriented, while the real workloads are event-driven and functions are typically short-lived. In order to generate representative workloads with a high degree of fidelity compared to real workloads, Richards et al. [2011] proposed JSBench, a tool that is able to synthesize JavaScript workload from real Web applications and then replay the workload. Although JSBench does not provide a "perfect" JavaScript performance benchmark, the proposed record-and-replay approach enables the community to improve the representativeness of the benchmarks.

Another important issue related to Web application client-side latency is JavaScript performance monitoring within browsers. Large and complex Web applications can consist of tens of thousands of lines of JavaScript executing in the user's browser [Kıcıman and Livshits, 2007]. However, Web application developers lack visibility of client-side behavior. This prevents developers to directly react to the JavaScript performance problems encountered by end users. Kıcıman and Livshits [2007] built AjaxScope, a proxy platform to instrument JavaScript-based Web applications on-the-fly when they are being sent to a user's browser. AjaxScope can inject arbitrary instrumentation code to monitor and report the dynamic runtime behavior of Web applications without imposing significant overhead to a browser. By applying AjaxScope one can expect comparative evaluations of JavaScript execution behaviors across different browsers. These techniques can be

used to guide browser vendors and Web application developers to improve client-side latency of Web applications.

2.1.2 Network-induced delays

Network delays depend on both network bandwidth and network latency. The performance of broadband networks reduces the impact of network bandwidth on the download time of small Web pages as typically delivered by Web applications. However, network latency still plays an important role on transmitting Web application responses. Over non-congested network, the network latency purely depends on the physical properties of network links that form the routing path. Therefore, efforts have been made to move Web applications closer to end users so that responses can be transmitted over short network distance. A typical hosting system used to reduce network delay is a Content Delivery Network (CDN) [Akamai, 2006; Amazon CloudFront, 2011]. CDNs replicate content from the origin server to cache servers placed across different geographical locations such that application providers can deliver contents to end users in a timely manner using nearby cache servers [Pathan and Buyya, 2007].

In order to serve users globally, caching and replication are two widely used techniques. Caching popular static contents in edge servers across multiple geographical locations is the simplest way to deliver contents in a timely manner [Novella et al., 2004]. However, today's Web applications are increasingly interactive and dynamically generate personalized contents for each end user. Simple content caching techniques help little to reduce network delays of such applications as the responses are generated at the original servers. Academia and industry proposed the edge-computing model to replicate the application codes into edge servers such that end users can take advantage of nearby edge servers to request user-specific contents with low delay [Davis et al., 2004].

Though edge computing can reduce network delays by letting edge servers generate responses for nearby users, the data access of Web applications still incurs wide-area network latency as it needs to request data from a central database [Sivasubramanian et al., 2007]. Database replication is an effective technique to address this problem by placing databases in edge servers to reduce network delay. Full database replication is easy to provide. However, when the number of database replicas increases, we observe performance degradation as database updates need to propagate to all replicas for maintaining consistency. Instead of fully replicating a database, partial database replication can help to alleviate performance degradation. Sivasubramanian et al. [2005] proposed GlobeDB that moved part of the underlying database to edge servers such that end users could experience low network delay. Gao et al. [2003] did similar work to replicate both business logic code and database to edge servers. However, their work requires the application-

specific knowledge to handle replication correctly.

In addition to replicating a database to edge servers, data caching can also help to reduce network delays by caching query results in edge servers. Data caching can be split into two categories. On the one hand, content-aware caching applies the concept of semantic caching and caches the record results of database queries provided by the Web application [Dar et al., 1996; Luo and Naughton, 2001; Amiri et al., 2003]. The edge server runs a database to store these structured responses. When the edge server receives a new query, it checks whether it contains enough local data to answer the query correctly. In case it lacks enough data, the query is sent to the origin database to process. For instance, a range query "SELECT c_name FROM customer WHERE c_id > 2000" fetches the names of customers whose IDs are greater than 2000. However, if the cached local data only maintains records of customers whose IDs are less than 1000, one needs to send this query to the origin database to fetch data. Consequently, this caching technique requires executing a so-called query containment check to see if query can be answered from cached data. However, query containment checks are compute intensive. As an alternative, Sivasubramanian et al. [2006] proposed GlobeCBC to provide content-blind caching. This technique blindly caches results to a specific query, regardless whether that query overlapped previous queries. Therefore, it can return results immediately in case of a cache hit. However, content-blind caching does not merge cached results so it can lead to storing redundant information.

These above techniques can be combined with other techniques used to optimize server-side performance to improve the end-to-end performance of the whole Web application.

2.1.3 Server-side latency

Unlike client-side latencies and network delays, server-side performance can be controlled by hosting providers. The server-side Web application serves incoming requests and processes business logic by running the application code on application servers. Meanwhile Web applications need to store business information on database servers for persistent usage. Therefore, Web applications are organized in a tier-based architecture to split different concerns when developing and maintaining Web applications. For instance, a business-logic tier is designed to handle business-level logic, while a data tier takes care of storing and accessing business data. One can monitor and diagnose individual tiers in case of encountering performance problem.

Providing performance guarantees on the server-side latency is made difficult by two factors. First, Web applications keep growing in complexity. The Web application architecture therefore becomes fundamental to guarantee performance guarantees as it has to be inherently scalable. Second, a Web application faces the

fact that its workload is highly unpredictable and fluctuating. A Web application has to be capable of dynamically adapting to varying workloads. Therefore, addressing the server-side performance guarantee involves three significant issues: (i) making the business-logic tier scalable; (ii) making the data tier scalable; and (iii) providing dynamic controls to adapt Web applications to their workload.

In the rest of this chapter, we survey research on server-side scalability mechanisms and control operations used to provide performance guarantees for Web applications. We organize the discussion along the following three questions:

1. Which metrics are used to assess Web application performance?

2. What are the available server-side mechanisms to enable a Web application to be scalable?

3. What kind of dynamic operations can be used to control a Web application to adapt to fluctuating traffic?

2.2 Metric determination

When assessing the performance of a Web application, one of the most widely used metrics is the response time. As shown in Figure 2.1, this response time includes several components. Different participants within the delivery of Web services are interested in different components (or groups of components). For instance, end users always care about the end-to-end response time of Web applications while the hosting providers pay more attention to the service-side response time. Another important metric for hosting providers is the cost they incur in hosting a Web application.

2.2.1 Performance metrics

Web application providers consider the end-to-end response time as an important metric to evaluate end-user experience [Rajamony and Elnozahy, 2001; Olshefski et al., 2004]. However, it is nontrivial to measure the client-perceived response time at the server. [Olshefski et al., 2004] proposed ksniffer, an online server-side traffic monitor that can accurately determine the client perceived pageview response time. ksniffer learns client pageview activity by analyzing network packets and reconstructing TCP connections. Consequently, ksniffer can capture network characteristics such as packet loss and delay. In case of performance degradation, Web application providers can use ksniffer to identify the source of the performance problem and react correspondingly.

As Web application providers usually have control only on the server side, the server-side response time is also an important metric for application providers. The average response time at the server-side is widely used in the community to evaluate the server-side performance of Web applications [Urgaonkar et al., 2005a]. In addition, some providers also pay attention to specific percentiles of response times [Sivasubramanian, 2007; DeCandia et al., 2007]. For instance, Andreolini et al. [2004] showed that detailed statistics can be more representative than average values to understand the performance variance of Web applications. In this thesis, we choose the average response time at the server-side as performance metric.

2.2.2 Cost metrics

Providing performance guarantees for Web applications involves costs. On the one hand, when applying admission control to guarantee Web application performance, the rejected user requests can be considered as a cost metric [Elnikety et al., 2004; Kamra et al., 2004]. Given the potential monetization benefits caused by serving user requests, Web application providers can derive the monetary cost of rejecting user requests.

On the other hand, when applying resource provisioning to guarantee Web application performance, the consumed resources can be considered as another cost. Resource cost can be measured in two categories: first, one can use the number of provisioned machines (either physical machines or virtual machines) [Ranjan et al., 2002; Urgaonkar et al., 2005a; Bennani and Menascé, 2005]; second, one can use the consumed fine-grained resource capacity, such as the CPU time, disk size and network bandwidth. For instance, Doyle et al. [2003] proposed to provision Web servers at the resource granularity of CPU, memory, and disk bandwidth. Chen et al. [2007] decomposed high-level performance requirements into low-level capacity requirements for various resource components, such as CPU and memory. Similarly, given the monetization price of provisioned resources, one can derive the economic cost for guaranteeing Web application performance.

In this thesis, we take resource provisioning into account to guarantee Web application performance. We therefore use the number of consumed machines (either physical machines or virtual machines) as the resource cost.

2.3 Scalability mechanisms

Scalability mechanisms are the foundation for providing performance guarantees of Web applications.

Web applications logically include two aspects: application processing, and data management. Therefore, developers usually organize a Web application in

two tiers, with a business-logic tier that runs application code, and a data tier that stores and manages related data. Both tiers must be scalable as they can both introduce performance bottlenecks. This section presents research works contributing to scalability mechanisms of the business-logic tier and the data tier.

2.3.1 Business-logic tier

Cluster-based server architectures have been shown to be an effective way to support the ever-growing traffic by distributing workloads across multiple server nodes [Armando et al., 1997; Cardellini et al., 2002]. Within a cluster, each node may support (part of) an entire service. For instance, content partitioning among server nodes allows the use of specialized servers to improve responses for different file types [Yang and Luo, 2000].

Replication is widely used to scale the computation part of a Web application by deploying multiple Web object replicas or application instance replicas either across a cluster or among several edge servers. For instance, Rabinovich and Aggarwal [1999] proposed the RaDaR (Replicator and Distributor and Redirector) architecture which is based on dynamic object replication and migration. RaDaR takes both server load and client-server proximity into account when deciding the number and placement of replicas. However, RaDaR mainly targets at scaling Web applications that host static and simple dynamic pages. Duvos and Bestavros [2000] presented an infrastructure to transparently replicate an application instance from one server to another. However, this infrastructure requires to first stop the original application instance before replicating it to a new server, which is not acceptable when continuous availability is required. Furthermore, simply copying the application code is not sufficient to enable a replicated application instance to work correctly on new servers. An application instance may rely on several custom libraries or third party modules. These dependencies should be also available on the new servers. To eliminate the online state and environment dependency problem, Awadallah and Rosenblum [2002] proposed a virtual machine monitor to encapsulate the whole state of the machine hosting an application instance into a virtual machine file. By replicating this virtual machine file into new real machines, one can launch multiple same application instances for scalability purpose. However, the virtual machine files can be pretty large, on the order of gigabytes. Instead of migrating the entire dynamic state of a running application from one server to another, Rabinovich et al. [2004] proposed an automatic deployment system to simplify the replication process. The authors implemented the deployment system by extending the concept of metafile used in software distribution systems. This system can place application instance replicas among edge servers according to the application load.

In addition, one can also increase the scalability of the business-logic tier by

improving the performance of individual application servers at the software level. Some efforts targeted at improving Web servers from the OS perspective, while others contributed to building more efficient servers.

One possible approach is to optimize the operating system for this type of network-intensive applications. For example, Pai et al. [2000] proposed IO-Lite a unified I/O buffering and caching system for general operating systems. In order to improve server application performance, IO-Lite eliminates all copying and multiple buffering of I/O data to avoid high CPU overload on copying data in the memory area. Nahum et al. [2002] analyzed how a general-purpose operating system and the network protocol stack could be improved to provide support for high-performance Web servers. de Bruijn [2010] proposed Streamline, which is a system I/O software layer allowing streaming I/O applications to achieve high throughput by reducing data movement and signaling between application, kernel and device.

Another approach is formed by alternative efficient Web server architectures. For instance, the Flash Web server ensures that its threads and processes are never blocked by using an asymmetric multiprocess event-driven architecture [Pai et al., 1999]. Similarly, Matt et al. [2001] proposed a new design framework for highly concurrent server applications, namely staged event-driven architecture (SEDA). In SEDA, applications are constructed as a network of stages, each representing a self-contained application component such as socket listen, socket read, and file access. Each stage stores incoming events into a queue and employs a controller to schedule and allocate threads to handle these events. The event-driven design based servers exhibit higher performance than those using a traditional thread-based design. We consider the staged event-driven design framework as a complementary work to further improve the scalability of a server cluster by improving the performance of individual nodes in the cluster.

2.3.2 Data tier

The data tier is at the heart of current data-intensive Web applications. Studies have shown that the data tier can often be the performance bottleneck of the whole application [Zhang et al., 2004]. In this section, we survey a number of research efforts working on mechanisms and techniques for the data tier scalability.

Similar to the usage of cluster architectures in the business-logic tier, clusters of commodity machines interconnected by a storage area network are well-suited to build a scalable data tier to support high-volume traffic. For instance, products such as Oracle Real Application Clusters [Oracle Cluster, 2010] and IBM DB2 Integrated Cluster Environment [Bialek and Tassi, 2006] use cluster computers to address the scalability of the data tier. The open-source community also developed

similar solutions such as MySQL cluster [MySQL Cluster, 2011] and Postgres-R [Bettina and Gustavo, 2000] to support data tier scalability.

When building a database cluster, one needs to replicate database contents across multiple servers. The read queries can be distributed among the replicas. However, all write queries, namely Update, Delete, and Insert (UDI queries), must first be executed at a "master" database, then propagated and re-executed at all other "slave" replicas. This master-slave replication schema provides strong data consistency in which read-only transactions will always see the latest snapshot of the database [Plattner and Alonso, 2004]. However, as all UDI queries must first execute in the master database, the scalability of a database cluster is limited by the throughput of the master database. Replication mechanisms such as Binary Replication in MySQL can greatly reduce the cost of re-executing UDI queries but they cannot reduce executing costs at the master node [MySQL Replication, 2011].

To provide strong data consistency while allowing good scaling behavior, Amza et al. [2003b] proposed a distributed versioning technique. This technique employs a scheduler which accepts transactions from clients and distributes them to a set of replicas. Clients send transactions to the scheduler with a declaration of the tables to read or write. The scheduler then assigns version numbers to these accessed tables in all replicas. The scheduler sends write queries in a transaction to all replicas while sending read queries to only one replica. All read-and-write operations on a particular table are executed in version number order to guarantee data consistency. Similarly, Cecchet et al. [2004] also used a scheduler named C-JDBC to support large database clusters. C-JDBC works without any modification to applications and database engines. However, the only premise is that both applications and databases should employ the same JDBC driver. Although these solutions offer better support for transactions than master-slave replication, these solutions send UDI queries to all replicas which strongly restrict their scalability.

Recent research efforts exploit the fact that database queries issued by Web applications belong to a relatively small number of query templates. This fact allows one to apply partial replication across all backend servers. Groothuyse et al. [2007] proposed the GlobeTP database replication system. GlobeTP allows one to carefully select table replica placements such that each query can be processed locally by at least one server. Meanwhile, GlobeTP provides major scalability gains by executing UDI queries only at a subset of all servers to reduce the write load. However, the efficiency of GlobeTP is constrained by the complex query templates that access multiple tables, such as join queries and database transactions. Therefore, data partitioning at finer granularity is necessary to reduce the constraint of table-based partial replication. We will return to this topic in Chapter 4 where we propose a finer-grained extension of this idea which achieves improved scalability.

Distributed relational database systems also partition tables either vertically or horizontally into smaller partitions according to workload to improve data access time [Navathe et al., 1995; Huang and Chen, 2001]. However, in these works, data partitions need to be reevaluated upon every workload changes [Kazerouni and Karlapalem, 1997]. Therefore, these works are not well-suited to the changing workloads of Web applications.

Finally, in recent years, the industry contributed a new family of scalable data stores for their demanding applications. The efforts mainly focus on scalability and high availability guarantees. For instance, Google's Bigtable provides a scalable data model to Web application developers for achieving improved scalability [Chang et al., 2006]. On the one hand, the data model is richer than simple key-value pairs with the support for sparse semi-structured data. On the other hand, the data model still remains simple for efficient usage. However, Bigtable does not support a full relational database model. In particular, Bigtable is targeted at storing data at Google products. Similarly, Amazon developed a highly available and scalable data store called Dynamo which met the performance requirements of a diverse set of applications [DeCandia et al., 2007]. Unlike relational databases, Dynamo exploits the fact that many services on Amazon's platform only need primary-key access to a data store. Therefore, Dynamo provides a simple key-based put/get interface to applications. Dynamo also shows the effectiveness of several techniques used to attain scalability and data consistency, such as data partition and replication, consistent hashing, and object versioning.

SimpleDB is another highly available, flexible, and scalable non-relational data store developed by Amazon [Amazon SimpleDB, 2011]. SimpleDB creates and manages multiple geographically distributed data replicas to enable high data availability. By using SimpleDB, Web application developers can focus on application development without considering the underlying infrastructure provisioning. In the open-source community, the Apache Cassandra project falls into the same category as a highly scalable distributed database [Cassandra, 2011]. In particular, Cassandra brings together Dynamo's fully distributed design and Bigtable's Column Family-based data model.

Yahoo PNUTS is also a massively parallel and geographically distributed database system but only for Yahoo!'s Web applications [Cooper et al., 2008]. In order to guarantee response time and data consistency of Web applications accessed by geographically dispersed users, PNUTS makes all high latency operations asynchronous while supporting record-level mastering to allow most requests to be satisfied locally. PNUTS entered the production version by serving data for some of Yahoo!'s social applications. It shows the effectiveness on providing rich database functionality and low latency at massive scale.

Note that these solutions are applicable to specific applications either sacri-

ficing certain features (i.e., consistency under failure scenarios) or requiring additional engineering efforts. In order to support the ACID property of relational databases, CloudTPS builds an ACID-compliant middleware layer on top of a distributed storage system such as Bigtable [Zhou et al., 2011]. CloudTPS assumes that transactions are short-lived and access only well-identified items. It provides an intermediate layer of servers called local transaction managers that handle transactions by running a global two-phase commit protocol. Data operations are subsequently made persistent in the underlying distributed storage system. As CloudTPS is intended to be a back end for a Web site, it has a strong focus on latency and partition tolerance.

We consider these solutions as fundamental mechanisms to build scalable data tiers for Web applications. However, these techniques only provide mechanisms to enable a Web application to be potentially scalable. In real world applications, workload is fluctuating and therefore dynamic control is required to adjust provisioning capacity to the incoming workload.

2.3.3 Load balancing

When applying replication to improve the scalability of either business-logic tier or data tier, one needs to select which replica should serve an incoming request or a query. Therefore, one must employ some switch that dispatches requests among back-end servers with the goal of evenly distributing the load among the server nodes. The switch can operate at the network level (a.k.a. layer-4 dispatching), or at the application level (a.k.a. layer-7 dispatching). Layer-4 switches select one of the servers in a cluster based on certain load sharing algorithms, such as weighted round-robin. However, layer-4 switches do not consider the requested contents when dispatching requests. Pai et al. [1998] showed that content-aware request distribution can offer significant performance improvement compared to schemas that purely take load into account.

In contrast, layer-7 switches examine the request and apply more sophisticated content-aware distribution schemas, which exploit information contained in the request, such as URL, cookies and requested content type to achieve cache affinity, client affinity and load sharing [Cardellini et al., 2002]. For instance, Pai et al. [1998] proposed a *locality-aware request distribution* (LARD) strategy to dispatch all requests for the same object to the same server node as long as its utilization was below a given threshold. By doing so, the requested object is more likely to be found in the back-end server's main memory cache with a improved cache hit rate. A similar implementation based on another content-aware request distribution also showed improved performance by focusing on cache locality [Xiaolan et al., 1999].

Zhang et al. [2005] studied the arrival rate and service characteristics of re-

quests in traces from the 1998 World Soccer Cup Web site. The results indicate that the real Web workload could have a huge variability in both request arrival rate and average request size over time. In addition, the workload presents a long-tailed feature. Therefore, they argue that request distribution policies have to allow rapid adaptation to the changing workload. They propose a new workload-aware request distribution policy called ADAPTLOAD. Apart from providing a locality-aware distribution policy, ADAPTLOAD employs knowledge of the historical request distribution to dispatch requests and readjust the load balancing parameters. A detailed performance analysis of ADAPTLOAD shows its effectiveness under changing workloads.

Most prior works employ a centralized request distribution strategy by using a single front-end switch. Although this strategy provides simplicity, it requires the front-end switch to act as an intermediary to forward client requests to back-end nodes and return server responses back to clients. Optimization mechanisms such as TCP splicing [Cohen et al., 1999] and TCP handoff [Pai et al., 1998] were introduced to eliminate the operation costs at the switch. However, these techniques do not completely alleviate the scalability limitation of a single front-end switch to only a small number of backend modes. Therefore, Aron et al. [2000] proposed to decouple the request distribution strategy from the single front-end switch. Instead, they let each back-end node in the cluster perform the distribution function and share the expensive operations of TCP connection establishment.

The above works focus on request distribution and load balancing at the entry point of a Web application, such as in front of the business-logic tier. Some ideas behind these techniques can also be applied to load balancing for database replication at the data tier. For instance, Amza et al. [2003a] proposed conflict-aware scheduling for distributing queries among database replicas. This scheduling technique requires that transactions specify the tables that they access at the beginning of the transaction, which can be considered as an extension of content-aware load balancing. This scheduling technique allows queries to be dispatched to replicas that are up-to-date. Similarly, Zuikevičiūtė and Pedone [2008] proposed conflict-aware load balancing technique to increase the concurrency and reduce the abort rate of middleware-based database replication so as to increase system performance.

Elnikety et al. [2007] proposed a memory-aware load balancing technique for scheduling transactions among replicas in a replicated database. This technique exploits knowledge of the working sets of transactions to assign them to replicas in such a way that these transactions execute in main memory. The authors showed that this technique can reduce disk I/O and thereby outperforms other techniques, such as round-robin, least connection, and locality-aware request distribution.

In a partially replicated database system, data is partitioned across replicas.

Therefore, queries can no longer be sent to arbitrary replicas. On the one hand, read queries can be dispatched only to a subset of database replicas instead of all replicas. On the other hand, UDI queries must be executed at all replicas that store the tables modified by the incoming UDI query. [Groothuyse et al., 2007] used two load balancing policies in their partial replication system GlobeTP. GlobeTP can schedule read queries in a round-robin fashion among the set of database replicas that can serve the incoming query. GlobeTP can also estimate the current load of each database server and dispatch queries to the least loaded database server (that also can serve the incoming query).

Load balancing is crucial to allow good utilization of resources, especially in heterogeneous environments. Georgiadis et al. [2004] focused on adaptive load balancing in heterogeneous systems where different nodes exhibit different job processing times. The authors proposed to equalize the expected response times on heterogeneous processing nodes by routing jobs according to processor capabilities. We return to this topic in Chapter 6, where we discuss adaptive load balancing in heterogeneous Cloud environments.

2.4 Dynamic control

Online Web applications receive fluctuating traffic so the performance can vary. In case of overload, there are three main approaches to prevent systems from being overwhelmed: (i) degrading the performance of admitted requests in order to serve a large number of aggregate requests; (ii) reducing the amount of work required by dropping a portion of requests and differentiating classes of customers to serve preferred clients; (iii) adding additional capacity to the application to support more requests.

The first approach targets at maximizing the system throughput at the cost of degrading performance. We therefore do not take this approach into account in this thesis, although some systems implemented this approach when the SLO permits temporary performance degradation. For instance, the Cataclysm hosting platform allows adaptive degradation of QoS to handle overload [Urgaonkar and Shenoy, 2005]. Similarly, Abdelzaher and Bhatti [1999] proposed to handle overloads by adapting the delivered content to load conditions, which can be considered as another kind of QoS degradation.

The second and third approaches dynamically adjust Web applications with the goal of guaranteeing Web application performance. Figure 2.2 shows the control loop to guarantee performance for Web applications, which involves several factors, such as application SLO, application workload, and server capability. The control loop monitors the quality of service of hosted applications and uses these measurements as the feedback. When the performance of a hosted application

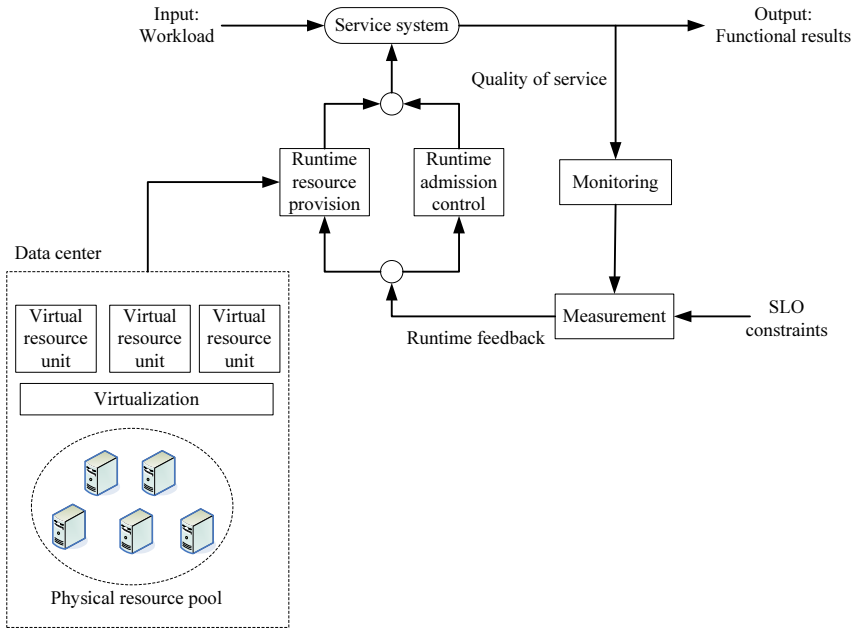


Figure 2.2: Control loop of performance guarantees for server-side Web application performance

cannot meet the SLO targets, dynamic control operations are triggered to adjust the performance of the hosted application. For instance, one can execute runtime resource provisioning to increase the resource capacity to the application. One can also apply runtime admission control to limit incoming workload to the application. This control loop continuously monitors the performance of the hosted application and triggers runtime actions to sustain its performance. Various techniques have been proposed in the context of this general control loop, such as workload characterization [Arlitt and Jin, 2000; Williams et al., 2005], admission control [Elnikety et al., 2004; Kamra et al., 2004], and dynamic resource provisioning [Urgaonkar et al., 2008, 2005a].

We organize the discussion along four directions: workload characterization, admission control and request scheduling, model-based dynamic resource provisioning, and virtualization environment.

2.4.1 Workload characterization

The performance of a Web application cannot be guaranteed without good knowledge of the input workload. Different types of workloads in Web applications can have various impacts on the performance of underlying systems. Workload char-

acterization typically relies on quantitative models [Calzarossa et al., 2000]. The objective is to capture and reproduce the behavior of the workload and its most important features. By understanding the workload characteristics, one can optimize several hosting techniques, such as caching and resource provisioning, to improve the performance of Web applications.

The first studies focused on workload of Web servers serving static documents. Arlitt and Jin [2000] present a detailed workload characterization of the Soccer World Cup'98 Web site. They highlighted several significant characteristics in the World Cup workloads and the corresponding performance implications. For instance, almost 19% of all responses contained the flag "Not Modified", which indicates that cache consistency traffic had a great impact on the World Cup workload. In addition, they also observed that the workload was bursty, although over longer time scale (hours or more) the arrival of these bursts was largely predictable. This indicates that the hosting provider could assign extra resource capacity to a Web site in advance, avoiding degraded performance of that Web site under high volume of traffic. Arlitt and Williamson [1996] conducted a comprehensive workload characterization study of Internet Web servers in 1995. Following this study, Williams et al. [2005] revisited this work by repeating many of the same analysis on new data sets collected in 2004. They analyzed the access logs from the same three academic Web sites and several different research and industrial Web sites with the focus on the document size distribution, document type distribution, and document referencing behavior. They observed similar Web server workload features in these two studies, although there had been many changes in Web technologies (i.e., new protocols, scripting languages, and caching techniques) during these 10 years. The most noticeable difference is a dramatic increase in Web traffic volumes. Other importance features, such as the heavy-tailed document size distribution and transfer size distribution, a small fraction of distinct server requests, and exponentially distributed and independent inter-reference times, were seen to remain largely unchanged. These two observations can be considered as fundamental characteristics of the way humans organize, store, and access information on the Web. Furthermore, these observations are helpful to improve the design of hosting systems such as a caching infrastructure for performance guarantees of Web sites.

More recently, similar research efforts have been done to study the workload characteristics of dynamic Web applications. Menascé and Akula [2003] provided a detailed workload characterization for an online auction application. Their study included: (i) a multi-scale analysis of auction traffic and bid activity within auctions; (ii) a closing time analysis in terms of number of bids and price variation within auctions; (iii) the characteristics of the auction winner in terms of entry time, entry price, and bidding activity. Their investigation results could be used to

devise business-oriented features, such as dynamic pricing and promotion models, and to design novel resource management policies for similar Web applications. Similarly, Shi et al. [2002] studied the characteristics of a medium-sized personalized Web site, *NYUHome*, which was a customized portal used by approximately 44,000 users from the New York University community. This study used detailed server-side traces of client activity over a two-week period. The authors showed that (i) a significant fraction of document bytes carried content that was "sharable"; (ii) clients perceived requests latencies over a wide range, determined primarily by their network connectivity. These observations could be used to derive general implications for efficient caching and edge generation of dynamic and personalized content.

Finally, Gill et al. [2007] characterized the traffic of the YouTube video sharing service over a period of three months. This study examined usage patterns, file properties, popularity and referencing characteristics, and transfer behaviors of YouTube. Similar to traditional Web applications, this study showed that caching could also improve the end user experience, reduce network bandwidth, and reduce the load on YouTube's core server infrastructure. However, due to the adoption of Web 2.0 technologies, additional metadata generated by Web 2.0 should be exploited to improve the effectiveness of strategies like caching.

2.4.2 Admission control and request scheduling

Web applications can encounter overload when the volume of requests exceeds the system's capacity for serving them. In such case, Web application providers often have the following goals: first, the system should remain operational even in the presence of extreme overload and even when the incoming request rate is several times greater than system's capacity; second, the system should maximize the number of requests completed during such an overload. Admission control can be an effective approach to achieve these two goals by dropping and scheduling requests based on certain policies. For instance, when dropping requests, a widely used principle is to provide preferential treatment to certain requests with the goal of achieving maximum revenue.

Web servers play an important role as the request entry point within a Web application. A number of groups worked on the admission control mechanisms in Web servers delivering static content only. These works enable a Web server to differentiate between requests from different classes and provide class-specific guarantees on performance. Bhatti and Friedrich [1999] proposed an architecture for Web servers to provide QoS to differentiated clients, incorporating request classification, admission control, and request scheduling. They implemented a prototype using Apache and showed how premium clients received preferential service over basic clients. However, they did not experimentally demonstrate sustained

throughput in the presence of overload when the premium requests themselves outstrip capacity. Li and Jamin [2000] proposed an algorithm for allocating differentiated bandwidth to clients in an admission-controlled Web server based on Apache. Similarly, Schroeder and Harchol-Balter [2006] showed how shortest-remaining-processing-time (SPRT) scheduling can be used to dramatically improve the response time of static content requests during transient overload conditions. Their study investigated various external factors, such as WAN delays and losses and different client behavior models. In particular, when applying SPRT scheduling, they showed that throughput was not sacrificed and requests for long files experienced only negligibly higher response times compared with those under standard FAIR (processor-sharing) scheduling. However, their work needs to modify the Web server machine at the operating system's kernel level.

The above works only address static content where one can assume that service costs are linear in proportion to the size of the response generated. However, real Web applications usually have multi-tier architectures that include dynamic contents and back-end databases. Service times for dynamic content have much greater variability with no direct relation to the size of the generated responses.

Some research works employ synthetic emulations to evaluate admission control for Web servers serving dynamic contents. For instance, Bhoj et al. [2000] designed and implemented a QoS-enabled Web server called Web2K. Web2K prioritizes requests into two classes: premium and basic. Connection requests are sorted into two different requests queues. Under overload conditions, admission control is based on two metrics: the accept queue length and measurement-based predictions of arrival and service rates of different classes of requests. The authors evaluated Web2K using Apache and emulated the use of dynamic content based on a simple model of execution costs. The evaluation showed how high priority requests maintain stable response times even in the presence of severe overload. Pradhan et al. [2002] developed an observation-based framework for "self-managing" Web servers that can adapt to changing workload while maintaining QoS requirements of different classes. Their evaluation was done primarily using static workloads, although they also examined a synthetic CGI script that blocks for a random amount of time.

Welsh and Culler [2003] proposed an adaptive approach to overload control in the context of the SEDA Web server. In SEDA Internet services are decomposed into multiple stages, each one of which can perform admission control. By monitoring the response time experienced on a stage, each stage can enforce a targeted 90th percentile response time. Admission is controlled by using an additive-increase multiplicative decrease control algorithm that adapts the rate at which tokens are generated for a token bucket traffic shaper. However, SEDA requires a complex administrator-driven discovery process to determine controller weights,

in which the control parameters are set by hand after running tests against the system.

