Automating Datacenter Operations Using Machine Learning

by

Peter Bodík

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Fall 2010

Automating Datacenter Operations Using Machine Learning

Abstract

Automating Datacenter Operations Using Machine Learning

by

Peter Bodík

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David A. Patterson, Co-chair
Professor Michael I. Jordan, Co-chair

Today's Internet datacenters run many complex and large-scale Web applications that are very difficult to manage. The main challenges are understanding user workloads and application performance, and quickly identifying and resolving performance problems. Statistical Machine Learning (SML) provides a methodology for quickly processing the large quantities of monitoring data generated by these applications, finding repeating patterns in their behavior, and building accurate models of their performance.

This dissertation argues that SML is a useful tool for simplifying and automating datacenter operations and demonstrates application of SML to three important problems in this area: characterization and synthesis of workload spikes, dynamic resource allocation in stateful systems, and quick and accurate identification of recurring performance problems.

Dedicated to Zuzu

Contents

List of Figures

List of Tables

Acknowledgments

Despite ultimately having one name as its author, this dissertation is, by any measure, the work of many who have helped, guided, encouraged, and stood by me all along the way. It is a pleasure to thank the many people who made this thesis possible.

First and foremost, I thank my advisors David Patterson, Armando Fox, and Michael Jordan. Dave's mentorship, years of experience and unique insights were an immense asset to my research. I was truly fortunate to have had the opportunity to learn from and work with him. I am indebted to Armando Fox for his guidance throughout my graduate education. I learned a lot from interactions with Armando; he kept me focused by always asking sharp and challenging questions and showed me the need to be persistent to accomplish any goal. I am also extremely grateful to my research advisor Michael Jordan for my statistical machine learning education. Mike's expertise was essential for applying statistical techniques to challenging problems described in this dissertation.

I was fortunate to spend most of my graduate career in the RAD Lab and I am thankful to many students for creating a stimulating and fun environment. Many of my colleagues provided invaluable suggestions that substantially improved the content of this dissertation. I thank Berkeley undergrads Aaron Beitch, Timothy Yung, Hubert Wong, Jimmy Nguyen, and David Schleimer who worked in the RAD Lab and together with Will Sobel built the infrastructure used in the early Director experiments. I want to thank the members of the SCADS team – Beth Trushkowsky, Michael Armbrust, and Nick Lanham – for building SCADS and thus providing the testbed for the Director framework. I am grateful to all RAD Lab affiliates who participated in the RAD Lab retreats and provided feedback along the way, and also to Cecilia Pracher, Kattt Atchley, Sean McMahon, Mike Howard, and Jon Kuroda who supported RAD Lab over the years.

I am grateful to Moises Goldszmidt for the opportunity to work with him for a year at Microsoft Research Silicon Valley. He introduced me to the industry research environment and our collaboration led to the results on crisis fingerprinting. I would also like to thank the people at the Monitoring team at Amazon.com led by Jon Ingalls where I interned during the summer of 2005. My experience at Amazon.com motivated most of my research.

I would like to thank Mike Franklin, Eric Brewer, and Joe Hellerstein with whom I interacted and who also provided me with guidance and advice. I also thank Philip Stark for kindly agreeing to be part of this committee, for his time and advice.

My deepest gratitude goes to my family for their unflagging love and support throughout my life; this dissertation would be simply impossible without them. I am thankful for my parents encouragement to pursue my interests, even when my interests made me leave my home country. I am grateful to my brother Ras Bodik for being my "shadow-advisor" during grad school. He introduced me to computer science at an early age and was also my inspiration to pursue grad school in the United States. Finally, I am extremely grateful to my amazing wife Zuzka for her support during the long seven years. She stood by me during the good times and the tough times and made my journey to the finish line much more enjoyable.

Chapter 1

Introduction

Good performance and high availability are the primary goals of companies running Web applications that we rely on in our daily lives, such as Google or Facebook. Once Web applications are deployed to a production environment, the task of datacenter operators is to constantly monitor the applications, notice possible performance issues, and quickly resolve them. Because application downtime could potentially cost millions of dollars, the operators are often required to respond to an alert within 10 or 15 minutes [Bodík *et al.*, 2006], even at night or on the weekends.