Elnikety et al. [2004] proposed an admission control method in 3-tiered E-Commerce Web sites that requires no modifications to current Web servers, application servers and operating systems. This method externally measures execution costs of requests online, differentiating between different types of requests and applying overload protection. As the execution costs of requests can change as a function of system load, this method is more robust to overload by measuring execution costs online than those using measurements under light load. The authors implemented their method in a proxy, called Gatekeeper, which is transparent to the database and application server. Using Gatekeeper, they showed consistent performance during overload and dramatic improvements to response times by using shortest job first scheduling algorithm. However, Gatekeeper requires an initial configuration stage to determine the system capacity.

Some of these works rely on a human administrator to correctly configure various parameter values and set policies on a system-wide basis. Doing so not only requires detailed knowledge of the expected workload but also a good understanding of how various OS and Web server configuration parameters affect the overall performance. Thus, these works exacerbate the problems of configuration and tuning. Instead, Kamra et al. [2004] proposed Yaksha, a Proportional Integral controller for managing performance of 3-tiered e-Commerce Web sites. The main advantage of Yaksha is its self-tuning capabilities which are based on a processor-sharing model. Instead of requiring parameterization of controller weights, Yaksha only requires a desired response time as the input. Yaksha is noninvasive without extensive operating systems modifications or a complete rewrite of the server, which allows rapid deployment and use of pre-existing components. The authors conducted extensive experiments to show that Yaksha can effectively bound the response times of requests and still maintain high throughput levels during the overload period.

Although most research applies admission control on a per-request basis, Cherkasova and Phaal [2002] observed that users interact with Web sites in sessions consisting of multiple requests, and thus performing admission control on requests in isolation could cause sessions to be aborted unnecessarily. They showed that by considering session characteristics in admission control, rather than just individual requests, fewer session would be rejected.

Admission control can guarantee reasonable response time and throughput even during the overload period, but at the cost of rejecting users. We consider it is a complementary technique for Web application performance guarantees.

2.4.3 Dynamic resource provisioning

Resource provisioning is a commonly used technique to allocate resources to a Web application in such a way that the Web application can support incoming workload. It assumes that a sufficient quantity of resources are available upon demand to host Web applications according to their target performance. We generally divide resource provisioning into two categories: static resource provisioning and dynamic resource provisioning.

Static resource provisioning makes a one-time decision about the resource capacity required by a Web application. However, real Web workloads typically fluctuate over time. It is hard to make an accurate estimation of the resource demand for a Web application in advance. In order to support arbitrary incoming workload without violating SLOs, static resource provisioning typically relies on an assumption about the peak workload that the application is likely to observe. For instance, one can overprovision resources to a Web application with the goal of meeting the peak workload requirement. However, the daily peak-to-average workload ratio of a Web application can be as high as 3:1 or more [Shen et al., 2002]. Thus, overprovisioning resources for a Web application to accommodate the peak workload can result in important waste of resources.

As a consequence, dynamic resource provisioning is proposed to guarantee Web application performance under fluctuating workload [Chandra et al., 2003; Urgaonkar et al., 2008]. Dynamic resource provisioning continuously varies resource allocations to Web applications. For instance, dynamic resource provisioning adds extra resources to a Web application when its response time is about to violate its SLO.

There are usually two options. One approach is based on trial-and-error methodology. For instance, one can increase the capacity of all tiers that are at or near saturation by a constant amount (e.g. 10%). This approach is independent of any model and robust to any errors in the measurements used to parameterize the model. However, the time involved with the trial-and-error approach may be high for actually evaluating the effects of new allocations. This section focuses on the second approach which relies on the works of building performance models for driving the choices of resource allocation.

Analytical performance models are widely employed to capture relationships between application performance and the corresponding factors, such as application workload, application architecture and resource capability. Many models are based on queueing theory. Slothouber [1996] employed a network of four queues to model a Web server serving static content. Within this network, two queues model the Web server itself (one for all one-time initialization processing and one for data processing). The other two queues model the Internet communication network (one for data transmission and one for client-side data reception). Simi-

larly, Doyle et al. [2003] used queueing models to capture the relationship between the Web server performance and the finer granularity of resources, such as CPU, memory, and disk bandwidth. Different types of queueing models are used to model performance features of various provisioning cases, such as a generalized processor sharing queueing model for provisioning individual resources [Chandra et al., 2003], a G/G/1-based queueing model for provisioning replicated Web servers [Urgaonkar and Shenoy, 2005], a M/G/1/PS-based queueing model for provisioning application servers [Villela et al., 2007], and a G/G/N-based queueing model for provisioning application tier with N node cluster [Ranjan et al., 2002]. These works focus on resource provisioning for single-tier Web applications and provide theory foundations on how to apply queueing theory to complex Web applications.

In recent years, a number of efforts focused on analytical models of multi-tier Web applications. Kamra et al. [2004] modeled the entire multi-tier Web application as a single queue using a M/GI/1/PS queueing model. However, this coarse-grained model cannot identify the performance bottleneck among multiple tiers. Urgaonkar et al. [2005a] used a network of queues to represent different tiers within a multi-tier Web application. Their proposal simplified the complex work of modeling a multi-tier Web application in two parts: modeling the request processing at individual tiers, and modeling the request flow across tiers. This approach captures the caching effects by establishing caching tiers and using request transition probabilities between tiers to represent cache hits and misses. In addition, this work models session-based workloads and the concurrency limits at tiers. By applying this model, one can predict the average response time of requests accessing a multi-tier Web application. In such a case, one can further choose the number of servers to be provisioned to a Web application such that its average response times meet its SLO. Similarly, Sivasubramanian [2007] applied queueing theory to model the request processing and request flow within a multi-tier Web application. His model also incorporated the effects of master-slave database replication at the data tier. Provided with a break down of the response time of a whole Web application into per-tier response times, one can also employ a G/G/1 performance model to derive the number of servers required at different tiers to sustain the tier-specific response time [Urgaonkar et al., 2008]. MOKA is a middleware that does not only employ a queueing theory-based analytic model to predict performance of multi-tier applications, but also takes availability prediction and cost calculation into account [Arnaud and Bouchenak, 2010, 2011]. Instead of using manual offline calibration to determine the analytic model parameters, MOKA provides automatic and online model calibration. MOKA also applies admission control when provisioning applications to guarantee performance and availability constraints.

All analytic models require a controlled environment to accurately estimate model parameters, such as service times and visit ratios. These model parameters are usually extrapolated using measurements under very low system utilization levels [Urgaonkar et al., 2005a]. However, these parameters may change under varying system utilizations and may later fail to accurately predict real system behaviors [Zhang et al., 2007]. In contrast, Zhang et al. [2007] proposed to use a statistical regression method to approximate resource demands, such as CPU demand, under varying workload. This approach allows the use of few parameters to model the workload demands and therefore improve the applicability of analytic models to complex, live systems.

An alternative approach to estimate the model parameters is to combine an analytic model with machine learning methods. For instance, a reinforcement learning approach provides a knowledge-free trial-and-error methodology in which a leaner tries various actions in numerous system states, and learns from the consequences of each action. However, as the reinforcement learning approach explores expected suboptimal actions through randomized action selection, it may be costly to implement online. Therefore, Tesauro et al. [2006] propose a hybrid approach to combine the advantages of both model-based methods and tabula rasa reinforcement learning. In order to avoid the poor performance in large state spaces, the authors proposed offline training using that initial model-based policy. This hybrid approach can handle transients and switching delays when one dynamically allocates resources, which is out of the scope of purely analytic models. Similarly, Cohen et al. [2004] used a probabilistic model called Tree-Augmented Bayesian Networks for performance diagnosis and forecasting from system-level metrics in a three-tier Web service under a variety conditions. This statistical learning approach assumes little or no domain knowledge, in contrast to analytic model-based approaches which require expert knowledge. In addition, the statistical learning approach can adapt to system changes while the analytic model based approach needs modifications in case of unanticipated conditions.

Finally, some research efforts apply empirical approaches to derive the resource requirements of Web applications. The empirical approach usually involves two steps: i) monitoring an application's resource usage, and ii) using these statistics to derive resource requirements [Urgaonkar et al., 2002; Stewart and Shen, 2005]. However, the accuracy of this approach largely depends on the profiling results. In case of varying conditions, one needs to reprofile the resource usages.

The analytic models that are rooted in queueing theory can effectively capture steady-state resource requirements but therefore have limited workload regions. However, Web applications can incur bursty workload, such as short uneven spikes. The bursty workload is characterized by periods of continuous peak arrival rate that significantly deviate from the average traffic density. Similarly, a

system incurred bursty workload also exhibits short uneven peaks in resource utilizations, which indicates that the system faces congestion. As the queueing-based analytic models can be unacceptably inaccurate when processing bursty workload, Mi et al. [2008] proposed to use the index of dispersion to capture workload variability and burstiness. The index of dispersion is frequently used as a measure of burstiness in the analysis of time series and network traffic. It can be calculated using the counts of requests completed within the busy periods, which is practical to measure. The authors showed that by integrating workload burstiness into the performance model, the accuracy of model prediction can be increased by up to 30% compared to standard queueing models.

Although performance models of multi-tier Web applications have been well studied, in the real world major Web applications are often not designed as monolithic 3-tier applications but as a complex group of independent services (or components) querying each other [Vogels and Gray, 2006; Shoup, 2008]. Stewart and Shen [2005] proposed a performance model for such multi-component online applications. In order to derive the resource requirements of online services, the authors built offline application profiles characterizing per-component resource consumption that may significantly affect the online service throughput and response time. Particularly, they focused on profiling inter-component communication patterns that may affect bandwidth usage and network service delay between distributed components. Based on these offline profiles, they build models to predict system throughput and response time. However, when modeling the system response time, they simply summed delays at each component which is only applicable to a small range of online services. We will return to this topic in Chapter 5 where we discuss resource provisioning for multi-service Web applications.

Another important problem related to dynamic resource provisioning is to decide when to provision resources. The decision depends on the dynamics of Internet workloads. Internet workloads exhibit long-term variations such as time-of-day or seasonal effects as well as short-term fluctuations such as flash crowds. While long-term variations can be predicted ahead of time by observing the past, short-term fluctuations are much less predictable. Urgaonkar et al. [2008] employed a workload predictor to predict the peak demand over the next several hours or a day. The workload predictor can estimate the tail of the arrival rate distribution for the next few hours by using the probability distribution of historical session arrival rate seen during each hour of the day over the past several days. The peak workload for a particular hour is estimated as a high percentile of the arrival rate distribution of that hour. In addition to using the observations from prior days, the authors used the mean prediction error over the past several hours to improve the accuracy of workload prediction. When the prediction error remains positive over the past few hours, one can correct the underestimated peak workload.

Instead of using workload predictor, Zhang et al. [2010] addressed the issue of agile resource provisioning in a virtualized hosting environment by taking advantage of quick virtual machine reconfiguration. They proposed the use of ghost virtual machines which participate in the hosting cluster but are not activated until needed under an significant load increase. This approach exhibits much better performance in timely reassigning resources compared with legacy systems.

2.4.4 Hosting environment

In order to deliver Web services to end users, Web application providers can host their applications on either a dedicated hosting platform or a shared hosting platform. A dedicated hosting platform either uses the whole cluster to run a single application (such as a Web search engine) or use each individual hosting element to run a single application separately. The hosted Web application has full and exclusive access to the resources in the dedicated hosting platform. ICT companies that provide large scale Web applications, such as Google and Facebook, often use a dedicated hosting platform to host their applications. In contrast, a shared hosting platform uses the whole cluster to run a number of third-party Web applications. A Web application typically runs on a subset of the cluster nodes which may overlap with other applications. A shared hosting platform is attractive to Web application providers that want to rent resources to host their applications.

Recently, virtual machines are being employed in shared hosting environments to support flexible resource usage (i.e. Cloud Computing). Within a virtualized resource environment, a software layer, called virtual machine monitor (VMM), virtualizes the resources of a physical server and supports the execution of multiple virtual machines [Smith and Nair, 2005]. A VMM enables server resources, such as CPU, memory, disk and network bandwidth, to be partitioned [Barham et al., 2003]. VMMs have been deployed in shared hosting environments to run multiple applications and their VMs on a single server. Although virtual machines provide flexible resource usage, the virtualized resources in the hosting environments are not homogeneous. On the one hand, there are many different virtual instance types. On the other hand, even a single instance type is seen to exhibit heterogeneous performance. For instance, some virtual instances have faster CPU while others have faster I/O on Cloud platforms such as Amazon EC2 and Rackspace [Schad et al., 2010].

Most of current resource provisioning techniques assume that the underlying resources are homogeneous [Urgaonkar et al., 2008, 2005a]. This is a reasonable assumption in medium-scale environments such as cluster computers. However, in hosting environments where resources are heterogeneous such as Clouds, these techniques do not apply.

In recent years, a few research works addressed the problem of provision-

ing Web applications in heterogeneous resource environments. Christopher et al. [2008] predicted the performance of Internet Services across various server platforms with different hardware capacities such as processor speeds and processor cache sizes. Similarly, Marin and Mellor-Crummey [2004] used detailed hardware models to predict the performance of scientific applications across heterogeneous architectures. These approaches rely on detailed hardware metrics to parameterize the performance model. However, in the Cloud such low-level metrics are hidden by the virtualization layer. In addition, Cloud hardware resources are typically shared by virtual instances, which makes it much harder for hardware-based performance models to capture the performance features of consolidated virtual instances. These works therefore cannot be easily extended to predict Web application performance in the Cloud.

JustRunIt is a sandbox environment for profiling new machines in heterogeneous environments using real workloads and real system states [Zheng et al., 2009]. When one needs to decide on the usage of new machines, this work clones an online system to new machines, and duplicate online workload to them. This approach can effectively capture performance characteristics of new virtual instances in the Cloud. However, it requires to clone online environment to new instances at each adaptation, which can be very time-consuming. We shall return to this topic in Chapter 6 where we show how to provision Web applications in heterogeneous Clouds without the cost of actually executing the application on new instances for performance prediction.

2.5 Conclusion

In this chapter, we have discussed the important aspects of performance guarantees for Web applications. We provided a component-view of the end-to-end response time of a Web application. We analyzed the impacts of each individual component on the Web application performance and classified these components into three main aspects: client-side latency, network-induced delays and server-side latency. This classification allows us to distinguish various research efforts in this work area. For each aspect, we have discussed its corresponding problems and reviewed representative research works. In particular, we focused on the research works that have contributed to guaranteeing Web application server-side performance.

From this chapter, we can see that guaranteeing Web application server-side performance requires supports of both scalable mechanisms and dynamic control operations. The mechanisms for scaling application business-logic part are well understood. However, some of the presented efforts for scaling application data tier have limitations due to the coarse-grained data replication, while other efforts target at specific types of Web applications.

Dynamic control operations adjust the hosting system to adapt Web applications to varying and unpredictable workload. We surveyed many significant research efforts that address problems in the dynamic control loop for guaranteeing Web application performance, such as workload characterization, admission control, and resource provisioning. However, the growing complexity of current Web applications and heterogeneous hosting resources impose new challenges to these operations, particularly resource provisioning.

These issues form the central research question addressed by this thesis. In subsequent chapters, we first present detailed analysis of challenges in guaranteeing Web application server-side performance. Subsequently, we propose different mechanisms and techniques that aim to solve these challenges.

Chapter 3

Challenges

Providing performance guarantees for Web applications may at first glance look like a relatively simple exercise. It consists in essence in hosting the application on an elastic platform, and in adding or removing resources following the demands dictated by the request workload. However, to make this possible, a number of difficult challenges must be addressed.

First, it is impossible to guarantee an application's performance under arbitrary workloads if the application itself employs a nonscalable architecture. The way in which a Web application is constructed dictates its potential performance bottlenecks. For instance, if one puts all application data together into a monolithic database, it is possible to encounter a database bottleneck. Monolithic database architectures incur scalability limitations regardless of the number of provisioned servers. One therefore first has to face an application scalability challenge.

Second, provisioning Web applications in a cost-effective manner requires accurate performance predictions. Performance predictions help dynamic resource provisioning in deciding the number of provisioned servers and the function they should have within the application. Performance prediction must capture the relationship between application performance and related factors, such as workload intensity and resource capacity. However, these relationships are often complex and difficult to capture accurately, especially for applications composed of multiple components.

Finally, cloud computing environments create new possibilities for scalable Web application hosting but they also introduce new challenges. Current resource provisioning techniques assume that the underlying resources are homogeneous. However, in cloud environments the performance of various virtual instances is highly heterogeneous. Even multiple individual instances belonging to the same type exhibit heterogeneous performance features. One must therefore take this resource heterogeneity into account when hosting applications in clouds.

This chapter illustrates these three challenges using concrete examples. We first focus on the scalability challenge of the application data tier by scaling the TPC-W benchmark application using a monolithic database. To demonstrate the performance modeling challenge, we dynamically provision a simple two-tier application and show the difficulty of finding the optimal provisioning decision path in the search space of possible provisioning decisions. We finally study the performance stability and homogeneity of the Amazon EC2 Cloud platform to demonstrate the performance heterogeneity of resources in the Amazon cloud.

3.1 Application scalability challenge

Web applications are traditionally built along a two-tier architecture that separates a Web application into a business-logic tier for processing application-level semantics and a data tier for storing data. Under this architecture, a classical scalability technique in the business-logic tier consists of distributing workload among a cluster of servers. Similarly, one can also control the capacity of the data tier by distributing workload across a number of database replicas.

However, the scalability of the data tier faces important challenges as current replication techniques do not provide sufficient scalability. Master-slave database replication creates multiple data replicas in such a way that read queries can be served in any replica. This mechanism can support high volumes of traffic by distributing read queries among all replicas. However, when serving UDI (Update, Delete, Insert) queries, the database has to guarantee data consistency across all replicas. It therefore maintains a master database which is responsible for first executing all UDI queries and then propagating data changes to all replicas. All UDI queries must be processed by the master, which imposes unbounded write and consistency maintenance workloads on the master database. The master database will then eventually become the scalability bottleneck. One therefore cannot guarantee application performance if the database write workload increases beyond the capacity of the master database.

We illustrate this challenge with the widely used TPC-W e-Commerce benchmark [TPC-W, 2011]. TPC-W is a transactional Web benchmark, which simulates the activities of an online bookstore. It consists of 14 Web interactions, involving browsing, searching, ordering, promotion and administration etc. Some only serve static Web pages, while others require dynamic HTML generations which query a backend database. The database includes 8 tables: CUSTOMER, ADDRESS, ORDER, ORDER LINE, CREDIT CARD TRANSACTION, ITEM, AUTHOR, AND COUNTRY. Most of these tables are modified over the course of an execution, while only the AUTHOR and COUNTRY tables are read-only. The browsing-related Web interactions, such as Best Seller and Product Detail, issue

all read-only queries. The ordering-related Web interactions, such as Buy Confirm and Shopping cart, issue most UDI queries.

TPC-W defines three standard types of workloads, namely Browsing mix, Shopping mix and Ordering mix, each with a specific ratio of browsing-related to ordering-related interactions. The percentages of ordering-related interactions within these three workloads are respectively 5%, 20%, and 50%. The workload is generated by starting a certain number of Emulated Browsers (EBs). Each emulated browser begins a session at the TPC-W bookstore home page and continues traversing other pages, following different links and entering information with varying probabilities. An emulated browser also incorporates a think time parameter to control the time it waits between receiving a response and issuing the next request. Think times are randomly distributed with exponential distribution and average value 7 seconds according to the TPC-W specification.

In order to demonstrate the scalability challenge of the data tier, we increase the number of machines used to host TPC-W and measure the maximum throughput under these different configurations. We will see that the system cannot scale beyond a certain point, regardless of the number of machines it uses.

3.1.1 Methodology

We consider the Browsing mix as representative of a read-dominant workload, while the Ordering mix constitutes a UDI-dominant workload. We measure the potential scalability of TPC-W under these two workloads.

We populate the database with 86,400 customer records and 10,000 items records. Other tables are scaled according to the benchmark requirements. As we focus here on the scalability challenge of the data tier, we overprovision the application servers to avoid performance bottlenecks in that particular tier. We initially host TPC-W using one database server and then increase the workload. The SLO specifies that the average response time for each type of Web interaction must remain below 500 ms. We measure the maximum throughput by counting the number of supported EBs when the application is about to violate the SLO. Once the SLO is violated, we dynamically add one extra database server. We repeat this process until we reach 10 servers.

All experiments run on the DAS-3, an 85-node Linux-based server cluster [DAS 3, 2011]. Each machine in the cluster has a dual-CPU / dual-core 2.4 GHz AMD Opteron DP 280, 4 GB of memory and a 250 GB IDE hard drive. Nodes are connected to each other with a gigabit LAN such that the network latency between the servers is negligible. We use Tomcat v6.0.20 in application servers, and MySQL v5.1.23 in database servers. We implemented the application server load balancer as a layer-4 switch. We also implemented the database load balancer to distribute read queries evenly among replicas and to forward write

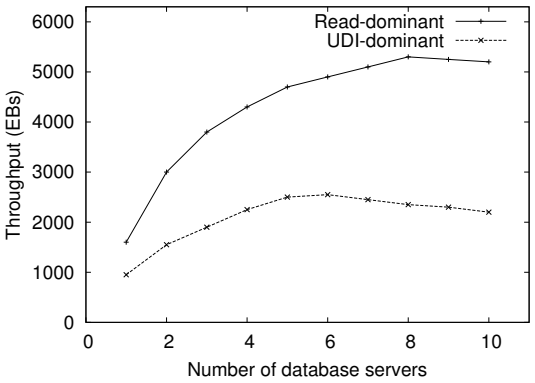


Figure 3.1: Scalability of TPC-W under Browsing and Ordering mixes

queries to the master database only.

3.1.2 Evaluation

Figure 3.1 shows the scalability of TPC-W under read-dominant workload (Browsing mix) and UDI-dominant workload (Ordering mix). Under the read-dominant workload, the throughput of TPC-W initially starts at 1600 EBs using one database server. It then increases when we add extra database servers. When the number of database servers reaches 8, the whole system can support a maximum throughput of 5300 EBs. However, the throughput then starts to *decrease* when we further add servers. Finally, the throughput stabilizes around 5200 EBs when we use 10 servers.

Similarly, under the UDI-dominant workload, the throughput of TPC-W initially starts at 950 EBs. This indicates that UDI queries are usually more resource intensive than read-only queries. The throughput also initially increases when we add extra database servers. However, it reaches the maximum value of 2550 EBs using 6 servers. After that, the throughput starts to decrease and finally reaches 2200 EBs when using 10 servers.

3.1.3 Discussion

Obviously, the scalability of TPC-W under both workload mixes has an upper bound. Typical multi-tier Web applications store their data in a single monolithic data tier. Consequently, it is necessary to replicate the entire data tier to manage its performance. In order to maintain the consistency of data replicas, the master database has to process all UDI queries and then propagate changes to all replicas. This data consistency requirement inevitably imposes additional management

costs on the master database. These administrative costs increase with the number of replicated servers. When the scale of replicas reaches a certain limit, the costs can even counterbalance the performance benefits from the replicas, which prevent applications from scaling further. If we keep on adding extra servers, the throughput decreases as the consistency management tasks consume most of the master database resources.

We also notice that the scalability of TPC-W under a read-dominant workload outperforms that under a UDI-dominant workload. This indicates that read-dominant workloads can result in less cost to maintain data consistency than the UDI-dominant workloads. As a result, a multi-tier Web application under a read-dominant workload can achieve better scalability than one under a UDI-dominant one. However, in reality, purely read-only workloads are not common in Web applications. Most Web applications issue significant numbers of UDI queries to their data stores.

This simple experiment demonstrates that adding extra servers to a multi-tier Web application does not necessarily improve its capacity. In other words, it is impossible to guarantee the performance of such an application using master-slave replication techniques if its workload increases beyond a few thousand simultaneous users. Supporting higher workloads requires different measures such as using faster hardware or restructuring the application. However, using a single fast machine requires one to provision the hardware capacity according to the peak application workload. Most Web application workloads fluctuate widely so this type of expensive high-performance hardware exhibits very low resource utilization at all times except at load peaks. In addition, even high-performance hardware cannot grow indefinitely. One still needs to face the scalability challenge when the workload exceeds the capacity of one high-performance server. We address this scalability challenge in Chapter 4 where we show how to restructure Web applications into a much more scalable multi-service architecture.

3.2 Performance modeling challenge

Provisioning a Web application may seem like a trivial task of simply adding machines to the application when it is about to violate the SLO. For instance, in Amazon EC2, one can employ the AutoScale service to dynamically provision the application by registering a provisioning trigger when the performance approaches the SLO [Amazon AutoScale, 2011]. However, in practice, it is hard to choose the optimal number of provisioned machines. On the one hand, the number of provisioned machines should be sufficient to support the workload while guaranteeing performance. On the other hand, using too many machines increases the hosting costs. Another difficulty is to deprovision a Web application when the current sys-

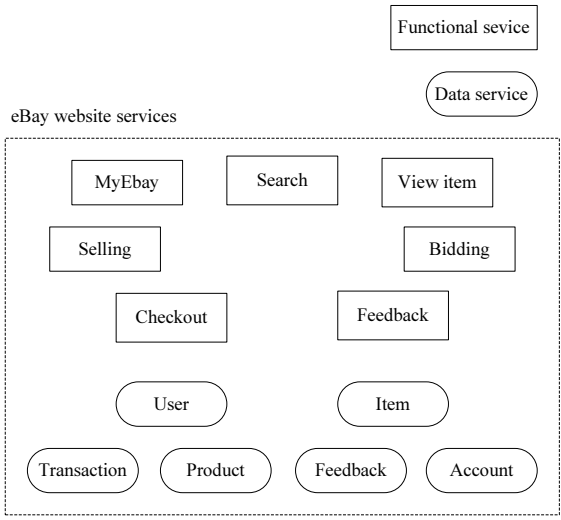


Figure 3.2: A set of services of eBay website

tem capacity is underutilized. Removing underutilized machines can save costs. However, this may also result in an SLO violation if we remove too many machines. One therefore needs to *predict* what the new performance would be if we added or removed one machine, before actually doing so.

An additional complication comes from the fact that Web applications are usually constructed in more than one tier. Real-world Web applications often involve large numbers of inter-dependent services. For instance, Figure 3.2 shows a set of services used to construct the eBay website [Shoup, 2008]. Although we do not know the exact invocation relationship among these services, it is reasonable to presume the invocations form a directed acyclic graph. (De-)provisioning such applications additionally requires one to answer the question: *in which service should one add or remove machines within the application?*. For instance, when provisioning a two-tier Web application, one needs to make a decision between two options at each adaptation point: provisioning the business-logic tier or provisioning the data tier. Different decisions can result in various performance effects. The choice of which tier to provision is not as simple as randomly selecting the provisioning tier, which can result in little performance improvement if chosen wrongly. One therefore needs to predict the different performance effects if we provisioned one machine to different tiers.

In order to predict the new performance obtained from various provisioning decisions, one must model the relationship between the application performance and several factors such as the workload, the resource capacity and the invocation relationship. However, performance modeling for complex Web applications is

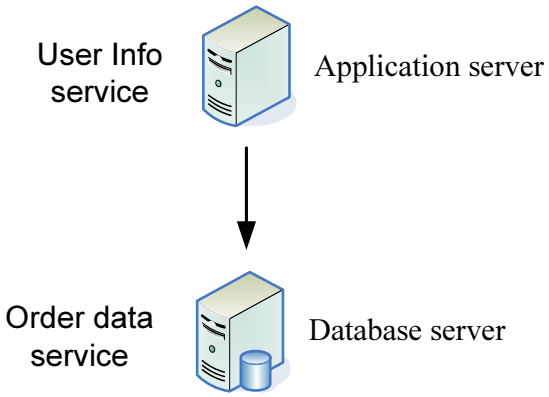


Figure 3.3: A two-tier Web application

challenging.

This section demonstrates these challenges. For simplicity, we use a two-tier Web application as shown in Figure 3.3. The application server tier issues one ordering transaction to the backend database for each HTTP request it receives. The ordering transaction purchases a small number of books according to the user's input. The application server tier then applies CPU-intensive XSLT transformation to transform XML templates into an HTML page which includes the concerned user information. The database includes CUSTOMER, ITEM, ORDER, ORDERLINE tables that are used in the ordering transaction. The database executes an ordering transaction by first inserting a purchasing order into the ORDER and ORDERLINE tables, and then updating the stocks of corresponding books. We dynamically provision this application under increasing workload. During the provisioning process, we explore all possible provisioning decisions at each adaptation step. This allows us to generate the entire search space for all provisioning decisions. Within this space, we show the resulting throughputs along various provisioning decision paths, and the difficulty to find the optimal one.

3.2.1 Methodology

In these experiments we initially populate the database with 500,000 product items. Although the whole database can be loaded in main memory, the query issued by the application is complex and imposes a CPU-intensive workload on the database server. At the beginning we use one application server and one database server to host this application. The initial workload intensity is set to 5 req/s. We then increase the workload intensity by steps of 2 req/s. During this process, we measure the response time for each workload intensity until the response time vi-

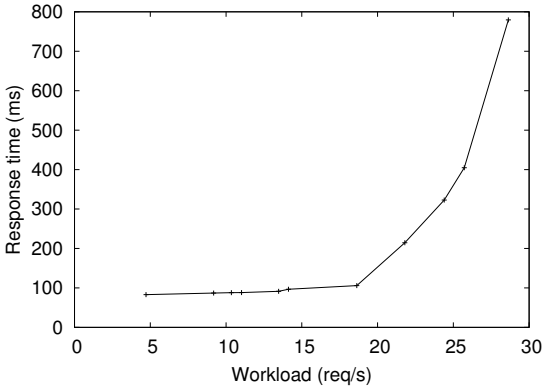


Figure 3.4: Performance behavior of the test Web application

olates the SLO which is set to 500 ms. We finally study the relationship between the response time and the workload intensity.

Once this application is about to violate the SLO, we face two different provisioning options: adding an application server or adding a database server. In order to show the difficulty of making provisioning decisions, we explore the search space covering all possible provisioning decisions from one application server and one database server, up to five machines of each type. When using a certain number of machines, we generate all possible server assignments. For each assignment, we execute it as one provisioning decision and then increase the workload until the application is about to violate the SLO. At that time we measure the achieved throughput in terms of request rate for that particular decision. Consequently, we can construct the entire decision space including the achieved throughputs and the server assignments of all provisioning decisions.

When dynamically provisioning this application, one must make a provisioning decision on the assignment of the new machine at each adaptation[1]. After multiple rounds of adaptations, these decisions form a decision path. As one faces two options at each adaptation, the number of possible decision paths increases with the rounds of adaptations. Therefore, we finally compare the achieved throughputs of various decision paths to find the optimal one. As previously we run these experiments on the DAS-3 cluster.

[1]Note that here we do not discuss the provisioning decision that shifts provisioned machines from one tier to another. Such decisions are necessary only when the workload mix changes.

3.2.2 Evaluation

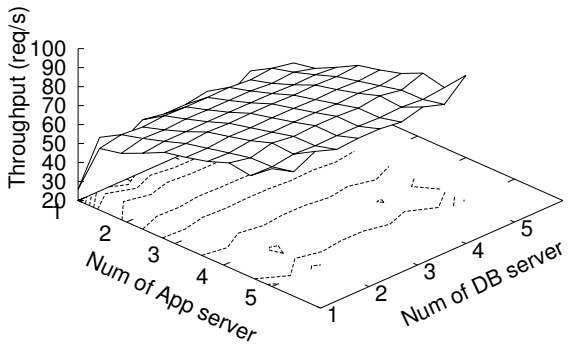
We first examine the performance behavior of the test application using one application server and one database server under an increasing workload. As shown in Figure 3.4, the performance curve is clearly nonlinear. As long as the workload intensity is lower than 19 req/s, the response time of the application remains nearly unchanged. When the workload increases roughly between 19 req/s and 26 req/s, the response time increases linearly with the workload. Finally, when the workload intensity increases beyond 26 req/s, the response time starts increasing more or less exponentially. In our research, we observed similar nonlinear behavior for individual tiers as well.

We further examine the challenge of making resource provisioning decisions for this application. Figure 3.5(a) shows the entire search space for resource provisioning decisions between one and five machines of each type. The X axis represents the number of provisioned application servers. The Y axis represents the number of provisioned database servers. The Z axis represents the achieved throughput under a given configuration. Each point in the space represents one possible provisioning decision (including the number of provisioned servers and the server assignment) and its corresponding throughput. When dynamically provisioning this application, one needs to find the optimal decision path along which the application can achieve the maximum throughput at each adaptation. Figure 3.5(b) shows the projection view of the optimal path and one possible sub-optimal path on the X-Y coordinate surface when using up to seven machines[2]. Each point is labeled with its corresponding throughput.

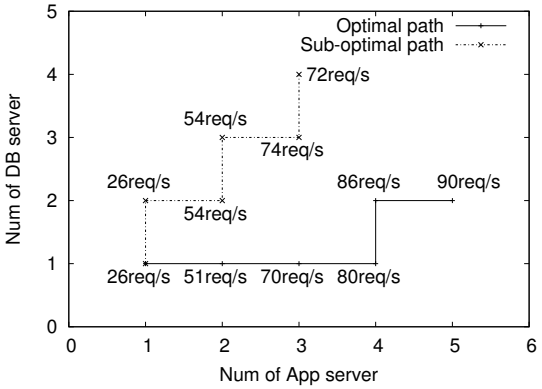
The initial configuration consists of one application server and one database server. When adopting the optimal path, one needs to provision this application successively adding three application servers in the first three adaptation, then one database server and finally one application server again. This application can finally achieve the maximum throughput of 90 req/s with the assignment of five application servers and two database servers.

The sub-optimal path represents a naive strategy where one alternately provisions one database server and one application server across the adaptation process. In this case, the application finally achieves the maximum throughput of 72 req/s with the assignment of four database servers and three application servers. Furthermore, as shown in Figure 3.6 the application achieves sub-optimal throughput at each adaptation along the provisioning path. For instance, when using 4 servers, the application achieves a throughput of 70 req/s along the optimal path while it achieves 54 req/s along the sub-optimal path. Therefore, adding a machine to different tiers results in various performance effects to the whole application.

[2]We observe that the application throughput cannot be improved when using more than seven machines, which is due to the application scalability challenge discussed in section 3.1.



(a) Search space of resource provisioning decisions



(b) Optimal/Sub-optimal decision paths

Figure 3.5: Provision decision search space and the optimal/sub-optimal paths

Although we can find the optimal path for this simple application by fully exploring the entire search space, it is impossible to apply this approach to live complex Web applications that consist of multiple tiers. The provisioning options increase with the number of application tiers, which correspondingly results in a much larger decision space. Note that the size of the search space also increases with the number of server types. For instance, within this two-tier application, we have two options at each adaptation point: adding an application server and adding a database. Assuming that we can also add an application cache server and a database cache server, the decision search space correspondingly increases to a four-dimensional one. In such cases, it is much harder to find the optimal path by exploring the search space.

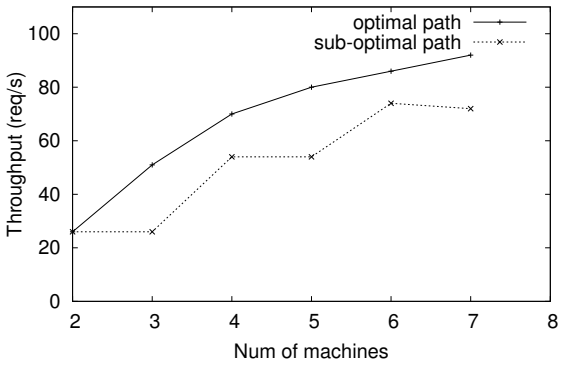


Figure 3.6: Nonlinear performance behavior of the test Web application

3.2.3 Discussion

The performance behavior of a Web application is non-linear. As shown in Figure 3.6, the throughput of the test application cannot increase linearly with the number of provisioned machines along either the optimal path or the sub-optimal path. When adopting the sub-optimal path, the throughput even sometimes remains unchanged when adding extra machines.

Another issue is the time needed to take a provisioning decision. Measuring the search space for Figure 3.5 required several hours of execution. It is clearly impossible to use the same technique in a production environment where decisions must be taken within a couple of seconds. Therefore, one needs good models capable of predicting the impact of resource provisioning decisions on performance at each step in a fast and accurate manner.

Chapter 2 reviewed a number of research efforts that focus on performance modeling of single-tier or multi-tier Web applications. In these works, queueing theory is used to model the non-linear performance behaviors of Web applications. In addition, the chain-like architecture of multi-tier Web applications facilitates the usage of queueing networks in modeling the interactions between any two tiers. One can find the optimal provisioning decision path by using these models. However, when a Web application is organized in a graph architecture such as in Figure 3.2, current performance models cannot apply to this architecture. More efforts are required to handle complex invocation patterns, which results in a complicated model. Furthermore, a complicated model is often difficult to parameterize and tune.

This section demonstrates that provisioning Web applications requires not only a detailed understanding of the performance behavior of individual tiers or components, but also accurate models for predicting the performance effect of individual

tier reconfigurations on the overall application performance. We will address this challenge in Chapter 5.

3.3 Resource heterogeneity challenge

Cloud computing is emerging today as a new paradigm for on-demand resource provisioning for Web applications. Cloud platforms typically do not give access to actual physical machines but rely on virtualization techniques for cost effectiveness and technical flexibility. Virtual machine monitors such as Xen allow fine-grained performance isolation between multiple virtual instances sharing the same physical resource [Barham et al., 2003]. The usual wisdom is that CPU performance can be isolated very effectively, while I/O performance is harder to isolate [Cherkasova and Gardner, 2005; Menon et al., 2005].

Dynamic resource provisioning traditionally relies on two fundamental performance properties of the available resource units:

- Performance stability: the performance of the provisioned resource units should remain constant over time. In clouds based on virtual machines, the performance of any virtual machine should be stable in time without being affected by the activity of other virtual machines on the same hardware.

- Performance homogeneity: the performance of different resource units should be predictable by profiling the current deployed resource units. This requires that the performance behavior of different resource units is homogeneous. In real-world applications, cloud providers commonly provide users with a set of different virtual machine types, each of which has different resource capacities in terms of CPU capacity, RAM size, disk I/O bandwidth and the like. The performance of different virtual machine types is obviously heterogeneous. However, the performance of multiple virtual machines of the same type should ideally be identical. Otherwise, it becomes very hard for one to quantify the number of virtual machines to be provisioned such that the performance of hosted applications meet its SLA targets.

Although these two properties are typically true for cluster-based systems where identical physical resources are exclusively dedicated to a single application, the introduction of virtualization in the cloud requires a re-examination of these properties. In this section, we study the above two performance properties of virtual instances provided by Amazon EC2 and demonstrate that EC2 exhibits relatively good stability but no performance homogeneity[3].

[3]Note that all performance observations in this section were realized in 2009. However, these

Table 3.1: Capacity details of virtual instances on EC2

Instance type	Compute units	RAM	I/O performance
Small	1	1.7GB	Moderate
Medium - high CPU	5	1.7GB	Moderate
Large	4	7.5GB	High
Extra large	8	15GB	High
Extra large - high CPU	20	7GB	High

3.3.1 Methodology

Amazon EC2 provides 5 types of virtual instances, each of which has different capacity in terms of CPU power, RAM size and I/O bandwidth. Table 3.1 shows the announced capacity details of virtual instances on EC2. To provide fault-tolerance, EC2 provides virtual instances across multiple data centers organized in so-called availability zones. Two virtual instances running in different availability zones are guaranteed to execute in different data centers. Of the six availability zones, four are located in the U.S.: US-EAST-1A, US-EAST-1B, US-EAST-1C, and US-EAST-1D. The other two are in Europe: EU-WEST-1A and EU-WEST-1B.

We examine the performance of Small instances on EC2. They are presumably the most widely used in real applications, being the default ones when creating a new virtual instance. To demonstrate that the same performance features appear on different types of virtual instances as well, we also partially benchmark medium instances with high CPU.

In practice, Web applications are commonly deployed in different data centers for fault tolerance and to deliver good quality of service to users in different locations. To match this common case we examine the performance of small instances in all six availability zones. This also allows us to make sure that the experimental instances do not interfere with each other.

In order to provision virtual machines for Web applications, it is important for one to predict its future performance if given one more or one less resource. This performance predictability in turn requires that performance of the same virtual machine remains constant over time. In addition, it requires that the performance of newly allocated virtual instances is similar to that of currently-deployed instances. We therefore carry out three groups of experiments to benchmark small instances on EC2.

Performance stability: The first group of experiments studies the performance stability of small instances under the constant workload intensity. As workloads of Web applications can be CPU-intensive and database I/O intensive, we develop the following three synthetic Web applications to simulate different types

features still hold true today.

of workload patterns:

- T1: a CPU-intensive Web application. This application consists of a servlet processing XML transformations based on client inputs. It issues no disk I/O (except for reading a configuration file when starting up) and very little network I/O (each request returns one HTML page of size around 1,600 bytes). The request inter-arrival times are derived from a Poisson distribution. The average workload intensity is 4 requests per second.

- T2: a database read-intensive Web application. This application consists of a servlet and a database hosted on two separate virtual instances. The database has 2 tables: CUSTOMER and ITEM. The CUSTOMER table holds 14,400,000 records while the ITEM table holds 50,000,000 records. The size of data set is 6.5 GB, which is nearly 4 times the RAM size of small instances. The servlet merely issues SQL queries to the backend database. It first gets customer order histories based on customer identification, and then fetches items related to those ones in the order history. Here as well, the request inter arrival times are derived from a Poisson distribution. The average workload intensity is 2 requests per second.

- T3: a database write-intensive Web application. This application consists of a servlet and a database hosted on two separate virtual instances. The servlet issues UDI (Update, Delete and Insert) queries to execute write operations on 2 tables: CUSTOMER and ITEM. The servlet first inserts 1,440,000 records into CUSTOMER table and then inserts 1,000,000 records into ITEM table. After populating the two tables, the servlet sends queries to sequentially update each record. Finally, the servlet sends queries to delete the two tables.

In this group of experiments, we randomly select one small instance in each availability zone and run T1, T2 and T3 on each instance separately. Each run of the tested application lasts for 24 hours in order to examine the potential interference of other virtual instances on the tested application performance. We compare the statistical values of the mean response times of each hour within the whole experiment period to evaluate the performance stability of small instances.

Performance homogeneity: The second group of experiments evaluates the performance homogeneity of different small instances. We randomly select one small instance in each availability zone and run T1 and T2 on each instance separately. Each run of the tested application lasts 6 hours such that database caches can fully warm up and the observed performance becomes stable. We repeat this process 5 times at a few hours interval such that we acquire different instances

Table 3.2: Software environment in all experiments

App server	DB server	JDK	OS	Kernel
Tomcat 6.0.20	MySQL 5.1.23	JDK 6 update 14	Ubuntu 8.10	Linux 2.6.21

Table 3.3: Response time of T1 on small instances

	US-EAST 1A	US-EAST 1B	US-EAST 1C	US-EAST 1D	EU-WEST 1A	EU-WEST 1B
Mean	684.8ms	575.5ms	178ms	185.2ms	522.9ms	509.9ms
Std	46.7ms	31.3ms	4.99ms	3.5ms	20.6ms	18.9ms
Std/Mean	6.8%	5.4%	2.8%	1.9%	3.9%	3.7%

in the same availability zone[4]. We compare the mean response times of tested application among all tested instances in order to evaluate the performance homogeneity of different instances.

CPU and I/O performance correlation: The third group of experiments studies the correlation between the CPU and I/O performance of small instances. The experiment process is similar to the second group. Instead of running T1 and T2 separately, we run them sequentially on the same instance, each one for 6 hours. We finally correlate the CPU performance and I/O performance in all tested instances to observe their relationships.

In all above experiments the clients run in a separate virtual instance in the same availability zone as the virtual instances hosting application servers and database servers. Table 3.2 shows the software environment used in all experiments.

3.3.2 Evaluation

This section first presents the results of CPU and disk I/O performance stability on small instances in each availability zone. We then show the performance behavior of different small and medium instances. Finally, we show the correlation between CPU performance and disk I/O performance of small instances. We measure the response time at the server side in all experiments in order to avoid the latency error caused by the network between servers.

Performance stability evaluation

To study the performance stability of small instances, we measure the response

[4]If one requested a virtual instance very quickly after another one is released, Amazon EC2 might recycle the virtual instances and return the previous one.

Table 3.4: Response time of T2 on small instances

	US-EAST 1A	US-EAST 1B	US-EAST 1C	US-EAST 1D	EU-WEST 1A	EU-WEST 1B
Mean	75.9ms	76.3ms	79.9ms	71.8ms	83.8ms	71.6ms
Std	1.28ms	2.05ms	6.36ms	1.14ms	2.1ms	2.34ms
Std/Mean	1.7%	2.7%	8.0%	1.6%	2.5%	3.3%

Table 3.5: Response time for INSERT operation of T3 on small instances

	US-EAST 1A	US-EAST 1B	US-EAST 1C	US-EAST 1D	EU-WEST 1A	EU-WEST 1B
Mean	0.33ms	0.33ms	0.33ms	0.53ms	0.47ms	0.46ms
Std	0.0012ms	0.0006ms	0.0022ms	0.0021ms	0.0019ms	0.0044ms
Std/Mean	0.4%	0.2%	0.7%	0.4%	0.4%	0.9%

time of each request and calculate the mean response time at a one hour granularity. We then compute the standard deviation of the 24 mean response times (each for 1 hour in the whole 24-hours experiment period). Table 3.3 shows the mean value and the standard deviation of the 24 mean response times for T1 in all zones.

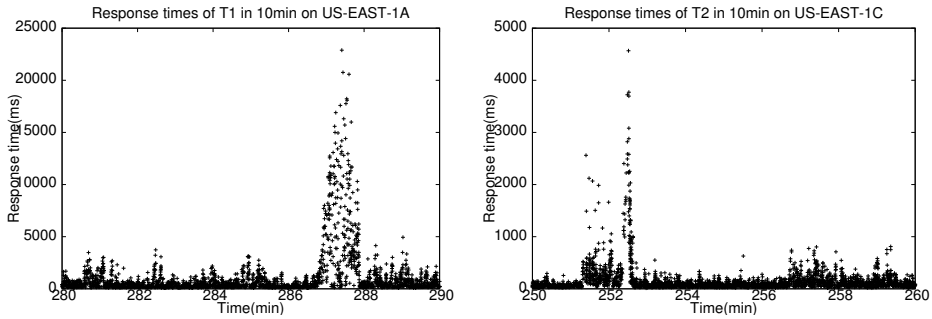
From the perspective of long-running time periods the CPU performance is quite stable. As shown in Table 3.3, the standard deviation of mean response times of each hour is between 2% and 7% of the mean value, which may be acceptable in real-world hosting environments.

However, we also observed that the CPU performance could be temporarily affected by the underlying resource sharing mechanism. Figure 3.7(a) shows the response time of T1 over a period of 10 minutes. We observe short periods during which the response time significantly increase. The duration of such peaks is relatively short (on average 1 to 2 minutes). We attribute these peaks to external factors such as the creation of a new virtual instance in the same physical machine. (the duration of such peaks is similar to the observed delay for creating a new virtual instance). Apart from these short peaks, the CPU-capacity sharing has little impact on the performance stability of CPU-intensive Web applications when considering performance behavior in long-running periods. We also observe that the performance of multiple small instances vary wildly from each other, from 185ms to 684ms of average response time.

Similarly, we measure the response time of application T2 and compute the mean response times of each hour. Table 3.4 shows the mean value and the standard deviation of mean response times of each hour within the 24-hours experiment period. The database read performance of small instances is also very sta-

Table 3.6: Response time for UPDATE operation of T3 on small instances

	US-EAST 1A	US-EAST 1B	US-EAST 1C	US-EAST 1D	EU-WEST 1A	EU-WEST 1B
Mean	3.84ms	3.5ms	2.67ms	3.09ms	3.72ms	3.91ms
Std	0.026ms	0.035ms	0.061ms	0.03ms	0.005ms	0.026ms
Std/Mean	0.7%	1%	2.3%	1.0%	0.1%	0.6%



(a) Response time samples of T1 (b) Response time samples of T2

Figure 3.7: Response time samples of T1 and T2 over a period of 10min

ble from the perspective of long-running time periods. For all tested instances, the mean response time of each hour deviates between 1.6% and 8% from the mean value. Similarly to CPU performance, the database read performance is affected by short interferences with the underlying I/O virtualization mechanism. Figure 3.7(b) shows the peak of response time of T2 over 10 minutes. The disturbances are also short, in the order of 2 minutes. The database read performance of different small instances (such as in different zones) are also different from each other.

We finally evaluate the performance stability of database write-intensive workloads. Tables 3.5 to 3.7 show response time statistics for database UDI operations. As shown in Tables 3.5 and 3.6, the performance of database INSERT and UPDATE operations is very stable. We observed that the standard deviation of those mean response times is small, between 0.2% and 2.3%.

However, Table 3.7 shows that the performance of database DELETE operation varies a lot. Similarly to CPU and database read operations, the performance of database UDI operations on different small instances are different from each other.

To demonstrate that other types of virtual instances on EC2 exhibit similar performance features, we partially benchmark medium instance with high CPU on

Table 3.7: Response time for DELETE operation of T3 on small instances

	US-EAST 1A	US-EAST 1B	US-EAST 1C	US-EAST 1D	EU-WEST 1A	EU-WEST 1B
Mean	30.1ms	21.7ms	30.3ms	17.8ms	15.9ms	21.7ms
Std	21.4ms	0.52ms	20.2ms	4.6ms	0.33ms	0.71ms
Std/Mean	71.1%	2.4%	66.7%	25.8%	2.1%	3.3%

Table 3.8: Mean response time of T1 on medium instances (high CPU)

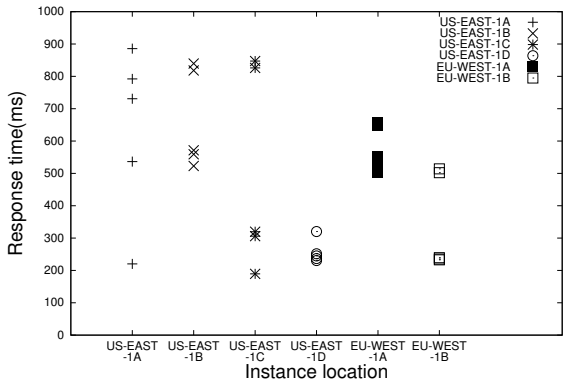
	US-EAST 1A	US-EAST 1B	US-EAST 1C	US-EAST 1D
Mean	307.7ms	791.5ms	197.3ms	199.8ms
Std	27.4ms	26.1ms	3.6ms	5.8ms
Std/Mean	9.6%	3.3%	1.8%	2.9%

EC2. We run T1 to profile CPU performance of medium instances after adjusting the request rate to match the capacity of medium instances. Table 3.8 shows the statistical values of mean response times of each hour on medium instances across the four US availability zones. Similar to the performance behavior of small instances, CPU performance is also relatively stable. The standard deviation of mean response time of each hour is between 2% and 10%. The CPU performance of different medium instances also varies a lot. One would however need more samples to fully explore the performance behavior of medium and large instances.

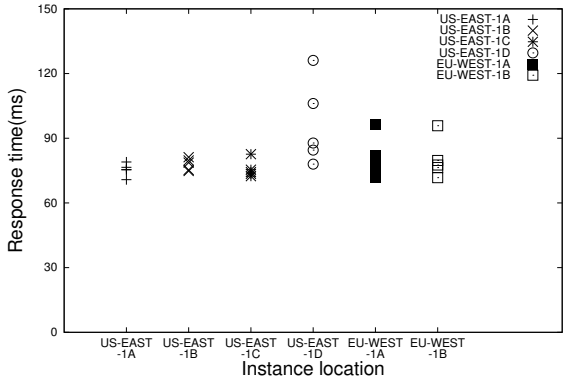
Performance homogeneity evaluation

So far we observed that the CPU performance and disk I/O performance of the same small instance are relatively stable from the perspective of long-running time periods (except for database DELETE operations). However, typical resource provisioning algorithms also expect that different small instances have homogeneous performance behavior such that the performance of future small instances is predictable based on current performance profiles. Thus, we evaluate the performance homogeneity of different small instances through the second group of experiments.

Figure 3.8(a) shows the mean response times of T1 for 30 different small instances across all zones (5 instances for each zone). Different instances clearly exhibit very different CPU performance when serving the exact same workload. Response times of different virtual instances vary up to a ratio 4. The same pattern appears both inside each zone and between different zones. Figure 3.8(b) shows similar heterogeneous performance behavior of different small instances for disk I/O operations, even though the variations are less important than for CPU. Thus, when provisioning small instances to host Web applications, it will be very hard for one to predict the performance of the newly-allocated virtual instances based



(a) CPU performance homogeneity



(b) Disk I/O performance homogeneity

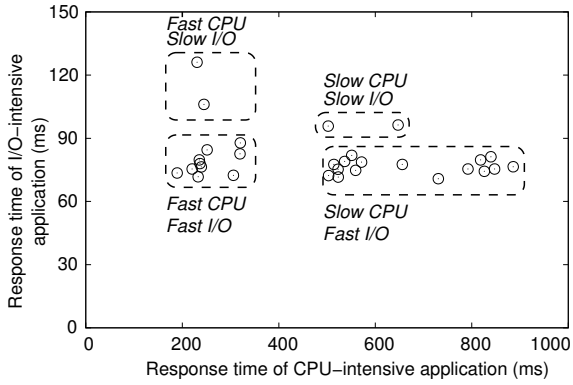
Figure 3.8: Performance homogeneity of different small instances across all zones

on the observed performance profiles of currently deployed ones. This property challenges traditional resource provisioning approaches which assume that the underlying infrastructure provides homogeneous resources.

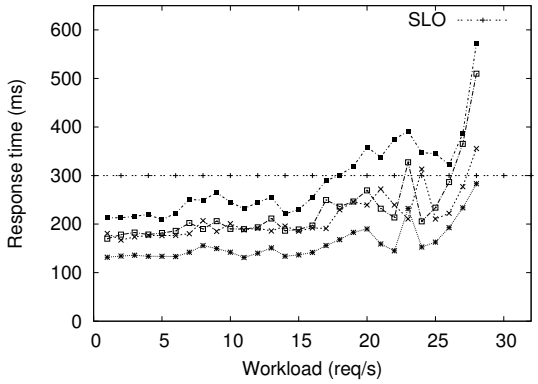
CPU and I/O performance correlation

As we observed that different small instances behave differently when serving CPU-intensive and I/O-intensive workloads, we further explore this phenomenon and run the third group of experiment to check if the CPU and disk I/O performances are correlated on supposedly identical small instances.

Figure 3.9(a) shows 30 samples of the correlation of CPU performance and disk I/O performance on identical small instances across all availability zones. Each point depicts the CPU and I/O performances of a single virtual instance.



(a) Cloud heterogeneity in CPU and I/O performance



(b) Web application performance heterogeneity on EC2

Figure 3.9: Cloud performance heterogeneity and its impact on Web applications

We do not observe any obvious correlation between the respective CPU and I/O performances of single instances. On the other hand, small instances can clearly be classified into three or four clusters with similar performance. Often (but not always) instances from the same availability zone are clustered together.

The performance heterogeneity of Cloud resources depicted in Figure 3.9(a) has important consequences on Web application hosting. Figure 3.9(b) shows the response time of a single-tier CPU-intensive Web application deployed on four 'identical' virtual instances in the Amazon EC2 Cloud, using Amazon's own Elastic Load Balancer (which addresses equal numbers of requests to all instances). As we can see, the four instances exhibit significantly different performance. The response time of the first instance exceeds 300 ms around 17 req/s while the fastest instances can sustain up to 28 req/s before violating the same SLO. As a result, it

becomes necessary to re-provision the application at 17 req/s, while the same virtual instances could sustain a much higher workload if they were load balanced according to their individual capabilities.

3.3.3 Discussion

These results suggest that individual small instances on Amazon EC2 behave consistently over time to provide stable performance for hosted Web applications. However, different small instances behave differently and therefore may be suitable to process different types of workload. Similar cloud performance results have also been observed in the context of scientific applications [Iosup et al., 2011a,b]. Within typical multi-tier Web applications, different tiers have different workload patterns. For example, an application server tier is commonly CPU-intensive while a database server tier is rather I/O intensive. Thus, one may consider employing well-suited virtual machine instances to provision resources to hosted applications such that each instance runs a task that matches its own performance profile. A virtual instance with fast CPU could be given an application server to run, while an instance with fast I/O would run a database server and a virtual instance with slow CPU and I/O may carry a modest task such as load balancing.

Provisioning heterogeneous instances in a profile-matching way can result in efficient resource usage. However, one must face the challenge of predicting the performance of a Web application if it was given a new instance of unknown performance. One cannot simply employ the performance features of current instances to make a prediction as the new one may behave differently. Therefore, one needs to profile each new virtual instance before deciding on its usage. It becomes even harder to decide where to assign the new virtual instance within a complex Web application such that it brings maximum performance benefit to the application as a whole. For instance, when provisioning a multi-tier Web application, one needs to profile the new instance with each tier in order to find the tier that receives maximum performance gain. However, such profiling cost is typically unacceptable in practice. We will address this challenge in Chapter 6.

Chapter 4

Making Web applications scalable

In order to retain a large number of customers, a Web application must guarantee a reasonable access performance regardless of the request load. Web application hosting systems therefore need the ability to scale their capacity according to business needs. Scaling application-specific computations is relatively easy as requests can be distributed across any number of independent application servers running identical code. The main challenge here, as explained in Section 3.1, is to scale access to application data.

We already showed the limitation of master-slave database replication for scaling Web applications in Chapter 3. To address this challenge, a number of techniques that exploit knowledge of the application data access behavior have been proposed to address the scalability issue of the data tier. For instance, database query caching improves the scalability of data tier by answering queries in caches and reducing the workload on back-end databases [Amiri et al., 2003; Bornhvd et al., 2004; Sivasubramanian et al., 2006]. Partial replication techniques use prior knowledge of data overlap between different query templates to reduce the data replication degree and limit the cost of database updates [Sivasubramanian et al., 2005; Groothuyse et al., 2007]. A query template is a parameterized SQL query whose parameter values are passed to the system at runtime. However, the effectiveness of database query caching relies on high temporal locality. Partial replication works best under simple workloads composed only of a few different query templates. When the number of query templates grows, an increasing number of constraints also reduce the efficiency of these two techniques: database caching mechanisms need to invalidate more cached queries upon each update to maintain consistency, and partial replication is increasingly limited in the possible choices of functionally correct data placements.

We claim that scalable Web applications should not be built along the traditional monolithic three-tier architecture. Instead, restructuring the application data is an effective measure to address the issue of scalable data access. In this chapter, we propose to restructure the application data into independent data services, each of which having exclusive access to its private data store. This allows one to reduce the workload complexity of each of the services. While this restructuring by itself does not lead to any performance improvements, it does allow for a more effective application of the aforementioned optimization techniques, thus leading to significantly better scalability. Importantly, this does not imply any loss in terms of transactional or strong consistency properties.

Restructuring a monolithic Web application composed of Web pages that address queries to a single database into a group of independent Web services querying each other requires one to rethink the data structures for improved performance – a process sometimes named *denormalization*. In previous research efforts, data denormalization has been largely applied to improve the performance of individual databases [Sanders and Shin, 2001; Shin and Sanders, 2006]. Data denormalization often creates data redundancy by adding extra fields to existing tables so that expensive join queries can be rewritten into simpler queries. This approach implicitly assumes the existence of a single database, whose performance must be optimized. In contrast, in this chapter we apply similar denormalization techniques in order to scale the application throughput in a multi-server system. Denormalization in our case allows one to distribute UDI queries among different data services, and therefore to reduce the negative effects of UDIs on the performance of replicated databases.

Note that similar techniques of data fragmentation have been studied in the design of distributed relational database systems [Navathe and Ra, 1989; Navathe et al., 1995; Ozsu and Valduriez, 1999; Huang and Chen, 2001]. In these works, tables are partitioned either vertically or horizontally into smaller fragments. Partitioning schemes are determined according to a workload analysis in order to optimize access time. However, these techniques do not fundamentally change the structure of the data, which limits their efficiency. Furthermore, changes in the workload require to constantly re-evaluate the data fragmentation scheme [Kazerouni and Karlapalem, 1997]. We consider that dynamic environments such as Web applications would make such approaches impractical. In contrast, this chapter proposes a one-time modification in the application data structure. Further workload fluctuations can be handled by scaling each service independently according to its own load.

In the rest of this chapter, we show how one can denormalize the data into data services for existing monolithic applications. To demonstrate the effectiveness of our proposal, we study three Web application benchmarks: TPC-W [TPC-W,

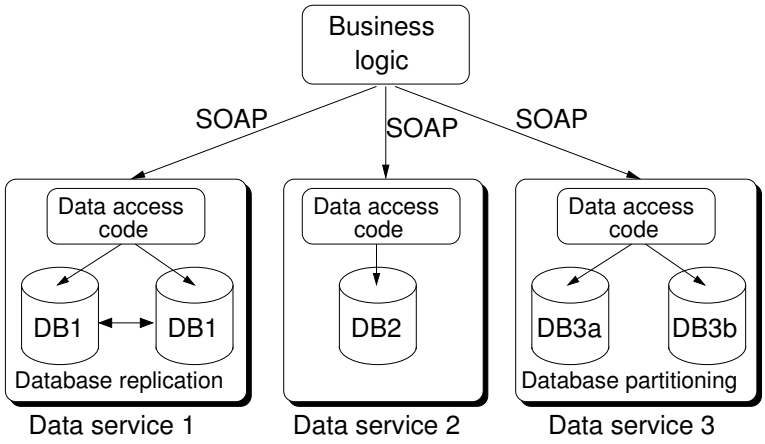


Figure 4.1: System model

2011], RUBiS [Amza et al., 2002], and RUBBoS [RuBBoS, 2011]. We show how these applications can be restructured into multiple independent data services, each with a very simple data access pattern. We then focus on the UDI-intensive data services from TPC-W and RUBiS to show how one can host them in a scalable fashion. Finally, we evaluate the scalability of TPC-W, the most challenging of the three benchmarks, and demonstrate that the maximum sustainable throughput grows linearly with the quantity of hosting resources used.

4.1 System model

4.1.1 Goal

The idea behind our work is that the data access pattern of traditional monolithic Web applications is often too complex to be efficiently handled by a single scalability technique. Indeed, proposed techniques work best under specific simple access patterns. Data replication performs best with workloads containing few or no UDI; query caching requires high temporal locality and not too many UDIs; partial replication or even data partitioning demand that queries do not span multiple partitions.

We claim that major gains in scalability can be obtained by restructuring Web application data into a collection of independent data services, where each service has exclusive access to its private data store. While such restructuring does not provide any performance improvement by itself, it considerably simplifies the data access pattern generated by each service. This allows one to apply appropriate scaling techniques to each service.

Figure 4.1 shows the system model of a Web application after restructuring. Instead of being hosted in a single database, the application data are split into three separate databases DB1, DB2 and DB3. Each database is encapsulated into a data service which exports a service interface to the application business logic. Each data service and its database can then be hosted independently using the technique that suits it best according to its own data access pattern. Here, DB1 is replicated across two database servers, DB2 is hosted by only one server, while DB3 has been further partitioned into DB3a and DB3b. Note that splitting the application data into independent services also improves separation of concerns: details about the internal hosting architecture of a data service are irrelevant to the rest of the application.

4.1.2 Data denormalization constraints

Denormalizing an application's data into independent data services requires deep changes to the structuring of the data. For example, a table containing fields $\langle key, attr1, attr2 \rangle$ and queried by templates "`SELECT key FROM Table where attr1=?`" and "`SELECT key FROM Table where attr2=?`" may be split into two tables $\langle key, attr1 \rangle$ and $\langle key, attr2 \rangle$, which may belong to two different data services.

However, not all tables can be split arbitrarily. In practice, data accessed by different queries often overlap, which constrains the denormalization. We identify two types of constraints: transactions and query data overlap.

Although database transactions are known as an adversary to performance, they sometimes cannot be avoided. An example is a checkout operation in an e-commerce application where a product order and the corresponding payment should be executed atomically. ACID requirements provide a strong motivation for maintaining all data accessed by one transaction inside a single database, and therefore inside a single data service. Splitting such data into multiple services would impose executing distributed transactions across multiple services, for example, using protocols such as 2-phase commit. We expect that this would negate the performance gains of the data decomposition.

Another source of constraints is created by queries executed outside transactions. Similar to constraints created by transactions, it seems logical to cluster the data accessed by each query. However, in most cases the overlap of different queries would lead to creating a single data service. Instead, we can apply two other transformations. First, certain complex database queries can be rewritten into multiple, simpler queries. Doing this reduces the data inter-dependency and allows better data restructuring. Second, data dependencies induced by overlapping queries can also be reduced by replicating certain data to multiple services. However, this implies a trade-off between the gains of splitting the data into more

services and the costs of replicating update queries to these data over multiple services.

4.1.3 Scaling individual data services

In all our experiments, we noticed that the services resulting from data denormalization maintain extremely simple data structures and are queried by very few query templates. Such a simple workload considerably simplifies the task of hosting services in a scalable fashion. For example, some data services receive very few or even no UDI queries at all. Such services can therefore benefit from massive caching or replication. On the other hand, some other services concentrate large number of UDI queries, often grouped together inside transactions. Such services are clearly harder to scale. However, they at least benefit from the fact that they receive less queries than the database of a monolithic application would. Additionally, we show in Section 4.3.1 that such services can often be *partitioned* so that UDI queries are distributed across multiple database servers.

4.2 Data Denormalization

Service-oriented data denormalization exploits the fact that UDI queries and transactions often access only a part of the columns of a table. Decomposing such tables into multiple smaller ones helps distributing UDI queries and transactions to more data services, and thereby simplifies their workload. As discussed in Section 4.1, two main constraints must be taken into account when denormalizing an application's data. First, one should split the data into the largest possible number of services, such that no transaction or UDI query in the workload spans multiple services. Second, one must make sure that read queries can continue to operate over the then partitioned data.

4.2.1 Denormalization and transactions

As discussed in previous sections, we need to cluster the data into services such that no transaction overlaps multiple data services. To this end, we first mark which data columns are accessed by each transaction. Then, simple clustering techniques can be applied to decompose the data into the largest possible number of independent data services.

We distinguish three types of "transactions" that must be taken into account here. First, real database transactions require ACID properties. This means that all the data they access must be accessed atomically and must be placed into the same service. One exception to this rule is formed by data columns that are never

updated, neither by the transaction in question nor by any other query in the workload. An example is the table that matches zipcodes to local names. Such read-only data does not need to be placed in the same data service, and can be abstracted as a separate data service.

The second type of transaction is a so-called "atomic set," where only the Atomicity property of a normal transaction is necessary. Atomic sets appear, for example, in TPC-W, where a query that reads the content of a shopping cart and the one that adds another element must be executed atomically. For such atomic sets, only the columns that are updated must be local to the same data service to be able to provide atomicity. Columns that are only read by the atomic set can reside outside the service, as they are not concerned by the atomicity property[1].

Finally, UDI queries that are not part of a transaction must be executed atomically, and therefore must be considered as an atomic set composed of a single query.

Once one has marked each transaction, UDI query and atomic set with the data columns that should be kept in a single service, simple clustering techniques can provide the first step of decomposition of the database columns into services. However, this step is not functional, as it accommodates only the needs of transactions and UDI queries. To become functional, one must further update this data model to take read queries into consideration.

4.2.2 Denormalization and read queries

Clearly, one can consider read queries similarly to UDI queries and transactions, and cluster data services further such that no read query overlaps multiple services. However, applying this method would increase the constraints to the data decomposition and lead to coarse-grain data services, possibly with a single data service for the whole application.

Instead, as shown in Figure 4.2, two different operations can be applied. First, certain read queries can be rewritten into a series of multiple sub queries, where each sub query can execute in one data service. For example, in TPC-W, the `CUSTOMER` and `ORDER` tables are located in different data services, whereas the following query spans both tables with a join operation: "`SELECT o_id FROM customer, orders WHERE customer.c_id = orders.o_c_id AND c_uname = ?`". However, this query can be easily rewritten into two sub-queries that access only one table: i) "`SELECT c_id FROM customer WHERE c_uname = ?`"; and ii) "`SELECT o_id FROM orders WHERE o_c_id=?`". The returned result

[1] In the case of actual database transactions, these data columns must reside inside the data service to be able to provide the Isolation part of ACID properties.

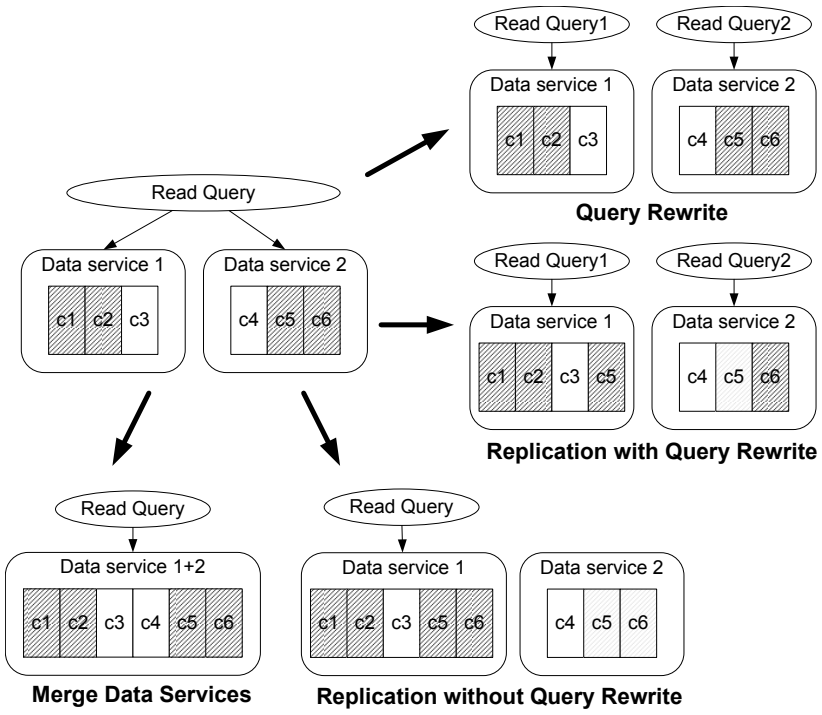


Figure 4.2: Different denormalization techniques for read queries

of the first query is used as input for the second one and the final result is returned by the second query.

Another transformation often applied in traditional database denormalization techniques consists of replicating data from certain database tables to other tables. This allows one to transform join queries into simpler queries. Note that traditional denormalization applies this technique to optimize the efficiency of query execution within a single database whereas we apply this technique to be able to split the data into independent data services. For example, the following query accesses two tables in two different data services: "`SELECT item.i_id,item.i_title FROM item,order_line WHERE item.i_id=order_line.ol_i_id AND item.i_subject=? LIMIT 50`". Replicating column `i_subject` from table `ITEM` to the other data service allows one to transform this query and to target a single data service. The only constraint is that any update to the `i_subject` column must be applied at both data services, preferably within a (distributed) transaction. This scheme is therefore applicable only in cases where the data to be replicated are rarely updated.

To conclude, complex query rewriting should be the preferred option if the

semantics of the query allows it. Otherwise, column replication may be applied if the replicated data are never or seldom updated. As a last resort, when neither query rewriting nor column replication is possible, merging the concerned data services is always correct, yet at the cost of coarse-grain data services.

4.2.3 Case studies

To illustrate the effectiveness of our data denormalization process, we applied it to three standard Web applications: TPC-W, RUBiS and RUBBoS.

4.2.3.1 TPC-W

TPC-W is an industry standard e-commerce benchmark that models an online bookstore similar to `Amazon.com`. Its database contains 10 tables that are queried by 6 transactions, 2 atomic sets, 6 UDI queries that are not part of a transaction, and 27 read-only queries.

First, the transactions and atomic sets of the TPC-W workload impose the creation of four sets of transactions whose targeted data do not overlap. The first set contains transaction `Purchase`, and the two atomic sets `Docart` and `Getcart`; the second set contains the `Adminconfirm` transaction, the third set contains only the `Updaterelated` transaction. Finally, the last set contains `Addnewcustomer`, `Refreshsession` and `Enteraddress`. This means for example that the original `ITEM` table from TPC-W must be split into five tables: `ITEM_STOCK` contains the primary key `i_id` and the column `i_stock`; table `ITEM_RELATED` contains `i_id` and `i_related1-5`; table `ITEM_DYNAMIC` contains `i_id`, `i_cost`, `i_thumbnail`, `i_image` and `i_pub_date`; the last table contains all the read-only columns of table `ITEM`.

The result of the first denormalization step is composed of five data services: a *Financial* data service contains tables `ORDERS`, `ORDER_ENTRY`, `CC_XACTS`, `SHOPPING_CART`, `SHOPPING-_CART_ENTRY` and `ITEM_STOCK`; data service *Item_related* takes care of items that are related to each other, with table `ITEM_RELATED`; data service *Item_dynamic* takes care of the fields of table `ITEM` that are likely to be updated by means of table `ITEM_DYNAMIC`; finally, data service *Customer* contains customer-related information with tables `CUSTOMER`, `ADDRESS` and `COUNTRY`. The remaining tables from TPC-W are effectively read-only and are clustered into a single data service. This read-only data service can remain untouched, but for the sake of the explanation we split it further during the second denormalization step.

The second step of denormalization takes the remaining read queries into account. We observe that most read queries can either be executed into a single data service, or be rewritten. One read query cannot be decomposed: it fetches the

Data service	Data Tables(included columns)	Requests
Financial	ORDERS ORDER_ENTRY CC_XACTS I_STOCK(i_stock) SHOPPING_CART SHOPPING_CART_ENTRY	getLastestOrderInfo, createEmptyCart, addItem, refreshCart, resetCartTime, getCartInfo, getBesterIDs, computeRelatedItems, purchase
Customer	CUSTOMER ADDRESS COUNTRY	getAddress, setAddress, getCustomerID, getCustomerName, getPassword, getCustomerInfo, login, addNewCustomer, refreshSession
Item_dynamic	ITEM_DYNAMIC(i_cost i_subject i_image i_pub_date i_thumbnail)	getItemDynamicInfo, getLatestItems, setItemDynamicInfo
Item_basic	ITEM_BASIC(i_title i_subject) Author	getItemBasicInfo, searchByAuthor, searchByTitle, searchBySubject
Item_related	ITEM_RELATED(i_related1-5)	getRelatedItems, setItemRelated
Item_publisher	ITEM_PUBLISHER(i_publisher)	getPublishers
Item_detail	ITEM_DETAIL(i_srp i_backing)	getItemDetails
Item_other	ITEM_OTHER(i_isbn i_page i_desc i_dimensions i_avail)	getItemOtherInfo

Table 4.1: Data services of the denormalized TPC-W

list of the best-selling 50 books that belong to a specified subject. However, the list of book subjects `i_subject` is read-only in TPC-W, so we replicate it to the *Financial* data service for this query[2]; `i_subject` is also replicated to the *Item_dynamic* data service for a query that obtains the list of latest 50 books of a specified subject.

The remaining read-only data columns can be further decomposed according to the query workload. For example, the "Search" web page only accesses data from columns `i_title`, `i_subject` and table `AUTHOR`. We can thus encapsulate them together as the *Item_basic* service. We similarly created three more read-only data services.

The final result is shown in Table 4.1. An important remark is that, although denormalization takes only data access patterns into account, each resulting data service has clear semantics and can be easily named. This result is in line with

[2]Note that we cannot simply move this column into the *Financial* service, as it is also accessed in combination with other read-only tables.

observations from Vogels and Gray [2006], where examples of real-world data services are discussed.

4.2.3.2 RUBBoS

RUBBoS is a bulletin-board benchmark modeled after `slashdot.org` [RuBBoS, 2011]. It consists of 8 tables requested by 9 UDI queries and 30 read-only queries. RUBBoS does not contain any transactions. Six tables incur UDI workload, while the other two are read-only. Furthermore, all UDI queries access only one table. It is therefore easy at the end of the first denormalization step to encapsulate each table incurring UDI queries into a separate data service.

All read queries can be executed in only one table except two queries which span two tables: one can be rewritten into two simpler queries; the other one requires to replicate selected items from `OLD_STORIES` into the `USERS` table. The `OLD_STORIES` table, however, is read-only so no extra cost is incurred from such replication. Finally, the two read-only tables are encapsulated as separate data services.

RUBBoS can therefore be considered as a very easy case for data denormalization.

4.2.3.3 RUBiS

RUBiS is an auction site benchmark modeled after `eBay.com`. It contains 7 tables requested by 5 update transactions. Except for the read-only tables `REGIONS` and `CATEGORIES`, the other five tables are all updated by INSERT queries, which means that they cannot be easily split. This means that the granularity at which we can operate is the table. The transactions impose the creation of two data services: the *Users* data service contains tables `USERS` and `COMMENTS`, while the *Auction* data service contains tables `BUY_NOW`, `BIDS` and `ITEMS`. The final result of data denormalization is shown in Table 4.2.

RUBiS is a difficult scenario for denormalization because none of its tables can be split following the rules described in Section 4.2.1. We note that in such worst-case scenario, denormalization is actually equivalent to the way GlobeTP [Groothuyse et al., 2007] would have hosted the application. We will show however in the next section that scaling the resulting data services is relatively easy.

4.3 Scaling Individual Data Services

In all cases we examined, the workload of each individual data service can be easily characterized. Some services incur either read-only or read-dominant work-

Data Service	Tables	Transactions
User	USERS[U] COMMENTS[C]	Storecomment(U,C) Registeruser(U)
Auction	ITEMS[I] BUY_NOW[N] BIDS[B]	Storebuynow(I,N) Registeritem(I) Storebid(I,B)
Categories	CATEGORIES	-
Regions	REGIONS	-

Table 4.2: Data services of RUBiS

load. These services can be scaled up by classical database replication or caching techniques [Sivasubramanian et al., 2007]. Other services incur many more UDI queries, and deserve more attention as standard replication techniques are unlikely to provide major performance gains. Furthermore, update-intensive services also often incur transactions, which makes the scaling process more difficult. Instead, partial replication or data partitioning techniques should be used so that update queries can be *distributed* among multiple servers. We discuss two representative examples from TPC-W and RUBiS and show how they can be scaled up using relatively simple techniques.

4.3.1 Scaling the financial service of TPC-W

The denormalized TPC-W contains one update-intensive service: the *Financial* service. This service incurs a database update each time a client updates its shopping cart or does a purchase. However, all tables from this service, except one, are indexed by a shopping cart ID and all queries span exactly one shopping cart. This suggests that, instead of replicating the data, one can *partition* them according to their shopping cart ID.

The *Financial* data service receives two types of updates: updates on a shopping cart, and purchase transactions. The first one accesses tables `SHOPPING_CART` and `SHOPPING_CART_ENTRY`. Table `SHOPPING_CART` contains the description of a whole shopping cart, while `SHOPPING_CART_ENTRY` contains the details of one entry of the shopping cart. If we are to partition these data across multiple servers, then one should keep a shopping cart and all its entries at the same server.

The second kind of update received by the *Financial* service is the `Purchase` transaction. We present this transaction in Algorithm 1. Similar to the `Updatecart` query, the `Purchase` transaction requires that the order made from a given shopping cart is created at the same server that already hosts the

```
1 Insert into ORDER with o_id=id;
2 Insert into CC_XACTS with cx_o_id=id;
3 foreach item i within the order do
4     Insert into ORDER_ENTRY with ol_o_id=id, ol_i_id=i;
5     Update I_STOCK set i_stock=i_stock-qty(i) where i_id=i;
6 end
7 Update SHOPPING_CART where sc_id=id;
8 Delete from SHOPPING_CART_ENTRY where scl_sc_id=id;
```

Algorithm 1: The purchase transaction

shopping cart and its entries. This allows one to run the transaction within a single server of the *Financial* service rather than facing the cost of a distributed transaction across replicated servers.

One exception to this easy data partitioning scheme is the `ITEM_STOCK` table, in which any element can potentially be referred to by any shopping cart entry. One simple solution would be to replicate the `ITEM_STOCK` table across all servers that host the *Financial* service. However, this would require to run the `Purchase` transaction across all these servers. Instead, we create an `ITEM_STOCK` table in each server of the *Financial* service in which all item details are identical except the available stock which is *divided* by the number of servers. This means that each server is allocated a part of the stock that it can sell without synchronizing with other servers. Only when the stock available at one server is empty, does it need to execute a distributed transaction to re-distribute the available stock.

The *Financial* service receives two more read queries that access data across multiple data clusters. These queries retrieve respectively the 3333 and 10,000 latest orders from tables `ORDERS` and `ORDER_ENTRY` in order to obtain either the list of best-selling items or the items most related to a given other item. We implement these queries in a similar way to distributed databases. Each query is first issued at each server. The results are then merged into a single result set, and the relevant number of most recent orders is re-selected from the merged results.

In our implementation, we wanted to balance the load imposed by different shopping carts across all servers of the *Financial* service. We therefore marked each row of tables `SHOPPING_CART`, `SHOPPING_CART_ENTRY` and `ORDERS` with a key equal to the shopping cart ID. We then hash this ID to $H = (7id + 4)\%M$ (where M is the number of servers) to determine which server H should be responsible for that row. Our experiments show that this very simple hash function balances the load effectively in terms of data storage size and computational load.

This example shows that, even for relatively complex data services, the fact that each service has simple semantics and receives few different queries allows

one to apply application-specific solutions. The resulting relative complexity of the service implementation, however, remains transparent to other parts of the application, which only need to invoke a simple service interface.

4.3.2 Scaling RUBiS

The denormalized RUBiS implementation contains two update-intensive services: *Auction* and *User*. Similar to the previous example, most queries address a single auction or user by their respective IDs. We were thus able to partition the data rows between multiple servers. A few read-only queries span multiple auctions or users, but we could easily rewrite them such that individual queries would be issued at every server, before their results can be merged.

4.4 Evaluation

As we have seen, RUBBoS and RUBiS are relatively simple to host using our denormalization technique. RUBBoS can be decomposed into several rarely updated data services. On the other hand, RUBiS requires coarser-grain update-intensive services, but they can be scaled relatively easily. We present here performance evaluations of TPC-W, which we consider as the most challenging of the three applications.

Our evaluations assume that the application load remains roughly constant, and focus on the scalability of denormalized applications. To support the fluctuating workloads that one should expect in real deployments, a variety of techniques exist to dictate when and how extra servers should be added or removed from each individual data service of our implementations [Abrahao et al., 2006; Cunha et al., 2007; Urgaonkar et al., 2005b].

We compare three implementations of TPC-W. “OTW” represents the unmodified original TPC-W implementation. We then compare its performance to “DTW”, which represents the denormalized TPC-W where no particular measure has been taken to scale up individual services. Finally, “STW” (scalable TPC-W) represents the denormalized TPC-W with scalability techniques enabled. All three implementations are based on the Java implementation of TPC-W from the University of Wisconsin [TPC-W implementation, 2011]. For performance reasons we implemented the data services as servlets rather than SOAP-based Web services.

We first study the performance of OTW and DTW to investigate the costs and benefits of data denormalization with no scalability techniques being introduced. We then study how replication and data partitioning techniques allow us to scale

individual data services of TPC-W. Finally, we deploy the three implementations on an 85-node cluster and compare their scalability in terms of throughput.

4.4.1 Experimental setup

All experiments are performed on the DAS-3, an 85-node Linux-based server cluster [DAS 3, 2011]. Each machine in the cluster has a dual-CPU / dual-core 2.4 GHz AMD Opteron DP 280, 4 GB of memory and a 250 GB IDE hard drive. Nodes are connected to each other with a gigabit LAN such that the network latency between the servers is negligible. We use Tomcat v5.5.20 as application servers, PostgreSQL v8.1.8 as database servers, and Pound 2.2 as load balancers to distribute HTTP requests among multiple application servers.

Before each experiment, we populate the databases with 86,400 customer records and 10,000 item records. Other tables are scaled according to the benchmark requirements. The client workload is generated by Emulated Browsers (EBs). We use the number of EBs to measure the client workload. The workload model incorporates a think time parameter to control the amount of time an EB waits between receiving a response and issuing the next request. According to the TPC-W specification, think times are randomly distributed with exponential distribution and average value 7 seconds.

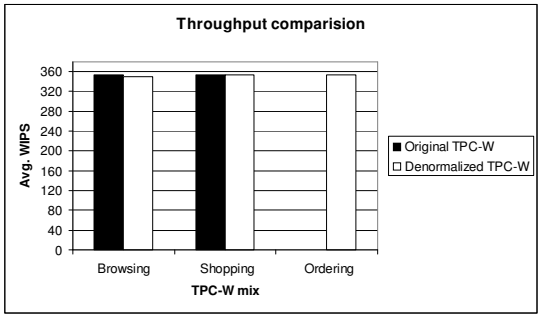
TPC-W defines three standard workloads: the browsing, shopping and ordering mixes, which generate 5%, 20% and 50% update interactions respectively. Unless otherwise specified, our experiments rely on the shopping mix.

4.4.2 Costs and benefits of denormalization

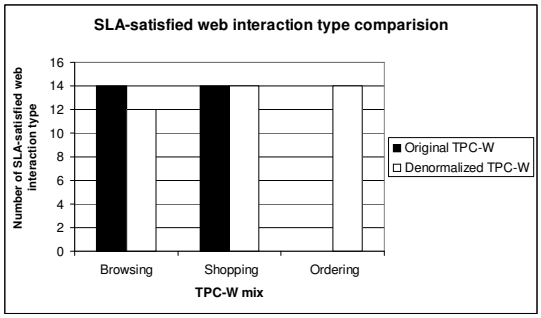
The major difference between a monolithic Web application and its denormalized counterpart is that the second one is able to distribute its UDI workload across multiple machines. Even though such an operation implies a performance drop when hosting the application on a single machine, it improves the overall system scalability when more machines are used. In this section, we focus on the costs and benefits of data denormalization when no special measure is taken to scale the denormalized TPC-W.

We exercise the OTW and DTW implementations using 2500 EBs, under each of the three standard workload mixes. Both systems are deployed over one application server and 8 database servers. In the case of OTW, the database servers are replicated using the standard PostgreSQL master-slave mechanism. DTW is deployed such that each data service is hosted on a separate database server.

We measure the system performance in terms of WIRT (Web Interaction Response Time) as well as WIPS (Web Interactions Per Second). According to the



(a) Average throughput comparison



(b) SLO-satisfied web interaction type number comparison

Figure 4.3: Throughput and performance comparison between original TPC-W and denormalized TPC-W. Note that the Ordering mix for the original TPC-W overloaded and subsequently crashed the application.

TPC-W specification, we defined an SLO in terms of the 90th percentile of response times for each type of Web interaction: namely, 90% of web interactions of each type must complete under 500 ms. The only exception is the "Admin confirm" request type, which does not have an SLO requirement. This request is issued only by the system administrator, and therefore does not influence the client-perceived performance of the system.

Figure 4.3 shows the performance of the different systems under each workload. Figure 4.3(a) shows the achieved system throughput, whereas Figure 4.3(b) shows the number of query types for which the SLO was respected.

The browsing mix contains very few UDI queries. Both implementations sustain roughly the same throughput. However, the denormalized TPC-W fails to meet its SLO for two out of the 14 interaction types. This is due to the fact that the concerned interactions heavily rely on queries that are rewritten to target multiple,

different data services. These calls are issued sequentially, which explains why the corresponding request types incur higher latency.

At the other extreme, the ordering mix contains the highest fraction of UDI queries. Here, DTW sustains a high throughput and respects all its SLOs, while OTW simply crashes because of overload. This is due to the fact that DTW *distributes* its UDI queries across all database servers while OTW *replicates* them to all servers. Finally, the shopping mix constitutes a middle case where both implementations behave equally good.

We conclude that data denormalization improves the performance of UDI queries at the cost of a performance degradation of rewritten read queries. We note, however, that the extra cost of read queries does not depend on the number of server machines, whereas the performance gain of UDI queries is proportional to the size of the system. This suggests that the denormalized implementation is more scalable that the monolithic one, as we will show in the next sections.

4.4.3 Scalability of individual data services

We now turn to study the scalability of each data service individually. We study the maximum throughput that one can apply to each service when using a given number of machines, such that the SLO is respected.

Since we now focus on individual services rather than the whole application, we need to redefine the SLO for each individual data service. As one application-level interaction generates on average five data service requests, we roughly translated the interaction-level SLO into a service-level SLO that requires 90% of service requests to be processed within 100 ms. The *Financial* service is significantly more demanding than other services, since about 10% of its requests take more than 100 ms irrespective of the workload. We therefore relax its SLO and demand that only 80% of queries return within 100 ms.

We measure the maximum throughput of each data service by increasing the number of EBs until the service does not respect its SLO any more. To generate flexible reproducible workloads for each data service, we first ran the TPC-W benchmark several times under relatively low load (1000 EBs) and collected the logs of the invocation of data service interfaces. We obtained 72 query logs, each representing the workload of 1000 EBs for a duration of 30 minutes. We can thus generate any desired workload, from 1000 EBs to 72,000 EBs step by 1000 EBs, by replaying the right number of elementary log files across one or more client machines concurrently.

Figure 4.4 shows the throughput scalability of three representative data services from the scalable TPC-W. The *Item_basic* data service is read-only. It is therefore trivial to increase its throughput by adding database replicas. Similarly,

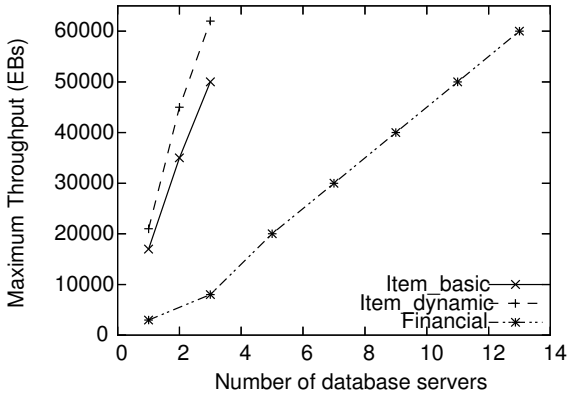


Figure 4.4: Scalability of individual TPC-W services

the *Item_dynamic* service receives relatively few UDI queries, and can be scaled by simple master-slave replication.

On the other hand, the *Financial* service incurs many database transactions and UDI queries, which implies that simple database replication will not produce major throughput improvements. We see, however, that the implementation discussed in Section 4.3.1 exhibits a linear growth of its throughput as the number of database servers increases.

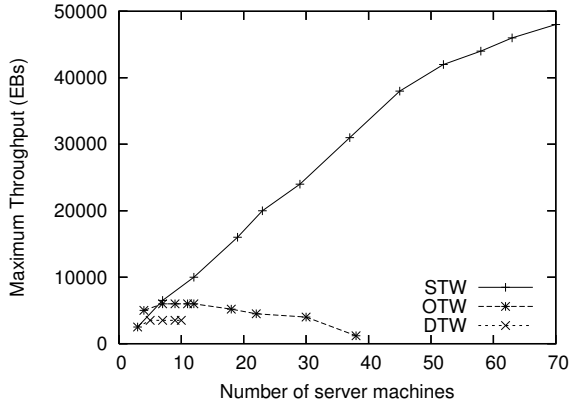
To conclude, we were able to scale all data services to a level where they could sustain a load of 50,000 EBs. Different services have different resource requirements to reach this level, with the *Item_basic*, *Item_dynamic* and *Financial* services requiring 3, 3, and 13 database servers, respectively.

We believe that all the data services can easily be scaled further. We stopped at that point as 50,000 EBs is the maximum throughput that our TPC-W implementation reaches when we use the entire DAS-3 cluster for hosting the complete application.

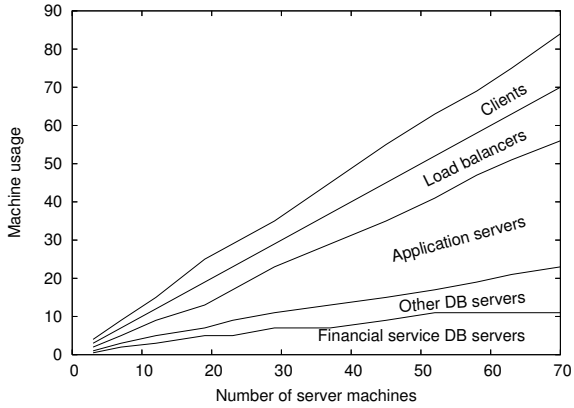
4.4.4 Scalability of the entire TPC-W

We conclude this performance evaluation by comparing the throughput scalability of the OTW, DTW and STW implementations of TPC-W. Similar to the previous experiment, we exercised each system configuration with increasing numbers of EBs until the SLO was violated. In this experiment, we use the application-level definition of the SLO as described in Section 4.4.2.

Figure 4.5(a) compares the scalability of OTW, DTW and STW when using between 2 and 70 server machines. In all cases we started by using one application server and one database server. We then added database server machines to the configurations. In OTW, extra database servers were added as replicas of the



(a) Maximum system throughput



(b) Allocation of machine resources for STW

Figure 4.5: Scalability of TPC-W hosting infrastructure

monolithic application state. In DTW, we start with one database server for all services, and eventually reach a configuration with one database server per service. In STW, we allocated the resources as depicted in Figure 4.5(b). Note that in all cases, we deliberately over-allocated the number of application servers and client machines to make sure that the performance bottleneck remains at the database servers.

When using very few servers, OTW slightly outperforms DTW and STW. With increasing number of servers, OTW can be scaled up until about 6000 EBs when using 8 servers. However, when further adding servers, the throughput decreases. In this case, the performance improvement created by extra database replicas is counterbalanced by the extra costs that the master incurs to maintain consistency.

As no individual scaling techniques are applied to DTW, it can be scaled up to at most 8 database servers (one database server per service). The maximum throughput of DTW is around 3500 EBs. Note that this is only about half of the maximum achievable throughput of OTW. This is due to the extra costs brought by data denormalization, in particular the rewritten queries. Adding more database servers per service using database replication would not improve the throughput, as most of the workload is concentrated in the *Financial* service.

Finally, STW shows near linear scalability. It reaches a maximum throughput of 48,000 EBs when using 70 server machines (11 database servers for the *Financial* service, 12 database servers for the other services, 33 application servers and 14 load balancers). Taking into account the 14 client machines necessary to generate a sufficient workload, this configuration uses the entire DAS-3 cluster. The maximum throughput of STW at that point is approximately 8 times that of OTW, and 10 times that of a single database server.

We note that the STW throughput curve seems to start stabilizing around 50 server machines and 40,000 EBs. This is not a sign that we reached the maximum achievable throughput of STW. The explanation is that, as illustrated in Figure 4.4, 40,000 EBs is the point where many small services start violating their SLO with two database servers, and need a third database server. In our implementation each database server is used for a single service, which means that several extra database servers must be assigned to the small data services to move from 40,000 EBs to 50,000 EBs. We expect that using more resources the curve would grow faster again up to the point where the small data services need four servers.

4.5 Discussion

Most approaches for scalable hosting of Web applications consider the application code and data structure as constants, and propose middleware layers to improve performance transparently to the application. We take a different stand and demonstrate that major scalability improvements can be gained by allowing one to denormalize an application's data into independent services. While such restructuring introduces extra costs, it considerably simplifies the query access pattern that each service receives, and allows for a much more efficient use of classical scalability techniques. Importantly, data denormalization does not imply any loss in terms of consistency or transactional properties. This aspect makes our approach unique compared to, for example, Gao et al. [2003].

In our experience, designing the denormalized data schema of an application from its original data structure and query templates takes only a few hours. On the other hand, the work required for the actual implementation of the required changes largely depends on the complexity of each data service. For instance, we

investigated the time required for denormalizing MediaWiki, a real-world application used by Wikipedia [Wikipedia, 2011d]. This work focused on denormalizing a simplified version of MediaWiki which supports basic functions for article and image management such as read, update and search [Li, 2010]. This process roughly cost 6 person-month. We argue that future Web applications should preferably be designed from the start along a service-oriented architecture. In such case, most implementation work such as re-writing application code and database queries can be saved.

Data denormalization exploits the fact that an application's queries and transactions usually target few data columns. This, combined with classical database denormalization techniques such as query rewriting and column replication, allows us to cluster the data into disjoint data services. Although this property was verified in all applications that we examined, one cannot exclude the possible existence of applications with sufficient data overlap to prevent any service-oriented denormalization. This may be the case of transaction-intensive applications, whose ACID properties would impose very coarse-grained data clustering. It is a well-known fact that database transactions in a distributed environment imply important performance loss, so one should carefully ponder whether transactions are necessary or not.

The fact that denormalization is steered by prior knowledge of the application's query templates means that any update in the application code may require to restructure the data to accommodate new query templates. However, the fact that all data services resulting from denormalization have clear semantics makes us believe that extra application features could be implemented without the need to redefine data services and their semantics.

4.6 Conclusion

This chapter demonstrates that the scalability of a Web application can be improved by denormalizing the application into a multi-service architecture. In order to guarantee the performance of such a multi-service application under fluctuating workload, one needs to dynamically provision this application. For instance, in the experiments from section 4.4.4 we dynamically added extra machines to both servlet services and data services when the overall performance violated the SLO. Figure 4.5(b) shows the resource provisioning results. However, to decide the service(s) to provision and the number of machines to provision, we had to use an exhaustive trial-and-error search among all possible provisioning choices at each provisioning step. Such an exhaustive search approach can obviously not work in a production environment. We discuss better techniques to provision such multi-service Web applications in the next chapter.

Chapter 5

Resource provisioning for multi-service Web applications

Scalability techniques from Chapter 4 result in designing Web applications that do not follow classical multi-tier organizations. Instead, denormalized applications are composed of a graph of services querying each other. Such structures are also often found in other large-scale Web applications. For example, major web sites such as Amazon.com and eBay are also not designed as a monolithic 3-tier application but as a complex group of independent services querying each other [Vogels and Gray, 2006; Shoup, 2008]. A service is a self-contained application providing elementary functionality, such as a database holding customer information or an application targeted at serving search requests.

Services hide their internal implementation details from the outside world and expose functionality through standard invocation interfaces. Typically the services participating in an application are composed in a directed acyclic graph. Web pages delivered to the users are generated by composing the results of many such services based on pre-defined workflows [Shoup, 2008]. An application's call graph between services may be known either thanks to explicit declaration by the programmers, or automatically [Mann et al., 2011].

To guarantee the performance for such Web applications, dynamic resource provisioning must take this multi-service organization into account. At each adaptation, one must make decisions on *which* service(s) should be (de-)provisioned. Such decisions are essential to be able to guarantee performance at minimal cost. However, as discussed in Chapter 3, even for a simple two-tier Web application, the decision search space is very large. Deriving an optimal decision path from an entire space exploration is impossible in production environments. This becomes even harder when provisioning multi-service Web applications, as they involve a large number of components that have complex re-

lationships with each other. For example, adding a cache to one service does not only improve its own response time, but also causes less traffic to the backend services it invokes. Predicting the performance effects of individual services on the overall application is non-trivial. Therefore, taking optimal provisioning decisions in multi-service applications remains a challenge.

One possible approach models the entire application as a single queuing network. However, modeling even a very simple application accurately can be difficult. A performance model for a multi-service application can become extremely complex to capture all service relationships, even without considering advanced provisioning options such as adding caching servers.

Another approach assigns a fixed SLO to each service separately. The SLO of the front-end service is trivially defined as the response time objective of the whole application. However, as we will show in this chapter, no choice of internal service SLOs can match the performance of our system: the per-service SLO approach necessarily wastes resources as it makes certain services struggle to maintain their SLOs when an equivalent gain in end-to-end performance could be obtained easier by reprovisioning another service.

In this chapter, we claim that only the front-end service should be given an SLO. A user commonly does not care about the performance of each particular service involved in the application, but only in the end-to-end response time that she observes. On the other hand, no service other than the front end should have a specific SLO. Instead, each service should be autonomously responsible for its own provisioning by collaboratively negotiating its performance objectives with the other services to maintain the front end's response time within the SLO. Negotiation between services is based on "what-if analysis:" each service continuously estimates the performance it would have in case it was assigned more/less resources, or if it received more/less traffic. The front-end service finally selects the optimal service(s) for resource provisioning from the perspective of the whole application.

We first describe the design of our system, and present the provisioning mechanisms for multi-service Web applications involving different service invocation patterns. We also discuss the issues of provisioning cache instances. Furthermore, in the case of subtle changes in workload patterns, a previous provisioning decision may need to be revoked without adding or removing resources, but by reassigning resources from one service to another. Our system allows such resource reorganization so as to accommodate long-term changes in user behavior. Finally, we conduct extensive experiments to show that our system allows one to effectively provision resources to both traditional multi-tier Web applications and multi-service Web applications under varying workload conditions, such as load intensity and locality.

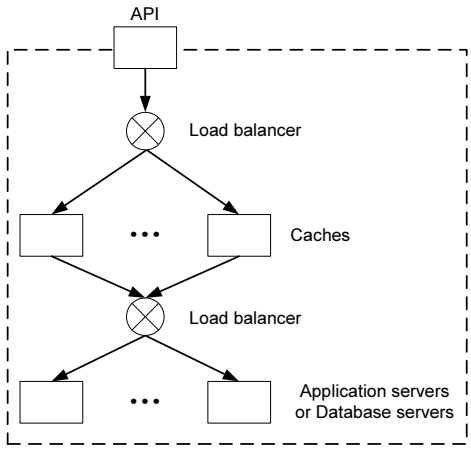


Figure 5.1: Hosting architecture of a single service

5.1 System design

5.1.1 System model

We define a service as either a single-tier functional service with an HTTP or SOAP interface hosted in an application server, or a single-tier data service with an SQL interface hosted in a database server. Although in real systems services may be composed of an application server and a database server, for provisioning we consider these as separate services. Within a multi-service application, services are commonly organized as a directed acyclic graph. We assume that inter-service invocations are synchronous and that the services of one application are not used simultaneously by other applications (which means that the directed acyclic graph has a single root node).

Figure 5.1 shows how a service is typically hosted. A service may have multiple instances representing multiple application servers with a copy of the service code or multiple database servers containing a replica of the service's data. To improve performance, a service may possibly employ one or more machines as caches that intercept incoming requests before accessing the service itself. We use consistent hashing to distribute cached objects across multiple caches [Karger et al., 1997]. This means in particular that increasing the number of caches attached to a service generates the same hit ratio as increasing the storage space of a single cache.

We assume that some machines are always available to be added to an application, as is commonly the case in environments such as a data center or in cloud computing. Our system relies on an exclusive provisioning model: each resource

can be assigned to only one service at a time. Such resource may be a physical machine or a virtualized instance with performance isolation as for example in Amazon EC2.

Figure 5.2 illustrates our approach based on an invocation tree consisting of 7 services. Resource provisioning is done in two steps. First, each service carries out "what-if analysis" checks to predict its future performance in case it was assigned an extra machine or one had been removed. These prediction results can be seen as a performance promise made by the service to its parent in the invocation tree. Predictions are realized by a provisioning agent attached to each service. Each service periodically sends its performance promises to its parent in the invocation tree.

In the second step services negotiate resources with each other. Each intermediate node in the invocation tree negotiates performance promises with its parent on behalf of itself and all its children nodes. This intermediate service is responsible for all local resource provisioning decisions among its own subtree. A local decision consists of selecting the maximum performance gain (or minimum loss) among the service's child nodes and itself. For example, in Figure 5.2, services 2 and 3 report their performance promises to service 1 but the promise of service 2 is an aggregate among its own promises and those of services 4, 5 and 6.

Finally, the root node selects which service(s) to provision across the tree when the SLO is (about to be) violated, or to deprovision when this is possible without violating the SLO.

5.1.2 Performance model of a single service

A good performance model in our system should not only explain the current performance of a given concerned service, but also predict its future performance if one more or one less machine was assigned to host the service. Additionally, it should predict future performance in case its received request rate would increase or decrease. We first present the model itself, then discuss its parameterization.

5.1.2.1 Performance model

We model a single-core machine as an M/M/1/PS queue, which is widely adopted in practice [Gunther, 2004]. Similarly, multi-core machines distribute their load evenly on each CPU core. Consequently, we use an M/M/n/PS queue to capture the performance of an n-core machine. We assume that all CPU cores of the provisioning machines are homogenous.

The performance model calculates the expected response time after adding or removing one server (such as application server or database server) as follows:

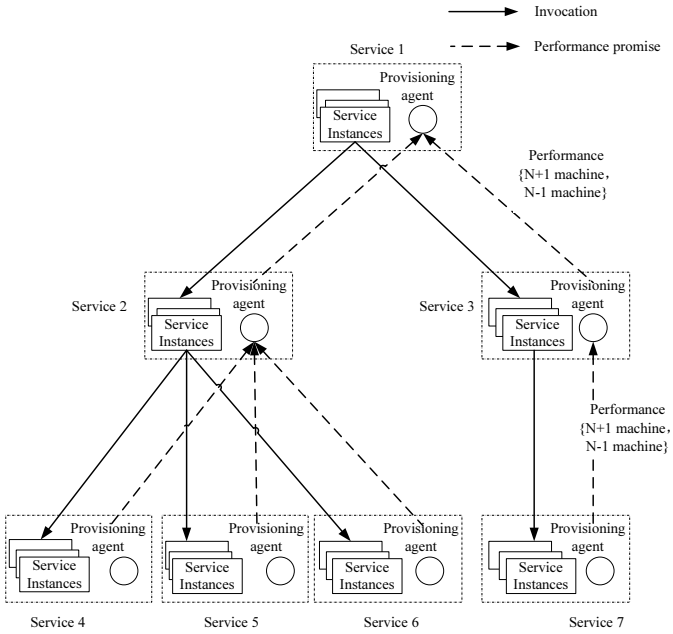


Figure 5.2: Autonomous resource provisioning system model

$$\Delta R_{server} = R(n \pm 1)_{server} - R(n)_{server} \tag{5.1}$$

$$R(n)_{server} = \frac{S_{server}}{1 - \frac{\lambda S_{server}}{n}} \tag{5.2}$$

where R_{server} is the average response time of the service, n is the number of CPU cores assigned to the service, λ is the average request rate and S_{server} is the mean service time of the server.

A service may also use caches to offload some of the incoming requests from the service itself. This is especially common in Web applications when the request locality is high. On the other hand, caches may also waste useful resources if the request locality is low. Adding caches potentially improves response time for two reasons. First, cache hits are processed faster than cache misses. Second, the service itself and all children nodes receive less requests, and can thus process them faster. After adding a cache, the service response time consists of the cache fetching time and the sojourn time in the service upon every cache miss. The performance model calculates the caching impact on the response time as follows:

$$\Delta R_{cache} = R(n \pm 1)_{cache} - R(n)_{cache} \tag{5.3}$$

$$R(n)_{cache} = p_n S_{cache}(n) + (1 - p_n) R(m) \tag{5.4}$$

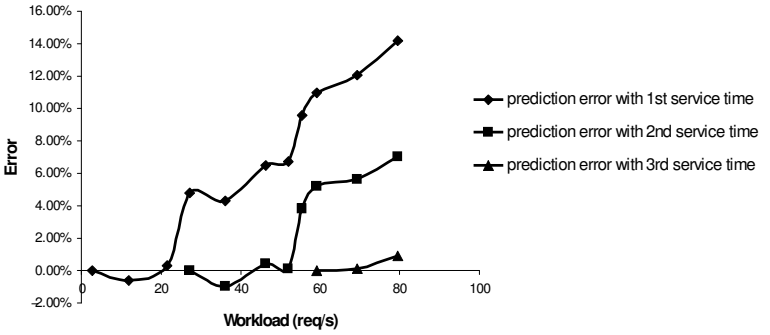


Figure 5.3: Dynamic service time correction

where $R(m)$ is the response time of the backend server across m CPU cores, S_{cache} is the cache service time, which is identical to the cache response time based on Little's Law [Trivedi, 2002], and p_n is the expected cache hit ratio with n nodes.

5.1.2.2 Model parameterization

Most of the model parameters can be measured offline or monitored at runtime. For example, the request rate can be monitored by the administrative tools of application servers and database servers. The cache service time can be obtained by measuring cache response time offline. However, the expected cache hit ratio p_n and the mean service time S_{server} are harder to measure.

We estimate the new cache hit ratio after a reconfiguration if one machine was added to or removed from the caching tier using virtual caches [Sivasubramanian, 2007]. A virtual cache is an online cache that stores only metadata such as the list of objects in cache and their sizes, but not the objects themselves. It receives all requests directed to the service and applies the same operations as a real cache with the same configuration would. It can thus estimate the hit ratio that a cache of any given size would have under the current workload.

Another crucial parameter is the service time S_{server}. Previous research works measure the service time via profiling under low workload [Sivasubramanian, 2007; Urgaonkar et al., 2005a,b]. However, we found that the service time changes under different workloads, probably because of extra overhead in the server implementation that is not captured by an M/M/n/PS queue. We illustrate this in Figure 5.3. We first measure the service time of a database service under a low workload of 1 req/s. We then use this value to predict the response time of the service under other workloads. The curve with the diamond label indicates the prediction error under various workloads compared to the corresponding measured value. The error initially remains close to 0. However it later increases and finally

reaches 14%, which is not acceptable for our purpose.

To achieve acceptable prediction results, we apply a classical feedback control loop to adjust the service time at runtime. The system continuously estimates the service's response time under the current conditions and compares the error between the predicted response time and the measured one. One can define a threshold as a configuration parameter. When the prediction error exceeds the threshold, the correction mechanism recomputes the service time:

$$S^*_{server} = \frac{nR_{server}}{n + \lambda R_{server}}$$

where S^*_{server} is the corrected service time, R_{server} is the latest measured response time, n is the number of current CPU cores, and λ is the current request rate.

Figure 5.3 shows the effectiveness of this mechanism. We define the error threshold as 5% and apply the correction mechanism to the whole prediction process carried out for the curve with diamond label. When the workload reaches 28 req/s, the prediction error exceeds the threshold. The control system then recomputes the service time. The curve with the square label presents the prediction error with this second service time value. Compared with the error caused by the original service time measured offline, this corrected service time causes much fewer errors when the workload increases further. Similarly, the control system triggers the correction again around 53 req/s. The curve with the triangle label displays the further prediction error, which is again within the limits.

The system also maintains a memory of the service time values that should be used for various workload intensities.

5.1.3 Resource provisioning of service instances

Resource provisioning within a multi-service application is based on negotiation among services, where services continuously exchange performance promises generated by the performance model. We first discuss the case where services are organized in a tree pattern and only service instances are added or removed, then extend to directed acyclic graphs.

5.1.3.1 Hierarchy structure

Multi-service applications are often organized along a hierarchical structure. To find out which service(s) should be reprovisioned, services exchange their future performance objectives if a resource reconfiguration would happen. Each service reports performance promises to its parent on behalf of its children and itself: it reports the best performance gain (resp. loss) possible by adding (resp. removing)

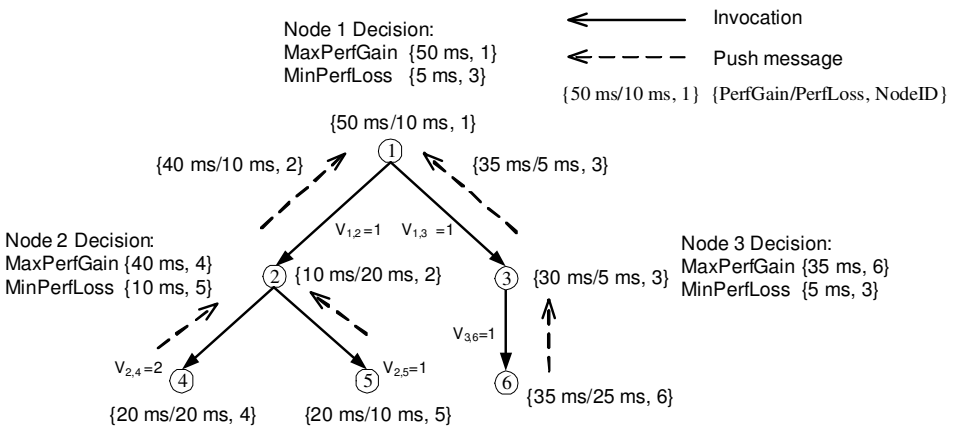


Figure 5.4: Resource provisioning in hierarchical structures

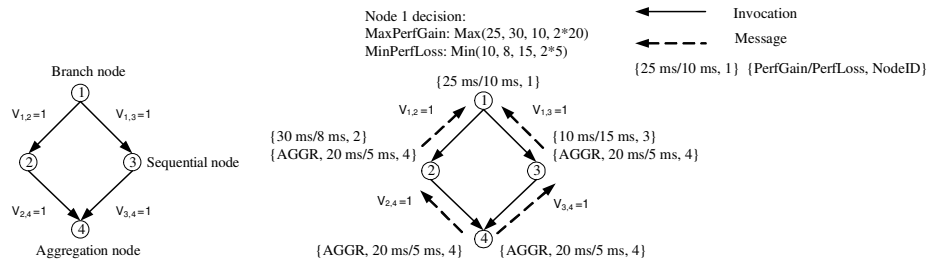
a server to (resp. from) a service of the subtree consisting of its child nodes and itself.

Figure 5.4 illustrates the decision processes within a typical hierarchical structure. The decision process between service 2 and its children 4 and 5 is the smallest decision unit in the whole application. Here, services 4 and 5 are responsible for reporting their performance promises to service 2. To generate its own promises, service 2 must find the maximum performance gain (resp. minimum loss) that the entire subtree can achieve with one more (resp. one less) machine. Assuming a service i has k immediate child services, it aggregates its own performance promises as follows:

$$MaxPerfGain = max\{V_{i,j} \cdot MaxPerfGain_j\} \quad (1 \leq j \leq k) \tag{5.5}$$

$$MinPerfLoss = min\{V_{i,j} \cdot MinPerfLoss_j\} \quad (1 \leq j \leq k) \tag{5.6}$$

where $V_{i,j}$ is the average number of service executions on service j caused by one request from service i. For example, in Figure 5.4 each request from service 2 results on average in two service executions on service 4 and only one on service 5. This parameter can be measured online by services 4 and 5 by comparing their local request rate with that of their parent. The child nodes adaptively adjust this parameter when they observe that the ratio changes. Here, although service 4 would gain 20 ms if it was given one more machine, the actual performance gain brought by service 4 to service 2 is 40 ms due to the double invocation ratio. Service 2 compares the received promises with its own local ones, and makes the local decision: if given one more machine, it should give it to service 4 which generates the greatest performance gain overall. If requested to release one machine, it



(a) Directed acyclic invocation graph

(b) Resource provisioning process

Figure 5.5: Resource provisioning in directed acyclic graphs

should remove it from service 5 which would incur the lowest global performance loss.

The same process is repeated at every level of the tree up to the root, which has sufficient information to take provisioning decisions upon variations in request rate. Here, service 1 can finally make the global decision: if given one more machine, it should keep it to itself as it can obtain the maximum performance gain. If removing one machine, it should remove it from service 3 as this causes minimum performance loss. Once service 1 decides to change resource allocation, it triggers the reconfiguration by sending a notification to the concerned service.

5.1.3.2 Directed acyclic invocation graphs

In real-world applications, multiple services may commonly share the same backend. For example, in Figure 5.5(a) service 4 may be a database accessed by multiple web services. We define a shared service as an aggregation node in the invocation path.

Any performance promise made by an aggregation node or any of its children has the same effect to each invocation path from the root node to the aggregation node. For example, in Figure 5.5(a), there are two invocation paths from root node 1 to aggregation node 4: $1 \rightarrow 2 \rightarrow 4$ and $1 \rightarrow 3 \rightarrow 4$. Assuming that aggregation node 4 would gain 20 ms due to its own resource reconfiguration, then root node 1 would gain a total performance improvement of 40 ms. The negotiation mechanism should reflect the multi-path performance effect of the aggregation node.

Aggregation nodes report their promises along each invocation path with special "AGGR" identifications. In Figure 5.5(b), service 4 sends {AGGR, 20 ms/5 ms, 4}. This means service 4 is an aggregation node, and would gain 20 ms with an extra machine or lose 5 ms with one less machine.

"AGGR" messages are handled differently than regular promises. Any node receiving such message should forward it upwards to the root in addition to the regular promises. If a node receives multiple "AGGR" messages originating from the same node ID, it must add them together before forwarding. Finally the root node compares performance promises from regular messages and the ones from "AGGR" messages to make its global decision. For example, in Figure 5.5(b), service 1 receives two "AGGR" messages with the same node ID 4. It thus adds them as the performance promise of service 4. As service 1 is the root node, it also compares other performance promises with the merged result 2*20 ms, and finds the maximum one as the final decision.

5.1.4 Resource provisioning of cache instances

Thus far we discussed only provisioning of service instances. Provisioning cache instances is harder because it not only changes the performance of the concerned service, but also changes the traffic to its children, which in turn affects their performance. Thus, each service should also calculate the performance it would have if it addressed more or less traffic.

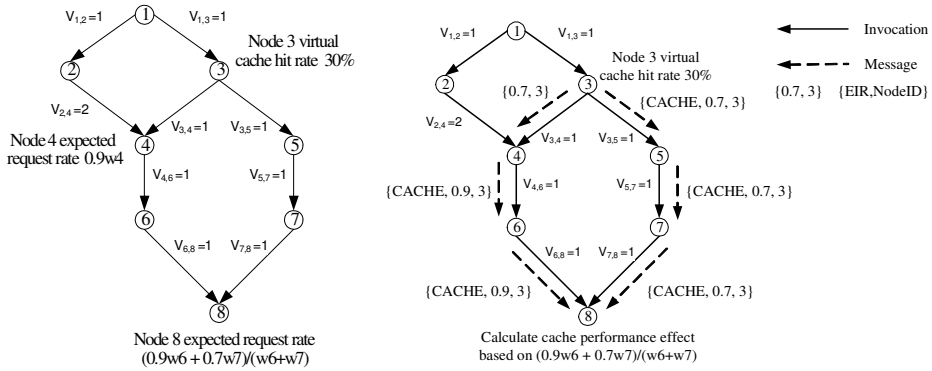
When considering whether to add or remove a cache to itself (instead of a service instance), each service must take into account the future expected performance of all its child services if they would receive more/less traffic.

In our system, each node operates two virtual caches with different sizes matching the situations where the service would be assigned one more or one less cache instance. Each service periodically informs its children of the relative workload decrease (resp. increase) it would address to them if it was given one more (resp. one less) cache instance. This expected invocation ratio EIR on the node originating cache reconfiguration is equal to the expected miss rate:

$$EIR = ExpectedMissRate \tag{5.7}$$

In such cases the children can anticipate a decrease or increase of the traffic they receive. We illustrate this information exchange process for the cache effect calculation in Figure 5.6(b), which features a complex situation with multiple aggregation nodes. When a node j receives the "CACHE"-labeled messages including expected invocation ratios from its parents, it first computes its local expected workload intensity as the sum of expected request rates promised by its predecessors:

$$w_j^* = \sum_{i=1}^{k} V_{i,j} * EIR_i * w_i$$



(a) Complex directed acyclic invocation graph

(b) Resource provisioning process

Figure 5.6: Resource provisioning in complex directed acyclic graphs

where $V_{i,j}$ is the average number of service executions on service j caused by one request from service i, EIR_i is the expected invocation ratio of parent i, w_i is the request rate of node i, and k is the number of its predecessors in the invocation graph. Then the node calculates its own expected invocation ratio:

$$EIR_j = \frac{w_j^*}{w_j}$$

For example, in Figure 5.6(b), node 4's expected invocation ratio is:

$$EIR_4 = \frac{w_4^*}{w_4} = \frac{w_2 + EIR_3 * w_3}{w_4} \tag{5.8}$$

The concerned node j forwards its expected invocation ratio EIR_j to its children, then calculates its own expected performance under its expected workload intensity. Finally, it returns the calculated performance objectives to all its parents.

In a directed acyclic graph, a performance change in an aggregation node affects all its predecessor branches but also other branches as well. For example, adding a cache to service 2 in Figure 5.6(b) would change the performance of service 4, and thereby also affect service 3. The "AGGR" messages are also employed to propagate information about these cascading effect through the invocation graph.

Note that, even though the system may need to propagate many "AGGR" messages simultaneously, there is no combinational explosion: in the worst case, the number of "AGGR" messages processed by a node is linear to the number of nodes in the invocation graph.

5.1.5 Shifting resources among services

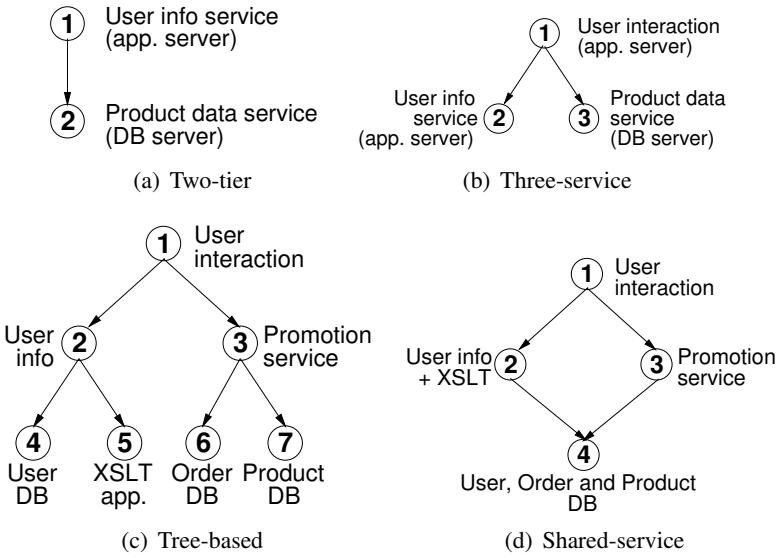
In many cases, instead of provisioning extra resources, it can be more efficient to simply reorganize resource assignments within the application without retrieving machines from the resource pool. Such reorganization may be necessary to follow changes in access patterns. For example, in Figure 5.4, the values $V_{2,4}$, $V_{2,5}$ and $V_{2,6}$ may change due to an update in the application code or a change in user behavior. Our system should reorganize the resource assignments so as to increase resource usage, and therefore improve application performance.

One could imagine letting each intermediate service shift resources autonomously within its children and itself. However, this could lead to inefficiencies such as having the application from Figure 5.4 shift resources from service 4 to service 5 (initiated by service 2), immediately followed by shifting the same resource again from service 5 to service 3 (initiated by service 1). We therefore prefer letting only the root node be responsible for such reconfigurations.

To prevent oscillating behavior, one should first define a performance improvement threshold as the criterion for deciding whether to shift resources. In a hierarchical invocation case, each service should compose its performance objectives in case one machine was shifted from the service having minimum performance loss to the one having maximum performance gain within the tree. These promises can get aggregated up in the invocation graph such that the root node finally selects the greatest reorganization performance promise and triggers the reconfiguration

In a directed acyclic graph, only the root node has complete information about performance promises from "AGGR" messages. In such case, any node receiving "AGGR" messages does not compose performance objective but forwards "AGGR" messages upwards. Instead, the root node is responsible for finding the maximum performance gain and minimum performance loss and composing these two values as the performance objective of shifting resources.

Note that when one shifts a cache resource upwards within the same invocation path, the affected node to which resources are shifted changes the traffic to all its children and itself. To help generate performance promises in this special case, each service should send expected performance objectives if addressed with the expected request rate. In the case a service shifted one machine upwards to any service on the same path as a cache, this service would serve requests with one machine less. Therefore, each service should send its expected performance objectives under the expected request rates on both the original resource configuration and the updated one.



(a) Two-tier (b) Three-service

(c) Tree-based (d) Shared-service

Figure 5.7: Web applications under test

5.2 Evaluation

This section first validates the performance model discussed in Section 5.1.2. We then compare our system with two state of the art representatives. Finally, we demonstrate the unique features of our approach for provisioning directed acyclic graphs of services querying each other.

5.2.1 Experiment setup

We evaluate our system using four reference applications depicted in Figure 5.7. Figure 5.7(a) shows a classical two-tier application. The application server tier receives HTTP requests and issues one query to the database to search for items related to the last ones purchased by the concerned client. It then applies CPU-intensive XSLT transformations to transform XML templates into HTML.

Figure 5.7(b) shows a three-service application with similar features to the first application. Here, however, the "User interaction" servlet first invokes the "User info" service through a SOAP interface and then the "Promotion" data service through a SQL interface.

The application in Figure 5.7(c) follows a strict tree-like invocation pattern. The root service invokes the left branch for gathering user information, and then the right branch for promoting product information to the same user. The "User info" service in turn accesses user data from the "User" data service, then invokes

Table 5.1: Model validation for XSLT service

# App. servers	# Caches	Request rate	Measured resp. time	Predicted resp. time
1	0	36 req/s	488.3 ms	N/A
1	1	36 req/s	172.3 ms	177.1 ms (+2.8%)
2	0	36 req/s	111.0 ms	**116.0 ms (+4.5%)**
2	1	90 req/s	125.6 ms	**131.2 ms (+4.4%)**
3	0	90 req/s	135.1 ms	139.8 ms (+3.5%)

Table 5.2: Model validation for Product service

# DB servers	# Caches	Request rate	Measured resp. time	Predicted resp. time
1	0	10 req/s	449.0 ms	N/A
1	1	10 req/s	209.0 ms	**219.0 ms (+4.8%)**
2	0	10 req/s	263.1 ms	271.4 ms (+3.2%)
1	2	18 req/s	111.6 ms	**112.7 ms (+1.0%)**
2	1	18 req/s	199.8 ms	201.8 ms (+1.0%)

an external "XSLT" service to transform XML templates into HTML. The "Promotion" service in the right branch first fetches order histories from the "Order" data service, then searches for items related to a user's last orders using the "Product" data service in order to recommend further purchases. Finally, the root service combines the results from the two branches in one web page and returns it to the client.

The last application in Figure 5.7(d) is similar to the third one but is structured so that all "User", "Order" and "Product" data are stored in a single, shared data service. The "User info" service also handles the XML transformation.

In all experiments, we emulate various numbers of end-user browsers which send requests to the applications following a Poisson distribution of arrival times. This distribution has been shown to be realistic for many Internet systems [Villela et al., 2007]. We implement the local performance monitor on the application server using the MBean servlet from JBoss. The database server monitoring is based on performance data collected by the admin tool of MySQL. We developed the negotiation agent in Java using plain sockets. All experiments are performed on the DAS3 cluster at VU University Amsterdam [DAS 3, 2011]. During the whole experiments, we set the prediction error threshold for dynamically adjusting the service time to 3%.

Table 5.3: Resource provisioning of two-tier Web application

# APP servers	# DB servers	# App caches	# DB caches	Request rate	Measured resp. time	Autonomous prediction	Analytic prediction
1	1	0	0	12 req/s	552.6 ms	N/A	N/A
2	1	0	0	12 req/s	303.5 ms	**309.1 ms (+1.8%)**	**296.2 ms (-2.4%)**
1	2	0	0	12 req/s	419.8 ms	443.0 ms (+5.5%)	447.2 ms (+6.5%)
1	1	1	0	12 req/s	515.3 ms	543.2 ms (+5.4%)	503.4 ms (-2.3%)
1	1	0	1	12 req/s	405.0 ms	391.4 ms (-3.4%)	389.8 ms (-3.8%)
2	1	0	0	18 req/s	511.2 ms	N/A	N/A
3	1	0	0	18 req/s	481.4 ms	473.5 ms (-1.6%)	491.1 ms (+2.0%)
2	2	0	0	18 req/s	210.1 ms	**223.3 ms (+6.3%)**	**197.3 ms (-6.1%)**
2	1	1	0	18 req/s	498.7 ms	505.2 ms (+1.3%)	507.2 ms (+1.7%)
2	1	0	1	18 req/s	241.4 ms	230.6 ms (-4.5%)	243.2 ms (+0.7%)

5.2.2 Model validation for single service

Before focusing on resource provisioning, we first validate our performance model using the "XSLT" and "Product" services from Figure 5.7(c) separately. The two services are, respectively, application-server intensive and database-server intensive. We set the SLO of each service to a maximum response time of 400 ms, and initially assign one server to each. We then increase the request rates until the SLO is violated. At that time, we issue performance predictions in case one more machine was assigned as a server replica or a cache, and compare predicted values with the measured response times after applying adaptations. Tables 5.1 and 5.2 show the results at two prediction points for the two services separately.

The first SLO violation of the "XSLT" service occurs around 36 req/s. Prediction errors of adding a server replica and adding a cache are under 5%. Results clearly show that adding a second server is more efficient in this case. We perform the adaptation and increase workload until 90 req/s when the SLO is violated again. Here as well the prediction errors remain under 5%. Similarly, the first SLO violation of the "Product" service occurs around 10 req/s. Both prediction errors are also under 5%. We add a cache to the service and increase workload

until 18 req/s when the SLO is violated again. The prediction errors again remain very low, which confirms the accuracy of our model.

5.2.3 Comparison with the state of the art

We now compare our system with two representatives of the state of the art in resource provisioning. One of the most cited papers on resource provisioning for multi-tier applications is [Urgaonkar and Shenoy, 2005]. Their approach is based on analytic models and thus we name it "Analytic" in this section while we name our system "Autonomous". The second approach assigns a fixed SLO to each service individually.

5.2.3.1 Comparison with Analytic

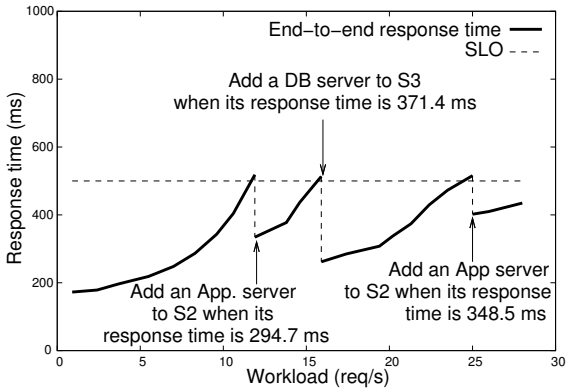
Analytic is designed to provision resources in multi-tier Web applications such as the two-tier application from Figure 5.7(a). We demonstrate here that both schemas work equally well for such applications. We assign an SLO of 500 ms for the whole application, and initially assign one application server and one database server to the application. As the performance model in [Urgaonkar and Shenoy, 2005] is based on single-core single-CPU machines, we run our experiments using only one core of each hosting machine on the DAS3 cluster.

We provision both servers and caches to the tested Web application. We increase the workload to obtain two successive adaptations. We record provisioning decisions of each schema and compare their predicted response times with the measured ones. Table 5.3 shows that both schemas issue slightly different performance predictions but take the same provisioning decisions: at 12 req/s, both systems add an application server to the application. At 18 req/s, both systems add a database server.

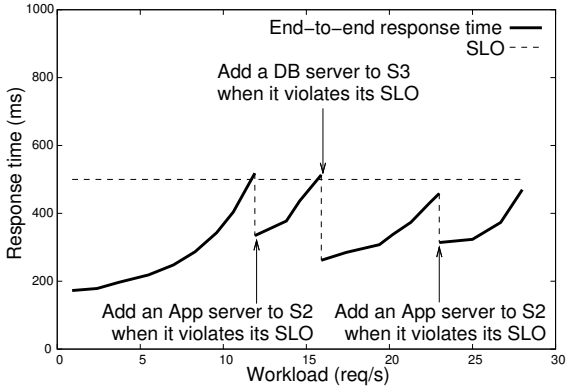
In all cases the prediction errors are lower than 7%, which confirms that both approaches can provision multi-tier applications with similar accuracy. On the other hand, Analytic does not address multi-service applications organized in hierarchical or directed acyclic graph patterns.

5.2.3.2 Comparison with per-service SLO

We now compare our system with the per-service SLO approach. This approach is popular in complex multi-service applications as in Figures 5.7(c) and 5.7(d). However, we claim that it often uses unnecessary resources due to the impossibility of defining suitable SLOs for internal services. We illustrate this using the application in Figure 5.7(b). We define the global application SLO as 500 ms, and



(a) Autonomous provisioning



(b) Per-service SLO provisioning

Figure 5.8: Comparison between our system and per-service SLO

run the two systems across three successive adaptations. For simplicity, in this section we do not consider cache provisioning.

We first use our system to provision the test application. As shown in Figure 5.8(a), our system adds an application server to service 2 at 12 req/s, then a database server to service 3 at 16 req/s, and finally another application server to service 2 at 25 req/s.

We now show that it is impossible to give a fixed SLO to service 2 such that the per-service SLO approach takes optimal provisioning decisions. We set the SLO of the front-end service to 500 ms, identical to the SLO of the whole application. The best possible SLO for service 2 in this case is 290 ms: it allows the system to reprovision service 2 at 12 req/s (which we know to be the optimal decision in this case), just before the application would violate its global SLO. Similarly, we set the SLO of service 3 to 365 ms.

Table 5.4: Prediction accuracy under increasing workload for tree application

	Add a cache at 10 req/s	Add a server at 10 req/s	Add a cache at 18 req/s	Add a server at 18 req/s
Serv. 1	520.9 ms (-1.1%)	522.5 ms (-1.3%)	492.2 ms (+1.2%)	500.8 ms (+1.6%)
Serv. 2	518.6 ms (+0.6%)	524.1 ms (-0.4%)	481.0 ms (+2.0%)	496.3 ms (+2.6%)
Serv. 3	493.3 ms (+2.3%)	501.9 ms (+1.9%)	503.3 ms (+1.8%)	510.8 ms (+1.9%)
Serv. 4	525.5 ms (-1.0%)	525.0 ms (-1.0%)	511.4 ms (+1.4%)	507.9 ms (+1.6%)
Serv. 5	489.2 ms (+2.9%)	409.1 ms (+3.3%)	453.2 ms (+3.0%)	**401.9 ms (+2.8%)**
Serv. 6	525.1 ms (-1.0%)	524.8 ms (-1.2%)	518.6 ms (+1.0%)	508.0 ms (+1.1%)
Serv. 7	**305.3 ms (+5.0%)**	399.1 ms (+3.8%)	449.5 ms (+2.0%)	463.8 ms (+1.6%)

Figure 5.8(b) shows the performance of the per-service SLO approach. The first two adaptations are identical to those of our own system. However, at 23 req/s service 2 violates its internal SLO although the application as a whole does not violate the global SLO. The per-service SLO strategy therefore adds a server to service 2 at 23 req/s, which is wasteful between 23 req/s and 25 req/s.

Selecting other values for the internal SLOs leads to even worse performance. If the SLO of service 2 was set lower than 290 ms, then the per-service SLO approach would reprovision service 2 too early at the first adaptation already. On the other hand, if its internal SLO was set to a greater value than 290 ms, then at the first adaptation this strategy would reprovision the front-end instead of service 2, which does not gain enough performance to maintain the application within its global SLO.

The per-service SLO approach allows one to provision arbitrary multi-service applications. However, even when configured with the best possible internal SLO values, it uses more resources than our proposed system.

5.2.4 Provisioning of multi-service applications

We now illustrate the unique features of our system using the tree-based application from Figure 5.7(c) and the shared-service one from Figure 5.7(d). We set the SLO to 500 ms.

Table 5.5: Prediction accuracy under increasing workload for shared-service application

	Add a cache at 7.5 req/s	Add a server at 7.5 req/s	Add a cache at 15 req/s	Add a server at 15 req/s
Serv. 1	400.2 ms (+1.5%)	493.6 ms (+2.1%)	428.1 ms (+2.2%)	481.3 ms (+0.9%)
Serv. 2	447.7 ms (+3.7%)	469.7 ms (+1.1%)	378.6 ms (-1.3%)	409.1 ms (+2.3%)
Serv. 3	465.3 ms (+1.3%)	489.1 ms (+1.8%)	452.7 ms (+0.8%)	473.4 ms (-0.7%)
Serv. 4	**342.3 ms (+3.9%)**	375.2 ms (+4.8%)	**295.7 ms (+2.7%)**	413.9 ms (+2.1%)

5.2.4.1 Provisioning under varying load intensity

Figure 5.9 shows the response time of the two applications when their request rates vary. Figure 5.9(a) depicts the test scenario: the workload first increases from 2 req/s to 22 req/s, then decreases back to 2 req/s.

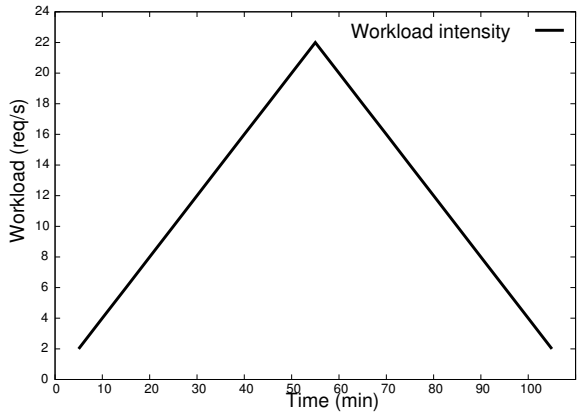
Figure 5.9(b) shows the performance of the tree-based application. Our system adds resources twice at 10 req/s and 18 req/s, adding a cache to service 7 then an application server to service 5. When the workload decreases, opposite decisions are taken at 16 req/s and 8 req/s. Figure 5.9(c) shows similar results for the shared-service application.

For all reconfigurations proposed by the provisioning system, we also verify the decisions by measuring the end-to-end response time of all other possible adaptations. Tables 5.4 and 5.5 show the prediction accuracy under increasing workload for the two applications at their respective adaptation points. In all cases the predictions remain very accurate and allow one to make the optimal provisioning decision. Similar accuracy is also obtained when decreasing the workload.

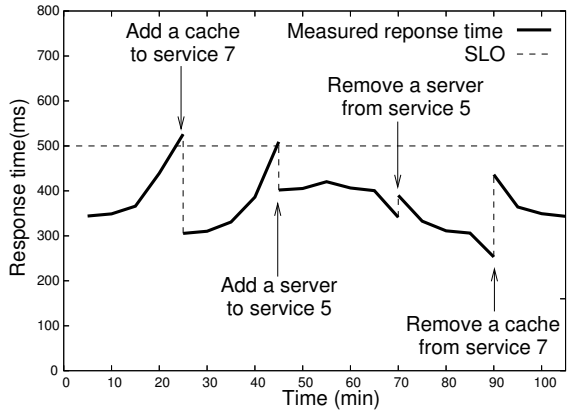
These results show that our provisioning system can correctly identify the most bottlenecked service within entire tree-based or shared-service applications when their SLO targets are violated. Meanwhile, our system can also save resource usage by removing resources from the least affected service while remaining within the SLO.

5.2.4.2 Provisioning under varying load distribution

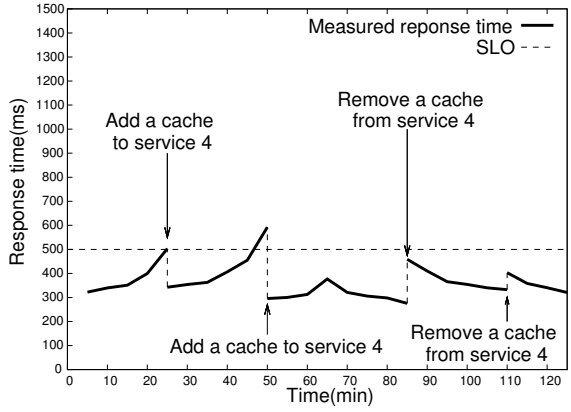
We now turn to more subtle cases where the front-end's request rate remains stable but internal parameters such as the invocation count from one service to another changes over time. The relative utilization of assigned resources may thus change over time. Figure 5.10(a) depicts the scenario for the tree-based application: the



(a) Experiment scenario



(b) Tree-based application



(c) Shared-service application

Figure 5.9: Resource provisioning under varying load intensity

workloads of services 2 and 3 first increase at the same rate. At time 35, the workload of service 3 drops by a factor 10, while service 2 maintains the same increase rate. We apply a similar scenario to services 2 and 3 of the shared-service application. In these experiments, we set the performance improvement threshold before shifting resources to 30%.

Figure 5.10(b) shows the behavior of the tree-based application. At time 25, our system proposes to add one cache to service 7 due to an SLO violation. At time 40, the response time of the whole application drops because less traffic is issued to service 3. Then the response time increases again. At time 70, the SLO is not violated but our system decides to shift one machine from service 7 to service 5 so as to gain 30% performance improvement with better resource organization. Figure 5.10(c) shows similar behavior for the shared-service application.

These results show that we can optimize resource organization without retrieving extra resources by identifying potential improvements of resource utilization.

5.2.4.3 Provisioning with varying load locality

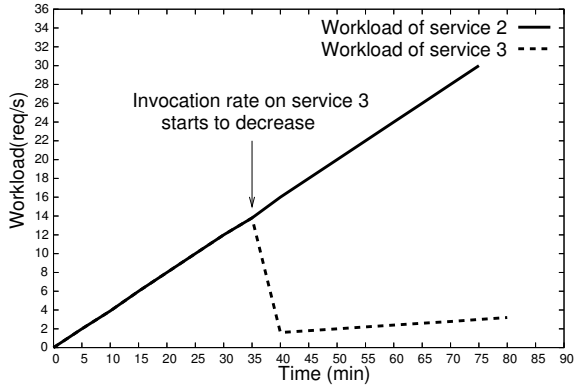
Another subtle form of change in workload is a variation of workload locality. Here, the potential performance of a cache varies over time. We define the locality as the hit rate for a cache holding 10,000 objects. Figure 5.11(a) depicts the evaluated scenario for the tree-based application: we first increase the workload until time 25 when the end-to-end response time violates the SLO target. Immediately after reconfiguration, we start changing the locality of service 3. We apply a similar scenario to the shared-service application.

Figure 5.11(b) shows that the tree-based application first adds one cache to service 7 at time 25. When the locality of service 3 changes, our system shifts the cache from service 7 to become a cache in service 1, such that the end-to-end response time improves by 35%. Figure 5.11(c) shows similar results for the shared-service application.

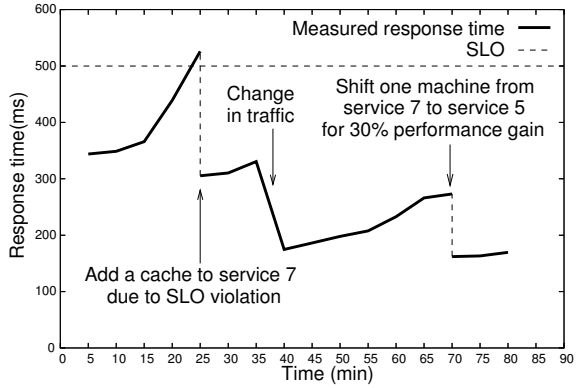
These results show that we can reorganize the cache assignment within a whole application to adapt to changes in traffic locality and improve application performance.

5.3 Conclusion

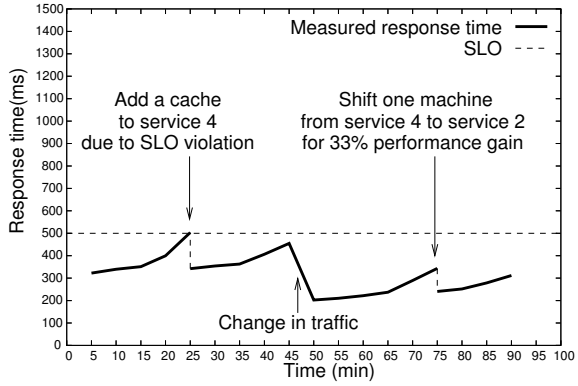
Most Web resource provisioning approaches rely on a single analytical queuing model to capture the application's performance features. However, applying such approaches to multi-service Web applications is a challenge due to complex service relationships and the cascading effects of caching. This chapter takes a different stand and demonstrates that provisioning resources for multi-service applica-



(a) Experiment scenario

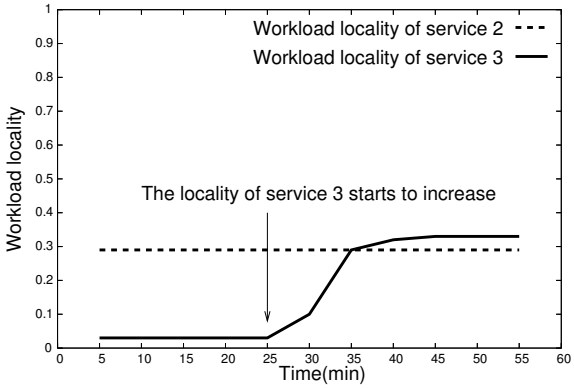


(b) Tree-based application

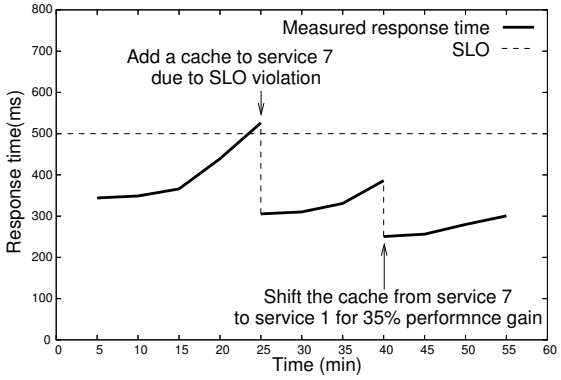


(c) Shared-service application

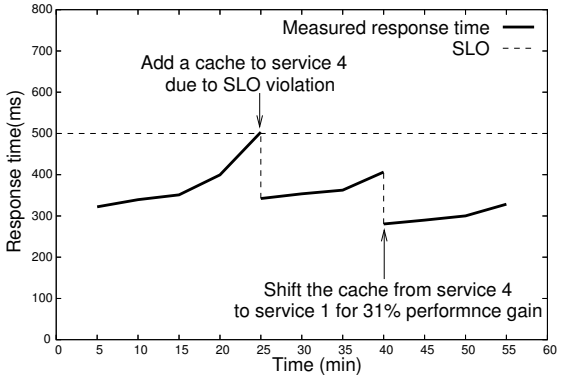
Figure 5.10: Resource provisioning under varying load distribution



(a) Experiment scenario



(b) Tree-based application



(c) Shared-service application

Figure 5.11: Resource provisioning under varying load locality

tions can be achieved in a decentralized way where each service is autonomously responsible for its own provisioning.

We propose to give an SLO only to the front-end service. All other services collaboratively negotiate their future performance objectives with each other to make provisioning decisions. Resource provisioning is based on "what-if analysis" where each service continuously reports its performance promises if it was assigned more/less resources, or if it received more/less traffic. The negotiation process occurs recursively between levels of the whole invocation graph. The root node is responsible for selecting which service(s) to provision so as to maintain the front-end service's SLO and maximize resource utilization. We show that this approach is more economical than its competitors as it can guarantee performance using fewer resources than them.

Note that in this chapter, we assumed that the hosting environment is fully homogeneous, such as in a cluster computer. However, cloud computing is emerging as a new hosting environment for Web applications. Cloud computing creates new opportunities for this type of applications, but also new challenges. In particular, cloud computing environments are largely heterogeneous in performance. In order to guarantee Web application performance in clouds, we must face this new challenge and adapt current resource provisioning algorithms to handle this performance heterogeneity. We address this issue in the next chapter.

Chapter 6

Resource provisioning in Cloud environments

Cloud computing is an attractive platform to host Web applications. Besides the advantages of outsourcing machine ownership and system management, Clouds offer the possibility to dynamically provision resources according to an application's workload and to pay only for the resources that are actually being used.

However, dynamic resource provisioning for Web applications in the Cloud faces two important challenges. First, Web applications are not monolithic. For instance, the two-tier architecture is widely used in Web application design with one application server tier and one database tier. As we have seen, large-scale Web applications employ multi-service architecture for improved scalability. Second, computing resources in the Cloud are largely heterogeneous across various virtual instance types as well as across multiple instances of a single type. On the other hand, we also observed that the performance of individual instances is stable over time, which provides a possible foundation for resource provisioning.

Efficient dynamic resource provisioning under these two challenges is very difficult. To provision a Web application dynamically and efficiently, we need to predict the performance the application would have if it was given a new machine. However, because of resource heterogeneity it is impossible to predict the performance profile of a new machine instance at the time we request it from the Cloud. We therefore cannot accurately predict the performance the application would have if it was using this new machine in one of its services. It is therefore necessary to profile the performance of each new machine once it has started, before deciding how we can make the best use of it.

One simple profiling method would consist of sequentially adding the new machine instance to each service of the application and measure the performance gain it can provide there. However, this approach is extremely inefficient and

time-consuming as profiling requires lots of time. For instance, adding a machine instance to a data service that runs a database may cost tens of minutes or more, which is not acceptable for dynamic resource provisioning.

In this chapter we show how one can efficiently profile new machines using real application workloads to achieve accurate performance prediction in the heterogeneous Cloud. By studying the correlation of demands that different services put on the same machine, we can derive the performance that a given service would have on a new machine instance, without needing to actually run this particular service on this particular machine instance. This per-service, per-instance performance prediction is crucial for making two important decisions. First, it allows us to balance the request load between multiple heterogeneous instances running the same service so as to use each machine instance according to its capabilities. Second, when the application needs to expand its capacity it allows us to correctly select which service of the application should be reprovisioned with a newly obtained instance.

We evaluate our provisioning algorithm in the Amazon EC2 platform. We first demonstrate the importance of adaptive load balancing in Cloud to achieve homogeneous performance from heterogeneous instances. We then use our performance prediction algorithm to drive the dynamic resource provisioning of the TPC-W benchmark. We show that our system effectively provisions TPC-W in a heterogeneous Cloud and achieves higher throughput compared to current provisioning techniques.

For simplicity, in this chapter we focus on the resource provisioning of multi-tier Web applications. We however believe that one can extend our approach to provision multi-service Web applications by integrating with the resource provisioning techniques discussed in Chapter 5.

6.1 System design

Dynamic resource provisioning for Web applications in the Cloud requires one to predict the performance that heterogeneous machine instances would have when executing a variety of tiers which all have different demands for the machine instances. This performance prediction allows us to choose which tier(s) should ideally benefit from this instance for optimal performance gains of this entire application.

6.1.1 Solution outline

We address the problem in four steps. First, when using multiple heterogeneous machines to run a single tier, one must carefully balance the load between them

to use each machine according to its capacity such that each provisioned instance features with equal response time. We discuss Web application hosting techniques and load balancing in section 6.1.2.

Second, we need to measure the individual performance profile of new machine instances for running specific application tiers. Benchmarking a machine instance for one tier does not generate a single measurement value, but an estimation of the response time as a function of the request rate. Profiling a tier in a machine requires some care when the tier is already used in production: we need to measure the response time of the tier under a number of specific request rates, but at the same time we must be careful so that the instance does not violate the application's SLO. We discuss profiling techniques in section 6.1.3.

Third, every time the application needs to provision a new machine instance, it is very inefficient to successively profile each of the application tiers on the new instance. Instead, we calibrate the respective hardware demands of different tiers of the Web application using a single 'calibration' machine instance. We also include two synthetic reference applications in the calibration. After this step, each new instance is benchmarked using the reference applications only. Thanks to the initial calibration, we can predict the performance that this particular machine instance would have if it was executing any tier of the real Web application. We discuss performance prediction in section 6.1.4.

Finally, knowing the performance that a new machine instance would have if we added it to any tier of the application, we can choose the tier where it would generate the greatest performance improvement for the overall application. We choose the targeted tier by modeling the whole Web application as a queueing network where each tier acts as a separate queue. We discuss the queueing model for provisioning tiers in section 6.1.5.

6.1.2 Web application hosting

Figure 6.1 shows the typical hosting architecture of a single tier of the Web application. The provisioned virtual instances m_1, m_2 and m_3 host either the replicated application code if this is an application server tier, or a database replica if this is a database tier. As the performance of provisioned instances largely differs from each other, it would be a bad idea to address the same request rate to all instances. A much better option is to carefully control the respective load of each instance such that they all exhibit the same response time. In this scenario, fast machine instances get to process more requests than slower ones.

We control the workload by deploying a custom load balancer in front of the provisioned instances. To guarantee backend instances to serve with equal response time, the load balancer calculates the weighted workload distribution according to their performance profiles by solving the following set of equations:

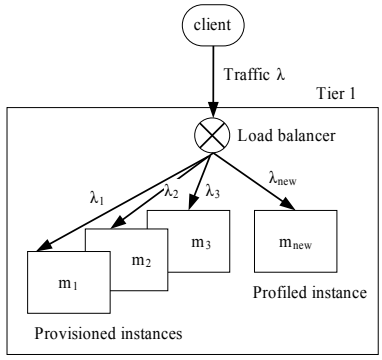


Figure 6.1: Web application hosting in the Cloud

$$\begin{cases} \lambda = \lambda_1 + \cdots + \lambda_n \\ r = perf(instance_1, \lambda_1) \\ \cdots \\ r = perf(instance_n, \lambda_n) \end{cases} \tag{6.1}$$

where λ is the total request rate seen by the load balancer, $\lambda_1, \ldots, \lambda_n$ are the request rates addressed to each provisioned instance respectively and r is the uniform response time of all provisioned instances. The $perf()$ functions are typically defined as a set of measured points, with linear interpolation between each consecutive pair of points.

This set of *n+1* equations can be easily solved to find the values of r, λ_1, ..., λ_n. The load balancer uses these values as weights of its weighted Round-Robin strategy.

When adding a new instance m_{new} into this tier, the load balancer thus needs to know the performance profile of this new instance such that it can balance the workload accordingly. This is the goal of instance profiling that we discuss in next.

6.1.3 Online profiling

Coming up with a machine instance's own performance profile when provisioning a given tier can be done in two different ways: either we measure the actual profile using real request traffic, or we derive the profile from other measured profiles. This section discusses the former.

Profiling a machine instance with a given tier workload consists in deploying the tier service on the machine instance, then addressing traffic with various load intensities to measure the corresponding response times.

We approximate the actual profile of a new instance by measuring performance at carefully selected workload intensities, and using linear interpolation between

each consecutive pair of measured points. The output of the online profiling of a new instance is therefore a set of n linear functions which cover consecutive workload ranges as follows:

$$r_i = a_i \times \lambda_i + b_i \qquad (1 \leq i \leq n) \tag{6.2}$$

where n is the total number of the consecutive workload ranges, and r, λ, a and b respectively represent average response time, request rate and linear parameters within the workload range i.

Generating a performance profile for a synthetic application is relatively easy: one only needs to deploy the synthetic application on the tested machine, and use a separate machine instance to generate a standardized workload and measure the tested instance's performance profile.

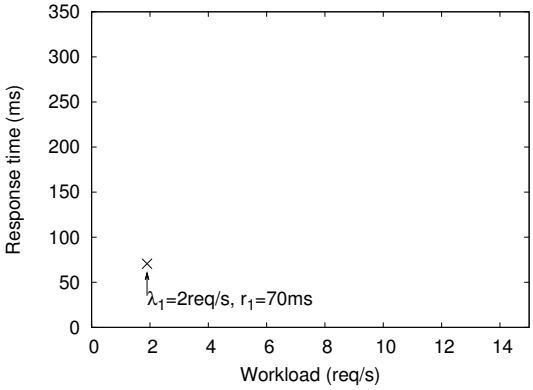
Generating a similar performance profile for a real tier of the Web application is harder. We want to address traffic that is as realistic as possible to increase the accuracy of the performance profile. Metrics which make a traffic workload realistic include the respective proportions of simple and complex requests, read/write ratio, popularity distribution of different data items, and so on. All these properties have a significant impact on performance. Instead of trying to synthesize a realistic workload, we prefer to provision the new instance in the tier to profile, and address real live traffic to it (which is realistic by definition).

Profiling a machine instance using live traffic however requires caution. First, we must make sure that profiling this instance will not create an SLO violation for the end users whose requests are processed by the profiled machine instance. For instance, one could simply replace one of the current instances used in the tier with the instance to profile. However, if the new instance is slower than the previous one, the application may violate its SLO. Second, we want to test specific workload intensities regardless of the actual workload received by the tier at the time of the profiling. Profiling a live tier therefore requires careful load balancing where we control the request rate addressed to the profiled instance.

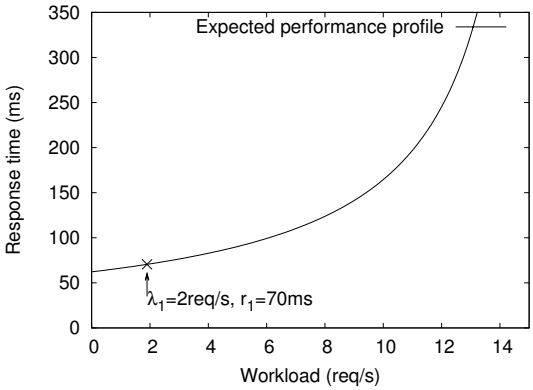
We first need a rough estimation of the variety of performance profiles from one instance to another. Such variety is specific to one Cloud provider, as it largely depends on the consolidation strategies and virtualized performance isolation that the Cloud implements. We calculate the performance variety rate N as follows.

$$N = \frac{T_{max}}{T_{min}} \tag{6.3}$$

where T_{max} and T_{min} respectively represent the throughput of the fastest and slowest instances in the Cloud when running a given Web application. We set an SLO defining the maximum response time to this Web application. We measure the



(a) First measurement point selected such that the instance will not violate its SLO



(b) First estimation of the instance's profile thanks to queueing theory

Figure 6.2: Online profiling process – first measurement and estimation

throughput of this Web application when it violates the SLO. The tested application exhibits either a CPU-intensive or an I/O-intensive workload for estimating the CPU and IO performance variety separately. For instance, in Chapter 3 we observed $N_{CPU} \approx 4$ for CPU-intensive tiers such as application servers and $N_{I/O} \approx 2$ for I/O-intensive tiers such as database servers in the Amazon EC2 Cloud. Similarly, in Rackspace we observed $N_{CPU} \approx 1.5$ and $N_{I/O} \approx 4$. In a new Cloud platform, one would need to sample a sufficient number of instances to evaluate these numbers.

Second, we carefully choose different workload intensities to address to the new machine. One needs to choose the key performance points (λ, r) that represent significant features of the performance profile. For instance, the performance

profile of a new instance under low workload can be approximated as a constant value regardless of the load. We ideally want a number of points varying from underload to overload situations, preferably at the inflection points and close to the SLO. The accuracy of the approximated curve increases with the number of measured points. However, this also increases the profiling time.

Figure 6.2 and Figure 6.3 illustrates our strategy to select the request rate for profiling the new instance. We first address the new instance with request rate $\lambda_1 = \frac{\lambda_{max}}{N}$, where λ_{max} is the current request rate of the fastest instance currently used in this tier. Assuming our estimation of performance variety N is correct, the profiled instance cannot violate the SLO even if it happens to be very slow. This gives us the first performance point (λ_1, r_1) as illustrated in Figure 6.2(a)

Using this first measurement, we can use queueing theory to generate a first estimate of the entire performance profile of this instance. If we model the tier as an M/M/n queue, then the instance's service time can be computed as:

$$s = \frac{r_1}{1 + \frac{\lambda_1 \times r_1}{n}} \tag{6.4}$$

where n is the number of CPU cores of this machine (as announced by the Cloud provider). The service time is the response time of the instance under low workload where the effects of request concurrency are negligible. It indicates the capability of the instance to serve incoming requests. We can then use this service time to derive a first performance profile of the new instance as follows:

$$r(\lambda) = \frac{s}{1 - \frac{\lambda \times s}{n}} \tag{6.5}$$

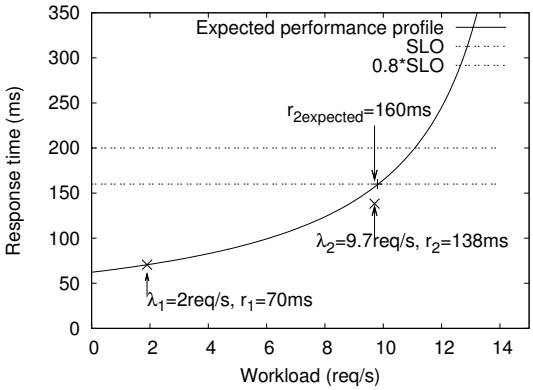
Figure 6.2(b) shows the first performance profile of the new instance derived based on the calculated service time. One should however note that this profile built out of a single performance value is very approximate. For a more precise profile, one needs to measure more data points.

Using this profile, we can now calculate a second workload intensity which should bring the instance close to the SLO. We select an expected response time r_2, then derive the workload intensity which should produce this response time.

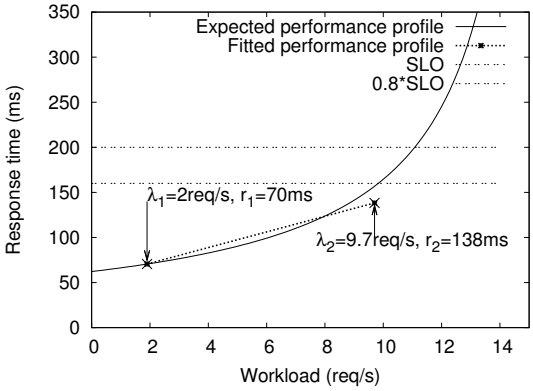
$$r_2^{expected} = 0.8 \times SLO \tag{6.6}$$

$$\lambda_2 = \frac{n \times (r_2^{expected} - s)}{r_2^{expected} \times s} \tag{6.7}$$

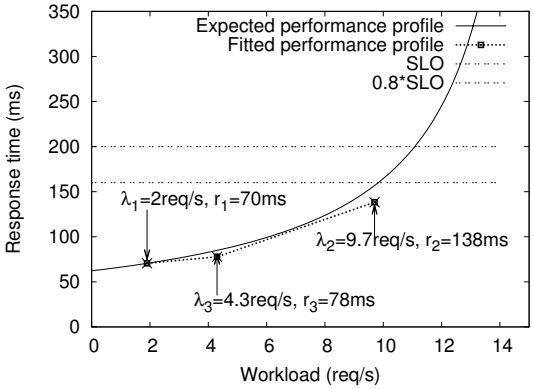
Here we set the target response time to 80% of the SLO to avoid violating the SLO of the profiled instance even though the initial performance profile will feature relatively large error margins. We can then address this workload intensity



(a) Selection of a second measurement point close to the SLO



(b) Fit performance profile of the new instance



(c) Correction of the performance profile of the new instance

Figure 6.3: Online profiling process – performance fitting and correction

to the new instance and measure its real performance value (λ_2, r_2). As shown in Figure 6.3(a), the real performance of the second point is somewhat different from the expected 80% of the SLO.

We apply linear regression between the two measured points (λ_1, r_1), (λ_2, r_2) and get the fitted performance profile of the new instance as shown in Figure 6.3(b). We then let the load balancer calculate the weighted workload distribution between the provisioned instance and the new one.

By addressing the weighted workload intensities to the two instances, we can measure the real response time of the new instance. However, as shown in Figure 6.3(c), the real performance of the new instance differs slightly from the expected one in Figure 6.3(b) due to the approximation error of its initial profile. We then correct the performance profile of the new instance by interpolating the third performance point(λ_3, r_3). We show that the above strategy is effective to profile heterogeneous virtual machines and provision single services in Section 6.2.

Although expressing performance profiles as a function of request rate is useful for load balancing, for performance prediction we need to express performance profiles as a function of CPU utilization (for application server tiers) or I/O utilization (for database server tiers). When profiling a machine instance, we also measure the relevant metrics of resource utilization, and use the exact same technique to build performance profiles that are suitable for performance prediction.

6.1.4 Performance prediction

To efficiently select the tier in which a new instance will be most valuable to the application as a whole, we first need to know the performance profile of this instance when running each of the application's tiers. A naive approach would be to successively measure this profile with each tier one by one before taking a decision. However, this strategy would be too slow to be of any practical use. Indeed, profiling a new machine with a real application tier requires to first replicate the hosted service to the new machine. For instance, when profiling a new machine for a database tier, one needs to first replicate the entire database to the new machine before starting the profiling process. Replicating a medium-sized database can easily take tens of minutes, and this duration increases linearly with the database size. We therefore need to be able to quickly *predict* the performance profiles, without needing to actually replicate the database.

We found that the most characteristic feature of a virtual instance to predict the performance profile of a given tier in this instance is its resource utilization. Although the absolute response time of two different tiers of the same category (such as application server or database server) in the same machine under the same CPU or I/O utilization are not identical, they are highly correlated.

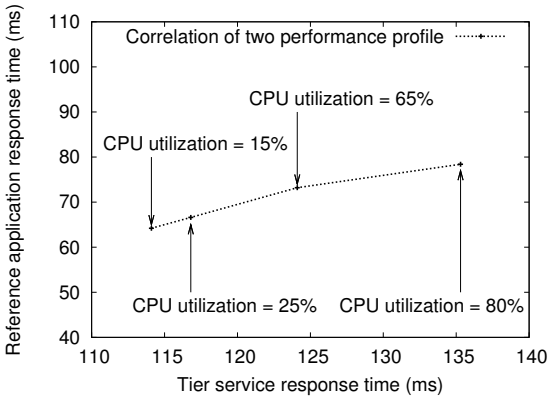


Figure 6.4: Performance correlation between reference application and tier service

We illustrate this in Figure 6.4. Each point in this graph represents the response times of two different application server tiers running in the same machine instance, and having the same CPU utilization (respectively 15%, 25%, 65% and 80%). The request rates necessary to reach a given CPU utilization varies from one application to the next. We however observe that the points form an almost perfect straight line. This allows us to derive new performance profiles from already known ones. The same observation is also true for database server tiers, taking the disk I/O bandwidth consumption as the resource utilization metric.

Given the response time and resource utilization of one tier in a given machine instance, we can infer the response time of the second tier in the same machine instance under the same resource utilization. Figure 6.5 illustrates the input and output of this prediction: we predict the performance of tier 1 and tier 2 on a new machine by correlating their performance and the reference application performance on a calibration machine.

First, we need to measure the application-specific demands of each tier of the application. This has to be done only once per application. This profiling should be done on a single calibration machine, which can be any particular virtual instance in the Cloud. To predict the performance of any particular tier on a new instance quickly, we also benchmark the calibration machine using two synthetic reference applications which respectively exhibit CPU-intensive features characteristic of application servers, and I/O-intensive features characteristic of database servers. The Ref_{CPU} application receives customer names and generates detailed personal information through CPU-intensive XML transformation. The $Ref_{I/O}$ application searches for items related to a customer's previously ordered items from a large set of items. The operations of the reference applications introduce typical CPU-

Input:
$perf(machine_{calibration},\ app_{ref}) = f(load)$
$perf(machine_{calibration},\ app_{tier1}) = f(load)$
$perf(machine_{calibration},\ app_{tier2}) = f(load)$
$perf(machine_{new},\ app_{ref}) = f(load)$

Output:
$perf(machine_{new},\ app_{tier1}) = f(load)$
$perf(machine_{new},\ app_{tier2}) = f(load)$

Figure 6.5: Input and output of the performance profile prediction

intensive and disk I/O-intensive workloads. The reference applications can be deployed very quickly on any new machine instance, for example by including it to the operating system image loaded by the virtual machine instances. We use Ref_{CPU} as a reference point to predict the performance profiles of application server tiers, and $Ref_{I/O}$ as a reference point to predict the performance profiles of database tiers.

Profiling the same calibration machine instance with one of the Web application's tiers and the corresponding reference application allows us to learn the relationship between the demands that these two applications put on the hardware:

$$perf(app_{tier}, utilization) = \alpha \times perf(app_{ref}, utilization) + \beta$$

The same relationship between the response times of the two applications, captured by the values of α and β, remains true on other machine instances. Knowing the performance profile of the reference application on a newly obtained virtual machine instance from the cloud, we can thus derive the predicted performance profile of *tier1* on the new instance, even though we never even installed this particular tier on this particular instance.

6.1.5 Resource provisioning

When provisioning a multi-tier Web application, upon a violation of the service-level objective, one needs to decide which tier to re-provision within the whole application. Once a new instance is acquired and profiled, one needs to perform a simple what-if analysis to predict the performance improvement that the whole application would observe if this instance was added in one of the tiers. For simplicity, in this paper we apply our Cloud instance profiling methods to simple two-tier Web applications only. Other performance models of composite Web applications can be used to extend this work to more complex setups [Urgaonkar et al., 2008].

The response time of a two-tier Web application can be computed as follows.

$$R_{app} = R_1 + N_{1,2} \times R_2 \tag{6.8}$$

where R_{app} is the response time of the whole application, R_1, R_2 are response time of the application server tier and the database tier respectively, $N_{1,2}$ is the request ratio that equals to the request number seen by the second (database) tier caused by one request from the first (application server) tier.

Given the performance profiles of the new instance for each of the application's tiers, we can issue a simple "what-if" analysis: we first use the performance profiles to compute the new application performance if the new instance was added to the first tier, then if it was added to the second tier. The best usage of the new instance is defined as the one which maximizes the application's performance.

6.2 Evaluation

In this section we evaluate the effectiveness and efficiency of our resource provisioning algorithm for provisioning Web applications on the Amazon EC2 platform.

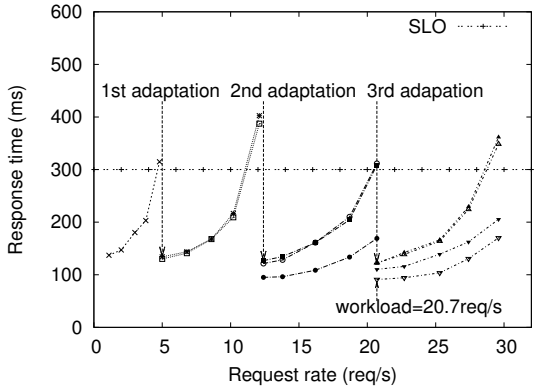
6.2.1 Experiment setup

The bulk of our system implementation lies in our custom layer-4 load balancer. In addition to distributing requests to backend servers, the load balancer also profiles new machines when we obtain them from the cloud. By deploying the load balancer in front of the tiers of Web applications, our system can provision Web applications over heterogeneous instances in the Cloud.

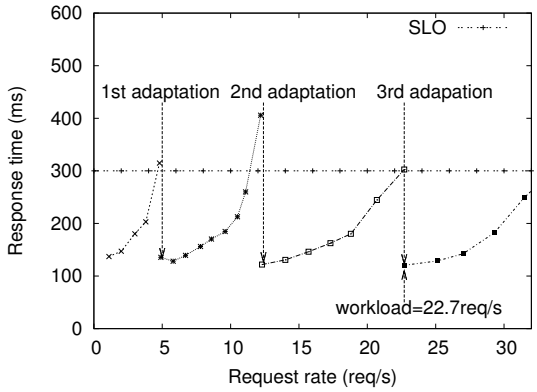
We evaluate our resource provisioning algorithm using three Web applications. The first two are the reference applications Ref_{CPU} and $Ref_{I/O}$. The last one is TPC-W Web application benchmark. This benchmark is structured as a two-tiered application which models an online bookshop like Amazon.com [TPC-W, 2011]. We run all our experiments on the Amazon EC2 platform using small instances.

6.2.2 Importance of adaptive load balancing

We first demonstrate the importance of adaptive load balancing in the Cloud using Ref_{CPU} and $Ref_{I/O}$. We deploy each application on a single machine instance, and increase the workload gradually. We set the SLO of the response time of Ref_{CPU} and $Ref_{I/O}$ to 300 ms and 500 ms respectively. We run each experiment using two different setups. First, we use Amazon's Elastic Load Balancer to distribute the traffic between the instances, and Amazon's AutoScale to provision new virtual machine instances when the SLO is violated. Second, we run the same experiment using the exact same instances with our system. Both applications are



(a) Using Amazon's Elastic Load Balancer

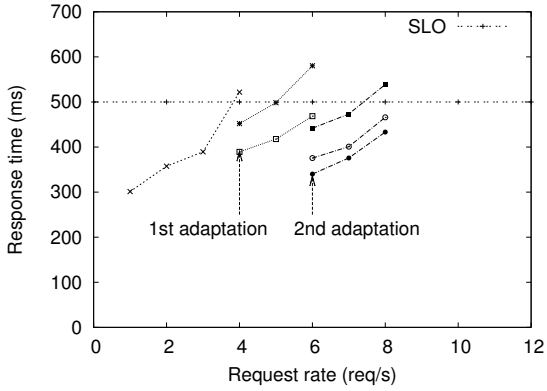


(b) Using our adaptive load balancer

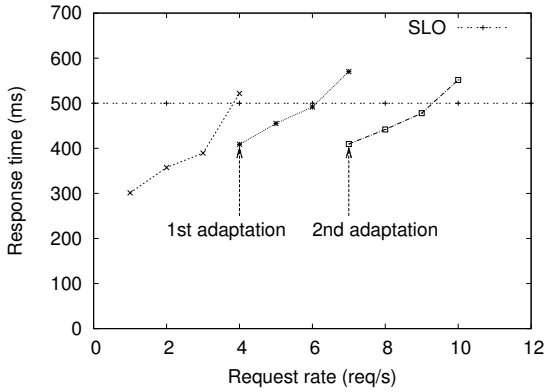
Figure 6.6: Provisioning Ref_{CPU} under increasing workload

single-tiered, so here we exercise only the capability of load balancing to adapt to heterogeneous resources.

Figure 6.6 shows the response time per machine instance running the Ref_{CPU} application. When using the Elastic Load Balancer (ELB), at 5 req/s the system violates the SLO and therefore provisions a new instance. By coincidence, the second instance has a performance profile very close to the first one so they exhibit extremely similar performance. However, after the second and third adaptation we see that different instances exhibit different performance. On the other hand, our system balances the workload such that all instances always exhibit the same performance. This has important consequences in terms of resource usage: when using ELB, one of the application instances violates its SLO at 20.7 req/s, triggering a request for a fourth instance. When using our system, the third instance (a



(a) Using Elastic Load Balancer



(b) Using our adaptive load balancer

Figure 6.7: Provisioning $Ref_{I/O}$ under increasing workload

very fast one) is given a higher workload than the others so the system requires a fourth instance only above 22.7 req/s.

Figure 6.7 shows similar results for the $Ref_{I/O}$ application. Here as well, our system balances traffic between instances such that they exhibit identical performance, whereas ELB creates significant performance differences between the instances. Our system can sustain up to 9 req/s when using three instances, while ELB can sustain only 7 req/s.

These results show that one should employ adaptive load balancing to correctly assign weights to forwarding instances when distributing traffics in the Cloud. By doing so, one can achieve homogeneous performance from heterogeneous instances and make more efficient usage of these instances.

6.2.3 Effectiveness of Performance Prediction and Resource Provisioning

We now demonstrate the effectiveness of our system to provision multi-tier Web applications. In this scenario, in addition to using our load balancer, we also need to predict the performance that each new machine would have if it was added to the application server or database server tiers to decide which tier a new instance should be assigned to. The Amazon cloud does not have standard automatic mechanisms for driving such choices so we do not compare our approach with Amazon's AutoScale.

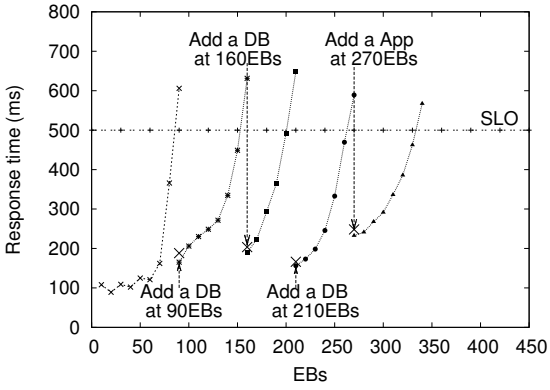
We use our system to provision the TPC-W e-commerce benchmark using the "shopping mix" workload. This standard workload generates 80% of read-only interactions, and 20% of read-write interactions. We set the SLO of the response time of TPC-W to be 500 ms. We increase the workload by creating corresponding numbers of Emulated Browsers (EBs). Each EB simulates a single user who browses the application. Whenever an EB leaves the application, a new EB is automatically create to maintain a constant load.

When the overall response time of the application violates the SLO, we request a new instance from the Cloud and profile it using the reference application. Thanks to the performance correlations between the tiers of TPC-W and the reference application, we use the performance profile of the new instance to predict the performance of any tier of TPC-W as if it was using the new instance. Finally, we compare the performance gains if the new instance was assigned to different tiers of TPC-W and select the tier which gives the most performance benefit. We run the entire experiment twice: our provisioning system takes different decisions depending on the characteristics of the machine instances it gets from the Cloud.

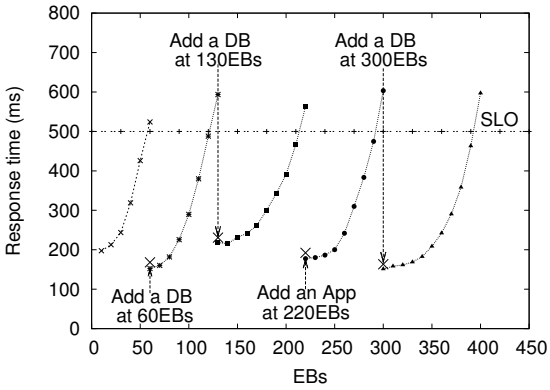
Figures 6.8(a) illustrates the response time of TPC-W in the first run of the experiment. The application violates its SLO around a workload of 90 EBs. We request a new instance from the Cloud, profile it, and predict that it would be most useful if it was assigned to the database tier. When we push the workload further, it adds another database server at 160 EBs, then yet another database server at 210 EBs, then finally an application server at 270 EBs.

Figure 6.8(b) shows that, if we run the exact same experiment a second time, the machine instances we obtain from the Cloud have different performances. This leads the resource provisioning to take different decisions. It adds two database servers respectively at 60 and 130 EBs, then an application server at 220 EBs, then another database server at 300 EBs.

We can see here that SLO violations occur at different workloads, depending on the performance of the machine instances running the application. We also see that our resource provisioning effectively distinguishes different performance profiles, and takes provisioning decisions accordingly. In particular, at the third



(a) First group



(b) Second group

Figure 6.8: Provisioning TPC-W under increasing workload

adaptation, the first run decides to use the new machine instance as a database server while the second run decides to use its own new machine instance as an application server.

At each adaptation point, the resource provisioning system issues two predictions: it predicts what the new response time of the overall application would be if we assigned the new machine instance to be an application server or a database server. At each adaptation point we also tested the accuracy of these two predictions by deploying each of the two tiers in the new instances and measuring the actual application performance. Tables 6.1 and 6.2 show the measured and predicted response times of the whole application at each adaptation point. We can see that all predictions remain within 14% of the measured response times. This level of accuracy is sufficient to take correct provisioning decisions: in this set of

Table 6.1: Prediction accuracy during the first experiment run

	Adapt at 90 EBs			Adapt at 160 EBs		
	Real	Predicted	Error	Real	Predicted	Error
Provision AS tier	554.6 ms	596.7 ms	+7.6%	578.3 ms	625.1 ms	+8.1%
Provision DB tier	**165.4 ms**	**188.1 ms**	+13.7%	**189.7 ms**	**203.4 ms**	+7.2%
	Adapt at 210 EBs			Adapt at 270 EBs		
	Real	Predicted	Error	Real	Predicted	Error
Provision AS tier	458.3 ms	490.1 ms	+6.9%	**232.5 ms**	**248.3 ms**	+6.8%
Provision DB tier	**156.2 ms**	**166.4 ms**	+6.5%	313.4 ms	329.1 ms	+5.0%

Table 6.2: Prediction accuracy during the second experiment run

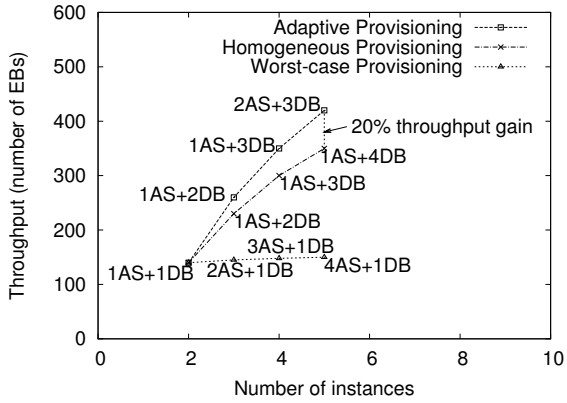
	Adapt at 60 EBs			Adapt at 130 EBs		
	Real	Predicted	Error	Real	Predicted	Error
Provision AS tier	511.7 ms	567.3 ms	+10.9%	427.9 ms	453.7 ms	+6.0%
Provision DB tier	**152.4 ms**	**168.2 ms**	+10.4%	**218.2 ms**	**230.7 ms**	+5.7%
	Adapt at 220 EBs			Adapt at 300 EBs		
	Real	Predicted	Error	Real	Predicted	Error
Provision AS tier	**177.5 ms**	**192.3 ms**	+6.9%	541.9 ms	579.2 ms	+6.9%
Provision DB tier	281.4 ms	302.7 ms	+7.6%	**151.2 ms**	**163.2 ms**	+7.9%

experiments, the provisioning always identifies the best use it can make of the new machine instance it received (written in bold text in the table).

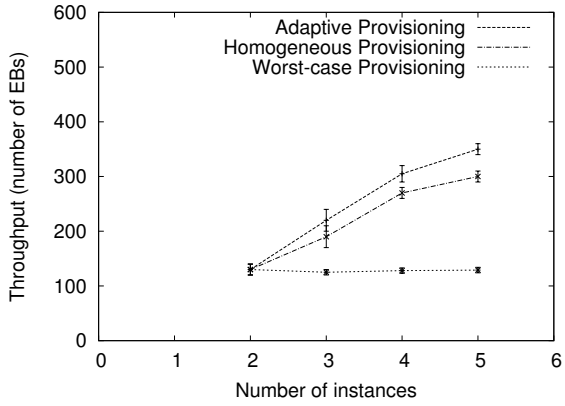
6.2.4 Comparison with other provision techniques

So far we showed the effectiveness of our system to provisioning TPC-W on EC2 by assigning heterogeneous instances to the tier where it gives maximum performance gain. We now demonstrate that our system can improve the throughput of TPC-W running on EC2 compared with two other provisioning techniques: "Homogeneous Provisioning" and "Worst-case Provisioning".

"Homogeneous Provisioning" provisions instances assuming that the performance of these instances is homogeneous. "Homogeneous Provisioning" first profiles the performance of the first two virtual instances hosting TPC-W. At each adaptation, "Homogeneous Provisioning" predicts the performance gains of new instances at each tier using the initial performance profiles, and assigns a new instance to the tier which receives maximum performance gain. "Homogeneous Provisioning" dispatches requests between instances using the round-robin policy. "Worst-case Provisioning" employs our algorithm to first figure out the tier to which a new instance should be assigned. However, "Worst-case Provisioning" systematically adopts the worst possible option. For instance, "Worst-case Provi-



(a) Throughput comparison of single round



(b) Statistical comparison of throughput over multiple rounds

Figure 6.9: Throughput comparison of three provisioning techniques

sioning" assigns a new instance to the application server tier if our system decides to assign this instance to the database tier. "Worst-case Provisioning" employs the same load balancing in our system. For comparison, we name our system as "Adaptive Provisioning".

We first use the three techniques to provision TPC-W on EC2 with increasing workload separately. We set the SLO to 500 ms and measure the maximum throughput that a system configuration can sustain before violating the SLO. We also record the instance configurations at each adaptation. Figure 6.9(a) shows the throughputs achieved by each provisioning technique during a single run of the system under increasing workload and the corresponding instance configurations at each adaptation. The three provisioning systems use the exact same instances in the same order so their respective performance can be compared.

"Worst-case Provisioning" decides to provision the application server tier at each adaptation and finally supports around 150 EBs with 5 instances. "Homogeneous Provisioning" and "Adaptive Provisioning" both decide to provision new instances to the database server tier at the first and second adaptation. However, they achieve different throughput at the first two adaptations. The throughput difference is caused by the different load balancing capability of adapting to heterogeneous instances used in each provision technique. At the third adaptation, "Adaptive Provisioning" decides to assign the new instance to the application server tier while "Homogeneous Provisioning" decides to assign the same new instance to the database server tier. After the third adaptation, "Adaptive Provisioning" supports around 420 EBs while "Homogeneous Provisioning" supports around 350 EBs. This represents a 20% gain in throughput.

We then run the same experiment 5 rounds, each with a different set of EC2 small instances. Within each round, we measure the throughput achieved by each provisioning technique using a certain number of instances. The throughputs achieved in different rounds are different due to the performance heterogeneity of small instances. Figure 6.9(b) shows the average and standard deviation of the throughput achieved by each provisioning technique across multiple rounds. As previously, the "Worst-case Provisioning" behaves as the statistical lower bound of the achievable throughput of TPC-W on EC2. When taking the first adaptation, "Adaptive Provisioning" and "Homogeneous Provisioning" behave similar in terms of achieved throughput. However, when taking more adaptations, "Adaptive Provisioning" supports 17% higher throughput than "Homogeneous Provisioning". This demonstrates that our system makes more efficient use of heterogeneous instances in Cloud and achieves higher throughput using the same resources.

6.3 Conclusion

Cloud computing provides Web application providers with an attracting paradigm to dynamically vary the number of resources used by their application according to the current workload. However, Cloud computing platforms also have important limitations. In particular, dynamic resource provisioning is made difficult by the fact that each virtual instance has its own individual performance characteristics. Standard resource provisioning techniques provided by Cloud platforms do not take this performance heterogeneity into account, and therefore end up wasting resources.

We demonstrated that taking performance heterogeneity into account in a resource provisioning system can be practical and bring significant resource savings. One must first capture the performance relationships between different tiers of an application. Second, when the application workload makes it necessary to provi-

sion a new instance, we can efficiently capture its own performance profile, and use this information to drive the resource provisioning decisions. It allows us to decide to which tier this new machine instance should be assigned and to adjust load balancing to make better use of the processing resources of each machine instance.

Chapter 7

Conclusion

Web application providers care about the performance of their applications. As noted earlier, server-side performance guarantees are often a primary concern from the perspective of both application providers and hosting providers. However, one needs to face the challenge of handling arbitrary levels of workload. Web workloads are unpredictable and fluctuating, which makes them difficult to handle efficiently. This thesis therefore addresses the question: *how to guarantee the server-side performance for Web applications in a cost-effective manner?*

An intuitive solution to address this challenge is to dynamically add or remove resources assigned to a Web application according to its workload. However, a key observation is that no one can guarantee application performance if the application architecture itself is not scalable. This thesis concludes that performance guarantees for Web applications require: (i) design principles to make Web applications scalable, and (ii) control techniques such as dynamic resource provisioning that allow one to provision scalable multi-service Web applications and exploit the on-demand resource usage model in clouds for cost-effective manner.

7.1 Research contributions

This thesis mainly contributes in two aspects: scalable Web application design and dynamic resource provisioning.

Scalable Web application design

A scalable Web application design is essential to guarantee performance. Instead of building Web applications along a traditional monolithic organization, one should decompose the application's business logic and data into separate services. We discussed several considerations for decomposing application data, such as constraints deriving from transactions and query rewriting. Constructing a Web application along a multi-service architecture can facilitate the use of various scal-

ing techniques to individual services according to their performance features. Consequently, one can create applications that scale naturally instead of having to face the difficult challenge of scaling a monolithic application. We showed that at least an order of magnitude scalability improvement can be gained by applying our approach.

Dynamic resource provisioning

Although a multi-service architecture enables elasticity and scalability improvements, we still face the issue of deciding which service(s) should be (de-)provisioned when traffic varies over time. We propose a decentralized approach to provision multi-service Web applications. This approach lets each service be responsible for its own performance prediction in case it would execute using different numbers of resources. Services collaboratively negotiate their performance objectives and find the one that would benefit the most (or lose the least) from such reconfigurations. We present the negotiation process within two typical multi-service architectures: tree based and directed acyclic graph based. Using our approach, one can guarantee Web application performance under various workload conditions, such as workload intensity change, workload locality change and workload mix change.

After demonstrating the effectiveness of our approach in a homogeneous computer cluster, we move to Cloud hosting environments. Two performance features of virtual machine instances in the Cloud are relevant from the point of view of resource provisioning: performance stability and performance homogeneity. We demonstrated that the performance of individual virtual machines is stable over time. However, the performance of multiple virtual machines of the same type is heterogeneous, which breaks the classical resource provisioning assumption about the homogeneity of hosting resources.

Performance heterogeneity in the Cloud requires adaptations to current resource provisioning techniques. To this end, we propose to profile each new virtual machine instance before predicting the effect it can have on the entire application. However, profiling a new virtual machine instance with real application workload is not practical. Instead, we present techniques to predict the future performance of the real application from measurements of a simple reference application. This allows one to quickly measure the individual performance profile of a newly acquired virtual instance. We propose to incorporate the profiling technique into the resource provisioning process in cloud environments. Our evaluations suggest that using this approach one can effectively guarantee Web application performance in a real Cloud environment.

7.2 Lessons learned

Providing performance guarantees for Web applications may initially look like an easy task. However, one cannot achieve this goal by applying ad hoc measures. Instead, performance guarantees for Web applications require a systematic solution ranging from offline design principles to online control mechanisms. Although this thesis focused on this systematic solution, we have also learned many other useful lessons along the way. In this section, we summarize the most relevant ones.

Understanding the performance behavior of Web application

To guarantee application performance, one must first fully understand the performance behavior of the target application. The performance behavior of a Web application is affected by many factors such as its workload, design principles, and hosting environment. One can take suitable actions only after understanding the performance impacts of these factors. For instance, a replication technique may be effective for guaranteeing Web application performance under a CPU-intensive workload. However, one cannot expect the same effect when applying this technique under a data-intensive workload. For a data-intensive workload, different data access patterns may lead to different actions to guarantee performance. The data denormalization technique proposed in this thesis is addressed to Web applications that organize their data along a relational schema. However, this technique is not a silver bullet for guaranteeing performance under all kinds of data-intensive workloads. Web applications that involve only simple data access patterns, such as key/value access, can benefit from NoSQL solutions for performance guarantees. NoSQL data stores, such as Bigtable [Chang et al., 2006], Dynamo [DeCandia et al., 2007], PNUTS [Cooper et al., 2008], Cassandra [Lakshman and Malik, 2010], and HBase [HBase, 2011], also differ in performance and scalability in different application scenarios. Having a detailed understanding of the Web application can help to take the right actions for performance guarantees.

Drawback vs. benefit of Cloud hosting

Cloud hosting is an attractive solution for medium- and small-size Web applications. Application owners can often not afford the cost and effort to build their own scalable hosting infrastructure. Meanwhile, the requirement of handling arbitrary levels of workload requires on-demand resource provisioning which fits well with the Cloud computing model. Web application owners can thus save costs on both hardware and human resource.

However, before moving to the Cloud, one must also understand the drawbacks of Cloud computing. One significant drawback is the inefficient resource usage caused by the performance heterogeneity of Cloud resources. Current Cloud provisioning services do not take this performance heterogeneity into account. Therefore, one may need to allocate unnecessary resources to make up for the

poor performance of a few virtual machine instances, while better performance effects could be achieved using the same instances by first understanding their performance profiles. One may also waste resources by assigning a new virtual machine instance to a service whose demands do not match the capabilities of this instance.

We hope that our observations on Cloud performance will attract the attention of Web application owners that are evaluating hosting solutions in the Cloud. One possible solution could be for Cloud providers to integrate techniques from Section 6.1.3 to their standard load balancing services. Doing so might further increase the attractiveness of Cloud technologies for Web application providers.

7.3 Future directions

This thesis presented several mechanisms and techniques to guarantee Web application server-side performance. Undoubtedly, there are a number of directions in which our research can be extended or complemented.

This thesis used average response time as the performance metric for Web applications. Although this metric is widely used in practice, percentiles of the response times are also a desirable metric to measure Web application performance [Menascé, 2002]. Web applications usually receive a mix of different requests which impose different workloads. Taking the distribution of response times into account can give Web application providers a detailed view of Web application performance. For applications which receive a broad mix of simple and complex requests, the distribution of the response times is more representative of the Web application performance than the average response time [Andreolini et al., 2004]. Extending our proposed techniques to guarantee percentiles of response times would be an important future direction to complement our work.

We observed that a decentralized resource provisioning approach is more effective than a centralized one to handle Web applications that expand in both scale and complexity. This thesis focused on multi-service Web applications where all the services belong to the same administrative domain. However, many real multi-service applications span multiple administrative domains. For example an eCommerce Web application may often invoke the payment service from a third-party eBanking service provider. A poor performance of the payment service would reflect in the overall performance of the eCommerce application. To handle such a case, our approach needs to be extended to address new issues deriving from this structure. Different administrative domains may be reluctant to disclose detailed information about their internal organization and the performance of their services. We however believe that decentralized resource provisioning provides a good basis to support such difficult scenarios.

We focused our efforts on Web applications that employ relational databases to store their data. However, a new family of large-scale Web applications tends to employ NoSQL data stores to provide scalable data access. Storage systems, such as Bigtable [Chang et al., 2006], Dynamo [DeCandia et al., 2007], PNUTS [Cooper et al., 2008], Cassandra [Lakshman and Malik, 2010], and HBase [HBase, 2011], are often used in Web applications that require simple key/-value data access pattern. Extensions to these systems are being developed to provide additional functionality such as strong data consistency and support for complex queries [Zhou et al., 2011; Das et al., 2010, 2009; Schütt et al., 2008; Kallman et al., 2008]. Provisioning such data stores is also challenging as their performance behavior is very different from that of relational databases. Furthermore, different NoSQL solutions employ different scalability techniques and thus exhibit different performance characteristics. Initial steps toward good performance models have been made in this direction [Istin, 2011; Trushkowsky et al., 2011]. However, this domain is still largely unexplored, and many research efforts will be necessary before we completely understand how to efficiently provision resources for applications relying on NoSQL data stores. In particular, NoSQL data stores have been shown to sometimes require extremely long stabilization periods before the performance effect of a new node is fully realized [Trushkowsky et al., 2011]. In such conditions, deciding *when* a data store should be reprovisioned becomes a crucial and difficult research challenge.

Samenvatting

Prestatiegaranties voor webapplicaties

Gebruikers stellen steeds meer eisen aan responsieve webapplicaties. Uit een onderzoek uit 2006 blijkt dat 62% van de internetgebruikers slechts 6 seconden of minder bereid is te wachten tot een enkele pagina te geladen is, voordat ze de website verlaten. Een meer recent onderzoek (2009) gaf aan dat deze prestatieverwachting hoger is geworden, en 83% van de internetgebruikers verwacht een webpagina te laden in 3 seconden of minder. Daarnaast toonde dit onderzoek ook dat 79% van de "online shoppers" die een bezoek aan een slecht presterende website brachten waarschijnlijk niets zouden kopen van deze site. Hieruit blijkt dat prestatiegaranties voor internetapplicaties bedrijfskritisch zijn.

Een belangrijke prestatienorm is de reactietijd van een webapplicatie. De reactietijd kan worden opgesplitst in drie delen: de wachttijd aan de gebruikerskant, de netwerkwachttijd en de wachttijd op de server. Onlangs hebben webapplicaties code die aan de op de machine van de gebruiker uitgevoerd wordt, zoals JavaScript, in gebruik genomen om de functionaliteit van applicaties uit te breiden. De wachttijd aan de gebruikerskant is de tijd die nodig is om de code aan de gebruikerskant uit te voeren. De onderzoeksgemeenschap heeft zich ingespannen om meerdere problemen met de prestaties aan de gebruikerskant op te lossen, zoals het gedrag van JavaScript code tijdens de executie te onderzoeken om de representativiteit van de benchmarkpakketten te verbeteren, het toepassen van bewaking op afstand ten behoeve van een prestatiediagnose van de gebruikerskant, en het doorvoeren van prestatieoptimalisaties voor JavaScript door op "traces" gebaseerde "just-in-time"-compilatie. De browseroorlog tussen de verschillende leveranciers in de ICT-industrie richt zich ook voor een belangrijk deel op prestatieverbeteringen voor JavaScript. De wachttijd aan de gebruikerskant hangt voornamelijk af van twee factoren: de toegepaste code aan de gebruikerskant, en specifieke mechanismen ingebouwd in elke webbrowser. Vanuit het perspectief van aanbieders van internetdiensten zijn deze twee factoren niet door hen te controleren.

De netwerkwachttijd verwijst naar de zendtijd van een reactie op een verzoek van de server naar de gebruiker over een netwerk zoals het Internet. Verschillende technieken, zoals “edge computing”, “caching” van gegevens, en de replicatie van gegevens zijn voorgesteld om deze wachttijden te verminderen. Commerciële producten zoals Akamai CDN en Amazon CloudFront richten zich ook op het waarborgen van de best mogelijke toegangsprestaties. Deze academische en industriële inspanningen tezamen reduceren de netwerkwachttijd van webapplicaties aanzienlijk, en zijn zeer succesvol.

Hoewel het optimaliseren van de wachttijd aan de gebruikerskant en op het netwerk van belang is, kunnen wij prestaties van een webapplicatie niet garanderen als niet ook de wachttijd op de server onder controle is. Eerdere experimenten toonden bijvoorbeeld aan dat de wachttijd op de server verantwoordelijk kan zijn voor bijna 50% van de totale wachttijd op een webapplicatie. Aangezien webapplicaties steeds complexer geworden kunnen we verwachten dat de wachttijd op de server alleen maar zal toenemen. De wachttijd op de server verwijst naar de verblijftijd van een inkomend verzoek op de server, dat wacht op een reactie. Een typische webapplicatie bestaat bijvoorbeeld uit een bedrijfslogicalaag en een gegevenslaag, waarbij de bedrijfslogicalaag kan worden uitgevoerd op een applicatieserver, terwijl de gegevenslaag vaak wordt uitgevoerd op een database-server. De wachttijd op de server omvat dan zowel de tijd benodigd voor het uitvoeren van de applicatiecode op de applicatieserver als de tijd benodigd voor het verkrijgen van gegevens van de database-server.

Het waarborgen van prestaties aan de aanbiederskant van een webapplicatie wordt bemoeilijkt door het feit dat de belasting van webapplicaties op computersystemen sterk fluctueert en zeer onvoorspelbaar is. Deze onvoorspelbaarheid en fluctuaties introduceren twee belangrijke eisen aan het hostingsysteem. Ten eerste moet een webapplicatiearchitectuur in staat zijn om willekeurige niveaus van belasting te kunnen accommoderen. Ten tweede moet het in staat zijn om haar eigen capaciteit aan te passen, teneinde wisselende volumes van webverkeer aan te kunnen.

Aan de ene kant kan men, gezien het feit dat webverkeer onvoorspelbaar is, niet op voorhand voorspellen wat de maximale werkbelasting van een webapplicatie zal zijn. Tegelijkertijd zijn aanbieders van webapplicaties erop gericht om zoveel mogelijk gebruikers aan te trekken voor een efficiëntere bedrijfsvoering. Daarom moet een webapplicatie schaalbaar zijn. Een schaalbare webapplicatie kan willekeurige volumes van verkeer verwerken door IT-middelen toe te voegen, zodat een acceptabel prestatieniveau behouden kan worden. De bouw van een schaalbare webapplicatie is echter niet eenvoudig, aangezien dit een zorgvuldige partitionering van zowel de bedrijfslogica- als datalaag vereist.

Aan de andere kant maken de fluctuaties in de werkdruk op de webapplica-

tie het onmogelijk om een "goede" vaste hostingcapaciteit tegen minimale kosten te bepalen. Kostenbewuste webapplicatieaanbieders, zoals bijvoorbeeld kleine en middelgrote aanbieders, verwachten een kosteneffectieve manier om hun applicaties te hosten. Door het "utility computing"-model toe te passen op de hosting van webapplicaties en het aantal IT-middelen dat webapplicaties gebruiken te variëren naar de daadwerkelijke belasting verwachten applicatieaanbieders de kosten te verminderen.

Utility computing biedt een model om IT-middelen te verpakken als een "bemeterde dienst". Sinds het jaar 2000 hebben IT-leveranciers zich ingespannen om producten en diensten te onwikkelen die het "utility computing"-model implementeren voor computerclusters en datacenters. Onlangs is cloud computing begonnen met het toepassen van utility computing door IT-middelen aan te bieden op een "pay-as-you-go" manier. In clouds worden IT-middelen zoals rekenkracht, dataopslag en netwerkcapaciteit verhuurd als diensten en afgerekend naar gebruik. Het "utility computing"-model voorziet in het dynamisch toekennen van IT-middelen aan webapplicaties om aan verschillende niveaus van vraag naar deze middelen te voldoen. Een efficiënte dynamische toekenning van middelen wordt echter bemoeilijkt door uitdagingen van zowel de kant van webapplicaties als van de kant van hostingomgevingen. Dat brengt ons tot de centrale onderzoeksvraag van dit proefschrift: hoe kunnen de prestaties op de server van webapplicaties gewaarborgd worden op een kosteneffectieve manier.

Dit proefschrift maakt gebruik van de gemiddelde reactietijd van de server als de prestatienorm voor webapplicaties. Andere prestatienormen, zoals percentielen van de reactietijd, zijn ook bruikbaar om prestatiegaranties mee uit te drukken. Wij geloven dat onze technieken kunnen worden uitgebreid om dergelijke normen ook te ondersteunen. De kwestie van het waarborgen van prestaties van de server kan worden vertaald naar het behouden van een redelijke gemiddelde reactietijd voor webapplicaties bij fluctuerende verkeersvolumes. Een redelijke reactietijd is gedefinieerd als de maximale reactietijd waarin een applicatie een binnenkomende aanvraag moet verwerken. Webapplicatieaanbieders definiëren deze maximale reactietijd doorgaans in hun "Service Level Objectives" (SLO's).

Naast het kiezen van prestatienormen gebruikt dit proefschrift het aantal gebruikte machines als een maat voor de kosten. Een gebruikte machine kan zowel een toegewezen fysieke machine in een cluster of een virtuele machine in een cloud zijn. Het aantal machines kan verder worden vertaald naar de monetaire kosten, als er een kostprijs per machine is vastgesteld.

Dit proefschrift behandelt voornamelijk twee aspecten van onze onderzoeksinspanningen om onze centrale onderzoeksvraag te beantwoorden: i) de bouw van een schaalbare webapplicatiearchitectuur, en ii) het ontwerpen van dynamische IT-middelentoewijzingssystemen.

De bouw van een schaalbare webapplicatie kan op twee manieren: opschalen en uitschalen. Opschalen betekent dat meer capaciteit, zoals de processorsnelheid en geheugen, aan de individuele applicatiesystemen en databasesystemen wordt toegevoegd. Uitschalen betekent daarentegen dat er meer systemen aan de twee lagen worden toegevoegd. Uitschalen presteert beter dan opschalen als de verhouding tussen prestaties en kosten voor webapplicaties in acht wordt genomen. Opschalen heeft ook een harde grens in de capaciteit van de hardware, terwijl uitschalen het mogelijk maakt om continu IT-middelen toe te voegen. Hierom construeren we in dit proefschrift een schaalbare webapplicatiearchitectuur met behulp van uitschaaltechnieken.

Het toevoegen van meer servers aan de bedrijslogicalaag van een webapplicatie kan de prestaties verbeteren, doordat de werklast voor iedere afzonderlijke server op dat niveau wordt verlicht. Het toewijzen van meer servers aan de gegevenslaag verbetert echter niet altijd de prestaties van die laag voor alle mogelijke niveaus van de belasting. Gedeeltelijke replicatie van gegevens en een zorgvuldige verdeling en plaatsing van gegevens kan leiden tot een verbeterde schaalbaarheid van de gegevenslaag, als er meer IT-middelen worden toegevoegd. De grove granulariteit van de verdeling beperkt echter de mate van schaalbaarheid van de huidige schaalvergrotingstechnieken. In dit proefschrift tonen we de potentiële schaalbaarheid van webapplicaties als gevolg van een fijnere granulariteit van de gegevensverdeling.

Hoewel een schaalbare webapplicatiearchitectuur veelbelovende mechanismen voor het waarborgen van de prestaties van webapplicaties biedt, worden webapplicaties nog steeds geconfronteerd met het probleem van fluctuerende verkeersvolumes. Het toewijzen van te veel middelen aan webapplicaties op basis van de maximale werkdruk kan leiden tot inefficiënt gebruik van IT-middelen, terwijl bij een toewijzing van te weinig IT-middelen een schending van de SLO wordt geriskeerd. De meest eenvoudige technologie die gebruikt wordt om de prestaties te garanderen voor webapplicaties bij fluctuerende verkeersvolumes is een dynamische IT-middelentoewijzing. Deze technologie bestaat uit het toewijzen van extra IT-middelen aan een webapplicatie wanneer de reactietijd de SLO dreigt te overtreden, en het afnemen van matig gebruikte IT-middelen van een webapplicatie indien dit mogelijk is met behoud van de SLO.

Complexe webapplicaties en heterogene hostingomgevingen vormen echter een probleem voor de huidige dynamische IT-middelentoewijzingstechnieken. Aan de ene kant zijn de huidige webapplicaties niet ontworpen als monolithische applicaties, bestaande uit drie lagen. De webapplicatie die gebruikt wordt om webpagina's van Amazon.com te genereren bestaat bijvoorbeeld uit honderden diensten. Het is in dergelijke applicaties, die bestaan uit meerdere interactieve diensten, moeilijk te achterhalen wat het knelpunt is voor de prestaties. Het is

nog moeilijker om dit probleem op te lossen met behulp van een dynamische en efficiënte toewijzinging van IT-middelen. Aan de andere kant leiden heterogene fysieke machines en virtuele machines in datacentra en clouds tot heterogene prestaties van virtuele hostingsmiddelen. Deze eigenschap beperkt ook de toepasbaarheid van de huidige middeltoewijzingstechnieken die uitgaan van het gebruik van homogene middelen.

Bibliography

Abdelzaher, T. F. and Bhatti, N. (1999). Web content adaptation to improve server overload behavior. In *Proceedings of the 8th International World Wide Web Conference*, pages 1563–1577.

Abrahao, B., Almeida, V., Almeida, J., Zhang, A., Beyer, D., and Safai, F. (2006). Self-adaptive sla-driven capacity management for internet services. In *Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium*, pages 557–568.

Akamai (2006). Akamai: The Leader in Web Application Acceleration and Performance Management, Streaming Media, Retrived in May, 2011 from http://www.akamai.com/.

Amazon AutoScale (2011). Retrieved in August, 2011 from http://aws.amazon.com/autoscaling/.

Amazon CloudFront (2011). Retrieved in May, 2011 from http://aws.amazon.com/cloudfront/.

Amazon EC2 (2011). Retrieved in May, 2011 from http://aws.amazon.com/ec2/.

Amazon SimpleDB (2011). Retrieved in July, 2011 from http://aws.amazon.com/simpledb/.

Amiri, K., Park, S., Tewari, R., and Padmanabhan, S. (2003). DBProxy: A dynamic data cache for web applications. *Proceedings of the 19th International Conference on Data Engineering*, pages 821–831.

Amza, C., Chanda, A., Cox, A., Elnikety, S., Gil, R., Rajamani, K., Zwaenepoel, W., Cecchet, E., and Marguerite, J. (2002). Specification and implementation of dynamic web site benchmarks. In *Proceedings of the International Workshop on Workload Characterization*, pages 3–13.

Amza, C., Cox, A. L., and Zwaenepoel, W. (2003a). Conflict-aware scheduling for dynamic content applications. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pages 6–19.

Amza, C., L., C. A., and Willy, Z. (2003b). Distributed versioning: consistent replication for scaling back-end databases of dynamic content web sites. In *Proceedings of the 4th International Middleware Conference*, pages 282–304.

Andreolini, M., Colajanni, M., Lancellotti, R., and Mazzoni, F. (2004). Fine grain performance evaluation of e-commerce sites. *SIGMETRICS Performance Evaluation Review*, 32(3):14–23.

Arlitt, M. and Jin, T. (2000). A workload characterization study of the 1998 world cup web site. *IEEE Network*, 14(3):30 – 37.

Arlitt, M. F. and Williamson, C. L. (1996). Web server workload characterization: the search for invariants. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 126–137.

Armando, F., D., G. S., Yatin, C., A., B. E., and Paul, G. (1997). Cluster-based scalable network services. In *Proceedings of the 16th Symposium on Operating Systems Principles*, pages 78–91.

Arnaud, J. and Bouchenak, S. (2010). Adaptive internet services through performance and availability control. In *Proceedings of the Symposium on Applied Computing*, pages 444–451.

Arnaud, J. and Bouchenak, S. (2011). *Performance, Availability and Cost of Self-Adaptive Internet Services*, chapter 4, pages 212–241. IGI Global.

Aron, M., Sanders, D., Druschel, P., and Zwaenepoel, W. (2000). Scalable content-aware request distribution in cluster-based networks servers. In *Proceedings of the USENIX Annual Technical Conference*, pages 26–39.

Awadallah, A. and Rosenblum, M. (2002). The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution*.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating Systems Principles*, pages 164–177.

Bennani, M. N. and Menascé, D. A. (2005). Resource allocation for autonomic data centers using analytic performance models. In *Proceedings of the 2nd International Conference on Autonomic Computing*, pages 229–240.

Bettina, K. and Gustavo, A. (2000). Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the 26th International Conference on Very Large Databases*, pages 134–143.

Bhatti, N. and Friedrich, R. (1999). Web server support for tiered services. *IEEE Network*, 13(5):64–71.

Bhoj, P., Ramanathan, S., and Singhal, S. (2000). Web2K: Bringing QoS to web servers. Technical Report HPL-2000-61, Internet Systems and Applications Laboratory, HP Laboratories Palo Alto.

Bialek, B. and Tassi, B. (2006). IBM DB2 integrated cluster environment v2. Retrieved in June, 2011 from ftp://ftp.software.ibm.com/software/data/pubs/papers/db2ice_v2setup.pdf.

Bornhvd, C., Altinel, M., Mohan, C., Pirahesh, H., and Reinwald, B. (2004). Adaptive database caching with DBCache. *Data Engineering*, 27(2):11–18.

Calzarossa, M., Massari, L., and Tessera, D. (2000). Workload characterization issues and methodologies. In *Proceedings of Performance Evaluation: Origins and Directions*, pages 459–481.

Cardellini, V., Casalicchio, E., Colajanni, M., and Yu, P. S. (2002). The state of the art in locally distributed web-server systems. *ACM Computer Survey*, 34(2):263–311.

Cassandra (2011). Apache Cassandra Project. Retrieved in August, 2011 from http://cassandra.apache.org.

Cecchet, E., Marguerite, J., and Zwaenepoel, W. (2004). C-JDBC: flexible database clustering middleware. In *Proceedings of the USENIX Annual Technical Conference*, pages 26–35.

Chandra, A., Gong, W., and Shenoy, P. (2003). Dynamic resource allocation for shared data centers using online measurements. In *Proceedings of the 11th International Conference on Quality of Service*, pages 381–398.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2006). Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 15–28.

Chang, M., Smith, E., Reitmaier, R., Bebenita, M., Gal, A., Wimmer, C., Eich, B., and Franz, M. (2009). Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the 5th International Conference on Virtual Execution Environments*, pages 71–80.

Chen, Y., Iyer, S., Liu, X., Milojicic, D., and Sahai, A. (2007). SLA decomposition: Translating service level objectives to system level thresholds. In *Proceedings of the 4th International Conference on Autonomic Computing*, pages 3–12.

Cherkasova, L. and Gardner, R. (2005). Measuring CPU overhead for I/O processing in the xen virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference*, pages 24–27.

Cherkasova, L. and Phaal, P. (2002). Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Transactions on Computers*, 51(6):669–685.

Christopher, S., Terence, K., Alex, Z., and Kai, S. (2008). A dollar from 15 cents: cross-platform management for internet services. In *Proceedings of the USENIX Annual Technical Conference*, pages 199–212.

Cohen, A., Rangarajan, S., and Slye, H. (1999). On the performance of TCP splicing for URL-aware redirection. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, pages 11–19.

Cohen, E. and Kaplan, H. (2001). Proactive caching of DNS records: Addressing a performance bottleneck. In *Proceedings of the Symposium on Applications and the Internet*, pages 85–94.

Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., and Chase, J. S. (2004). Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *Proceedings of the 6th Symposium on Opearting Systems Design and Implementation*, pages 16–29.

Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. (2008). Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of VLDB Endowment*, 1(2):1277–1288.

Cunha, I., Almeida, J., Almeida, V., and Santos, M. (2007). Self-adaptive capacity management for multi-tier virtualized environments. In *Proceedings of the 10th International Symposium on Integrated Network Management*, pages 129–138.

Dar, S., Franklin, M. J., Jónsson, B. T., Srivastava, D., and Tan, M. (1996). Semantic data caching and replacement. In *Proceedings of the 22th International Conference on Very Large Databases*, pages 330–341.

Das, S., Agrawal, D., and El Abbadi, A. (2009). Elastras: an elastic transactional data store in the cloud. In *Proceedings of the Conference on Hot Topics in Cloud Computing*.

Das, S., Agrawal, D., and El Abbadi, A. (2010). G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st Symposium on Cloud computing*, pages 163–174.

DAS 3 (2011). DAS 3: The Distributed ASCI Supercomputer 3. Retrieved in June, 2011 from http://www.cs.vu.nl/das3.

Davis, A., Parikh, J., and Weihl, W. E. (2004). Edgecomputing: extending enterprise applications to the edge of the internet. In *Proceedings of the 13th International World Wide Web Conference*, pages 180–187.

de Bruijn, W. (2010). *Adaptive Operating System Design for High Throughput I/O*. PhD thesis, Vrije Universiteit Amsterdam.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st Symposium on Operating Systems Principles*, pages 205–220.

Dejun, J., Pierre, G., and Chi, C.-H. (2009). EC2 performance analysis for resource provisioning of service-oriented applications. In *Proceedings of the 7th International Conference on Service Oriented Computing*, pages 197–207.

Dejun, J., Pierre, G., and Chi, C.-H. (2010). Autonomous resource provisioning for multi-service web applications. In *Proceedings of the 19th International Conference World Wide Web*, pages 471–480.

Doyle, R. P., Chase, J. S., Asad, O. M., Jin, W., and Vahdat, A. M. (2003). Model-based resource provisioning in a web service utility. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pages 5–18.

Duvos, E. and Bestavros, A. (2000). An infrastructure for the dynamic distribution of web applications and services. Technical Report BUCS-TR-2000-027, Department of Computer Science, Boston University.

Elnikety, S., Dropsho, S., and Zwaenepoel, W. (2007). Tashkent+: memory-aware load balancing and update filtering in replicated databases. In *Proceedings of the 2nd European Conference on Computer Systems*, pages 399–412.

Elnikety, S., Nahum, E., Tracey, J., and Zwaenepoel, W. (2004). A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th International World Wide Web Conference*, pages 276–286.

Forrester Research (2009). eCommerce web site performance today: An updated look at consumer reaction to a poor online shopping experience. Retrieved in May, 2011 from http://www.akamai.com/2seconds.

Gao, L., Dahlin, M., Nayate, A., Zheng, J., and Iyengar, A. (2003). Application specific data replication for edge services. In *Proceedings of the 12th International World Wide Web Conference*, pages 449–460.

Georgiadis, L., Nikolaou, C., and Thomasian, A. (2004). A fair workload allocation policy for heterogeneous systems. *Journal of Parallel and Distributed Computing*, 64:507–519.

Gill, P., Arlitt, M., Li, Z., and Mahanti, A. (2007). Youtube traffic characterization: a view from the edge. In *Proceedings of the 7th Internet Measurement Conference*, pages 15–28.

Google V8 (2011). V8 google's open source javascript engine. Retrieved in May, 2011 from http://code.google.com/apis/v8/benchmarks.html.

Groothuyse, T., Sivasubramanian, S., and Pierre, G. (2007). GlobeTP: template-based database replication for scalable web applications. In *Proceedings of the 16th International World Wide Web Conference*, pages 301–310.

Gunther, N. J. (2004). *Analyzing Computer Systems Performance: With Perl: PDQ*. SpringerVerlag.

HBase (2011). Apache HBase Project. Retrieved in August, 2011 from http://hbase.apache.org/.

Huang, Y. and Chen, J. (2001). Fragment allocation in distributed database design. *Journal of Information Science and Engineering*, 17:491–506.

Iosup, A., Ostermann, S., Yigitbasi, N., Prodan, R., Fahringer, T., and Epema, D. (2011a). Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):931–945.

Iosup, A., Yigitbasi, N., and Epema, D. (2011b). On the performance variability of production cloud services. In *Proceedings of the 11th International Symposium on Cluster, Cloud and Grid Computing*, pages 104–113.

Istin, M.-D. (2011). Resource provisioning for NoSQL datastores. Master's thesis, Vrije Universiteit Amsterdam.

Jin, L., Machiraju, V., and Sahai, A. (2002). Analysis on service level agreement of web services. Technical Report HPL-2002-180, Software Technology Laboratory, HP Laboratories Palo Alto.

Jupiter Research (2006). Retail web site performance: Consumer reaction to a poor online shopping experience. Retrieved in May, 2011 from http://www.akamai.com/4seconds.

Kallahalla, M., Uysal, M., Swaminathan, R., Lowell, D. E., Wray, M., Christian, T., Edwards, N., Dalton, C. I., and Gittler, F. (2004). SoftUDC: A software-based data center for utility computing. *Computer*, 37(11):38–46.

Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. (2008). H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of VLDB Endowment*, 1:1496–1499.

Kamra, A., Misra, V., and Nahum, E. M. (2004). Yaksha: a self-tuning controller for managing the performance of 3-tiered web sites. In *Proceedings of the 12th International Workshop on Quality of Service*, pages 47–56.

Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. (1997). Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Symposium on Theory of Computing*, pages 654–663.

Kazerouni, L. and Karlapalem, K. (1997). Stepwise redesign of distributed relational databases. Technical report, Department of Computer Science, Hong Kong University of Science and Technology.

Kıcıman, E. and Livshits, B. (2007). AjaxScope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Proceedings of the 21st Symposium on Operating Systems Principles*, pages 17–30.

Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2):35–40.

Li, K. and Jamin, S. (2000). A measurement-based admission-controlled web server. In *Proceedings of the 19th INFOCOM Conference*, pages 651–659.

Li, P. (2010). Service-oriented data denormalization for mediawiki. Master's thesis, Vrije Universiteit Amsterdam.

Luo, Q. and Naughton, J. F. (2001). Form-based proxy caching for database-backed web sites. In *Proceedings of the 27th International Conference on Very Large Databases*, pages 191–200.

Mann, G., Sandler, M., Krushevskaja, D., Guha, S., and Even-dar, E. (2011). Modeling the parallel execution of black-box services. In *Proceedings of the 3rd Workshop on Hot Topics in Cloud Computing*.

Marin, G. and Mellor-Crummey, J. (2004). Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 2–13.

Matt, W., David, C., and Eric, B. (2001). SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th Symposium on Operating Systems Principles*, pages 230–243.

Menascé, D. and Akula, V. (2003). Towards workload characterization of auction sites. In *Proceedings of the International Workshop on Workload Characterization*, pages 12–20.

Menascé, D. A. (2002). QoS issues in web services. *IEEE Internet Computing*, 6(6):72–75.

Menon, A., Santos, J. R., Turner, Y., Janakiraman, G. J., and Zwaenepoel, W. (2005). Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st International Conference on Virtual Execution Environments*, pages 13–23.

Mi, N., Casale, G., Cherkasova, L., and Smirni, E. (2008). Burstiness in multi-tier applications: symptoms, causes, and new models. In *Proceedings of the 9th International Middleware Conference*, pages 265–286.

Michael, M., Moreira, J. E., Shiloach, D., and Wisniewski, R. W. (2007). Scale-up x scale-out: A case study using nutch/lucene. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 1–8.

MySQL Cluster (2011). MySQL cluster architecture. Retrieved in June, 2011 from http://www.mysql.com/products/cluster/architecture.html.

MySQL Replication (2011). MySQL Replication. Retrieved in August, 2011 from http://dev.mysql.com/doc/refman/5.0/en/replication.html.

Nahum, E., Barzilai, T., and Kandlur, D. D. (2002). Performance issues in WWW servers. *Transactions on Networking*, 10(1):2–11.

Navathe, S. B., Karlapalem, K., and Ra, M. (1995). A mixed fragmentation methodology for initial distributed database design. *Computer and Software Engineering*, 3.

Navathe, S. B. and Ra, M. (1989). Vertical partitioning for database design: a graphical algorithm. In *Proceedings of the International Conference on Management of Data*, pages 440–450.

Novella, B., Emiliano, C., and Salvatore, T. (2004). A walk through content delivery networks. In *Performance Tools and Applications to Networked Systems*, volume 2965 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin Heidelberg.

Olshefski, D. P., Nieh, J., and Nahum, E. (2004). ksniffer: determining the remote client perceived response time from live packet streams. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 23–36.

Oracle Cluster (2010). Oracle 11g oracle real application clusters. Retrieved in June, 2011 from http://www.oracle.com/technetwork/database/clustering/overview/index.html.

Ozsu, M. T. and Valduriez, P. (1999). *Principles of distributed database systems*. Prentice-Hall, Inc.

Pai, V. S., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., and Nahum, E. (1998). Locality-aware request distribution in cluster-based network servers. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216.

Pai, V. S., Druschel, P., and Zwaenepoel, W. (1999). Flash: an efficient and portable web server. In *Proceedings of the USENIX Annual Technical Conference*, pages 15–27.

Pai, V. S., Druschel, P., and Zwaenepoel, W. (2000). IO-Lite: a unified I/O buffering and caching system. *Transactions on Computer Systems*, 18(1):37–66.

Pathan, A. K. and Buyya, R. (2007). A taxonomy and survey of content delivery networks. Technical Report GRIDS-TR-2007-4, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia.

Paulson, L. D. (2005). Building rich web applications with Ajax. *Computer*, 38(10):14–17.

Plattner, C. and Alonso, G. (2004). Ganymed: scalable replication for transactional web applications. In *Proceedings of the 5th International Middleware Conference*, pages 155–174.

Pradhan, P., Tewari, R., Sahu, S., Ch, A., and Shenoy, P. (2002). An observation-based approach towards self-managing web servers. In *Proceedings of the 10th International Workshop on Quality of Service*, pages 13–22.

Rabinovich, M. and Aggarwal, A. (1999). Radar: a scalable architecture for a global web hosting service. *Computer Networks*, 31(11-16):1545–1561.

Rabinovich, M., Xiao, Z., and Aggarwal, A. (2004). Computing on the edge: A platform for replicating internet applications. In *Web Content Caching and Distribution*, pages 57–77. Springer Netherlands.

Rackspace (2011). Cloud computing, managed hosting, dedicated server hosting by rackspace. Retrieved in May, 2011 from http://www.rackspace.com/.

Rajamony, R. and Elnozahy, M. (2001). Measuring client-perceived response times on the www. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, pages 16–27.

Ranjan, S., Rolia, J., Fu, H., and Knightly, E. (2002). QoS-Driven server migration for internet data centers. In *Proceedings of the 10th International Workshop on Quality of Service*, pages 3–12.

Ratanaworabhan, P., Livshits, B., and Zorn, B. G. (2010). JSMeter: comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the 1st Conference on Web Application Development*, pages 3–14.

Richards, G., Gal, A., Eich, B., and Vitek, J. (2011). Towards automatically constructing representative javascript workloads. Technical report, Secure Software System Lab Department of Computer Science Purdue University.

Richards, G., Lebresne, S., Burg, B., and Vitek, J. (2010). An analysis of the dynamic behavior of javascript programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 1–12.

RuBBoS (2011). RUBBoS: bulletin board system benchmark. Retrieved in July, 2011 from http://jmob.ow2.org/rubbos.html.

Sanders, G. and Shin, S. (2001). Denormalization effects on performance of RDBMS. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, pages 3013–3021.

Schad, J., Dittrich, J., and Quiané-Ruiz, J.-A. (2010). Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment*, 3(1-2):460–471.

Schroeder, B. and Harchol-Balter, M. (2006). Web servers under overload: How scheduling can help. *Transactions on Internet Technology*, 6(1):20–52.

Schütt, T., Schintke, F., and Reinefeld, A. (2008). Scalaris: reliable transactional p2p key/value store. In *Proceedings of the 7th SIGPLAN Workshop on ERLANG*, pages 41–48.

Shankland, S. (2009). Browser war centers on once-obscure javascript. Retrieved in May, 2011 from http://news.cnet.com/browser-war-centers-on-once-obscure-javascript.

Shen, K., Tang, H., Yang, T., and Chu, L. (2002). Integrated resource management for cluster-based internet services. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 225–238.

Shi, W., Wright, Y., Collins, E., and Karamcheti, V. (2002). Workload characterization of a personalized web site and its implications for dynamic content caching. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution*.

Shin, S. K. and Sanders, G. L. (2006). Denormalization strategies for data retrieval from data warehouses. *Decision Support Systems*, 42(1):267–282.

Shoup, R. (2008). eBay's architectural principles. Retrieved in July, 2011 from http://jaoo.dk/london-2008/file?path=/qcon-london-2008/slides/RandyShoup_-eBaysArchitecturalPrinciples.pdf.

Shoup, R. and Pritchett, D. (2006). The eBay architecture: Striking a balance between site stability, feature velocity, performance and cost. Retrieved in May, 2011 from http://www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf.

Sivasubramanian, S. (2007). *Scalable Hosting of Web Applications*. PhD thesis, Vrije Universiteit Amsterdam.

Sivasubramanian, S., Alonso, G., Pierre, G., and van Steen, M. (2005). GlobeDB: autonomic data replication for web applications. In *Proceedings of the 14th International World Wide Web Conference*, pages 33–42.

Sivasubramanian, S., Pierre, G., and Steen, M. V. (2006). GlobeCBC: Content-blind result caching for dynamic web applications. Technical Report Technical report IR-CS-022, Vrije Universiteit Amsterdam.

Sivasubramanian, S., Pierre, G., van Steen, M., and Alonso, G. (2007). Analysis of caching and replication strategies for web applications. *IEEE Internet Computing*, 11(1):60–66.

Slothouber, L. P. (1996). A model of web server performance. In *Proceedings of the 5th International World wide web Conference*.

Smith, J. E. and Nair, R. (2005). The architecture of virtual machines. *Computer*, 38(5):32–38.

Stewart, C. and Shen, K. (2005). Performance modeling and system management for multi-component online services. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, pages 71–84.

Sun Cloud (2000). Sun cloud. Retrieved in May, 2011 from http://en.wikipedia.org/wiki/Sun_Cloud.

SunSpider (2011). Sunspider javascript benchmark. Retrieved in May, 2011 from http://www.webkit.org/perf/sunspider/sunspider.html.

Tesauro, G., Jong, N., Das, R., and Bennani, M. (2006). A hybrid reinforcement learning approach to autonomic resource allocation. In *Proceedings of the 3rd International Conference on Autonomic Computing*, pages 65–73.

TPC-W (2011). TPC-W: a transactional web e-commerce benchmark. Retrieved in July, 2011 from http://www.tpc.org/tpcw.

TPC-W implementation (2011). Java TPC-W implementation distribution. Retrieved in June, 2011 from http://www.ece.wisc.edu/ pharm/tpcw.shtml.

Trivedi, K. S. (2002). *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons Ltd.

Trushkowsky, B., Bodík, P., Fox, A., Franklin, M. J., Jordan, M. I., and Patterson, D. A. (2011). The SCADS director: scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th Conference on File and Storage Technologies*, pages 12–25.

Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. (2005a). An analytical model for multi-tier internet services and its applications. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 291–302.

Urgaonkar, B., Prashant, S., Abhishek, C., Pawan, G., and Timothy, W. (2008). Agile dynamic provisioning of multi-tier internet applications. *Transactions on Autonomous and Adaptive Systems*, 3(1):1–39.

Urgaonkar, B. and Shenoy, P. (2005). Cataclysm: policing extreme overloads in internet applications. In *Proceedings of the 14th International World Wide Web Conference*, pages 740–749.

Urgaonkar, B., Shenoy, P., Chandra, A., and Goyal, P. (2005b). Dynamic provisioning of multi-tier internet applications. In *Proceedings of the 2nd International Conference on Autonomic Computing*, pages 217–228.

Urgaonkar, B., Shenoy, P., and Roscoe, T. (2002). Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 239–254.

Villela, D., Pradhan, P., and Rubenstein, D. (2007). Provisioning servers in the application tier for e-commerce systems. *Transactions on Internet Technology*, 7(1).

Vogels, W. and Gray, J. (2006). A conversation with Werner Vogels. *Queue*, 4(4):14–22.

Welsh, M. and Culler, D. (2003). Adaptive overload control for busy internet servers. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pages 4–17.

Wikipedia (2011a). Browser wars. Retrieved in June, 2011 from http://en.wikipedia.org/wiki/Browser_wars.

Wikipedia (2011b). Scalability. Retrieved in May, 2011 from http://en.wikipedia.org/wiki/Scalability.

Wikipedia (2011c). Utility computing. Retrieved in May, 2011 from http://en.wikipedia.org/wiki/Utility_computing.

Wikipedia (2011d). Wikipedia. Retrieved in May, 2011 from http://en.wikipedia.org/wiki/Wikipedia.

Williams, A., Arlitt, M., Williamson, C., and Barker, K. (2005). Web workload characterization: Ten years later. In *Web Content Delivery*, volume 2 of *Web Information Systems Engineering and Internet Technologies*, pages 3–21. Springer US.

Xiaolan, Z., Michael, B., Bradley, C. J., and Margo, S. (1999). HACC: an architecture for cluster-based web servers. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 16–25.

Yang, C.-S. and Luo, M.-Y. (2000). A content placement and management system for distributed web-server systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, pages 691–698.

Zari, M., Saiedian, H., and Naeem, M. (2001). Understanding and reducing web delays. *Computer*, 34(12):30–37.

Zhang, Q., Cherkasova, L., and Smirni, E. (2007). A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Proceedings of the 4th International Conference on Autonomic Computing*, pages 27–36.

Zhang, Q., Riska, A., Riedel, E., and Smirni, E. (2004). Bottlenecks and their performance implications in e-commerce systems. In *Web Content Caching and Distribution*, volume 3293 of *Lecture Notes in Computer Science*, pages 273–282. Springer Berlin / Heidelberg.

Zhang, Q., Riska, A., Sun, W., Smirni, E., and Ciardo, G. (2005). Workload-aware load balancing for clustered web servers. *Transactions on Parallel Distributed Systems*, 16(3):219–233.

Zhang, W., Qian, H., Wills, C. E., and Rabinovich, M. (2010). Agile resource management in a virtualized data center. In *Proceedings of the 1st Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 129–140.

Zheng, W., Bianchini, R., Janakiraman, G. J., Santos, J. R., and Turner, Y. (2009). JustRunIt: experiment-based management of virtualized data centers. In *Proceedings of the USENIX Annual Technical Conference*, pages 18–33.

Zhou, W., Dejun, J., Pierre, G., Chi, C.-H., and van Steen Maarten (2008). Service-oriented data denormalization for scalable web applications. In *Proceeding of the 17th International World Wide Web Conference*, pages 267–276.

Zhou, W., Pierre, G., and Chi, C.-H. (2011). CloudTPS: Scalable transactions for web applications in the cloud. *IEEE Transactions on Services Computing*, 99.

Zuikevičiūtė, V. and Pedone, F. (2008). Conflict-aware load-balancing techniques for database replication. In *Proceedings of the 23rd Annual Symposium on Applied Computing*, pages 2169–2173.