Due to increasing complexity and large-scale deployment of popular Web applications, the job of datacenter operators is becoming more difficult. For example, Web applications are often deployed in multiple, globally-distributed datacenters for improved performance and redundancy. Typical datacenters run tens of thousands of commodity servers that frequently fail; during the first year of a typical 2,000-node cluster, Google experiences 20 rack failures, 1000 machine failures, thousands of disk failures as well as many other types of failures [Dean, 2009]. The functionality of Web applications is often provided by tens or hundreds of independent software services that are constantly evolving. For example, eBay deploys 300 features per quarter and adds 100,000 new lines of code every two weeks [Shoup, 2007]. Moreover, requests of different applications share the same hardware or network equipment.

To maintain good performance and high availability in such a complex and constantly changing environment, the datacenter operators thus face three main challenges: understanding application workload, understanding application performance under different workloads and system configurations, and quickly detecting, identifying, and resolving performance problems.

First, operators need to understand typical workload patterns and long-term trends and make sure the application can handle unexpected workload spikes. Most Web applications have highly predictable daily, weekly, and yearly patterns that the operators use for short-term and long-term resource provisioning. However, tens or hundreds of millions of users can create very unexpected workload spikes and data hotspots that can overload the application. For example, after Michael Jackson's death, the increase in aggregate workload on Wikipedia.org was only about 5%, however, 13% of all requests were directed at Jackson's article [Bodík *et al.*, 2010]. Datacenter operators have to understand properties of these spikes so they can stress-test the application and appropriately provision resources.

Second, because of large-scale deployment, complex workloads, and software dependencies, very few people understand the impact of changes in application code, resource utilization, or hardware on the end-to-end request latency. However, understanding what affects application performance is crucial for optimizing request latency or for dynamically provisioning more resources in response to changes in user workload. Because in modern datacenters different applications share the same hardware, it is also important to know the resource requirements of individual applications in order to maximize the utilization of the whole datacenter.

Finally, datacenter operators need to quickly detect, identify, and resolve various performance crises that occur in the datacenter. Most crises are detected automatically through increased request latency or decreased throughput. However, some problems are difficult to detect, because they are caused by small, but important changes in the Web site content or the cause of the problem is outside of the datacenter. The difficulty of crisis resolution varies significantly. Many of the less severe crises are resolved automatically in the application through various methods such as redundancy, but more severe crises require human intervention because no known automatic resolution is available. While crisis resolution often lasts only a few minutes, in some cases it might last up to a few hours and require a large group of operators to analyze the system.

Even though today's datacenters and Web applications are very well instrumented and record workload statistics, utilization metrics of servers, and detailed application logs, it is often difficult to extract useful information from the collected data. Operators and application developers use this data for debugging and troubleshooting the system because it provides insights into workload, performance and various failures. However, there exist very few tools for automatic analysis of this data that would allow the operators to notice trends or detect anomalies. Due to large volume of the monitoring data and application logs, datacenter operators can't take advantage of information available in this data. Current approaches to datacenter operations are thus slow and manual.

Statistical Machine Learning (SML) provides a methodology for quickly processing the large quantities of monitoring data generated by these applications, finding repeating patterns in their behavior, and building accurate models of their performance. For example, *regression* methods allow us to create models of application performance as a function of user workload, and software and hardware configuration. *Anomaly detection* methods can be used to automatically spot deviations from normal system behavior that could correspond to application failures. *Feature selection* allows the operators to automatically find performance metrics correlated with occurrences of application failures which could lead to faster diagnosis of these problems. Moreover, SML could be combined with non-SML methods like optimization, visualization, or control theory to create even more powerful operator tools.

In this dissertation, we argue that SML is a useful tool for simplifying and automating datacenter operations and we demonstrate application of SML to three important problems in this area: characterization and synthesis of workload spikes, dynamic resource allocation in stateful systems, and quick and accurate identification of recurring performance problems.

In Chapter 2, we analyze changes in application workload that occur during workload spikes and create a model of such spikes. This model is simple and statistically motivated, and can be used to synthesize new workload spikes for stress-testing of web applications.

In Chapter 3, we address the problem of dynamic resource allocation in stateful systems. We build a model of system performance under different workloads and then use Model-Predictive Control to add or remove resources in response to changes in workload.

Finally, in Chapter 4, we propose a method for quick identification of recurring performance crises in datacenter applications. We first use feature selection techniques to pick a small set of performance metrics that correlate with occurrences the crises and use these metrics to construct a unique fingerprint of each crisis. When a new crisis is detected, its fingerprint is compared to a fingerprint database of past crises to help operators identify the current crisis and quickly resolve it.

As the datacenters and Web applications grow larger, maintaining them is becoming more difficult and time-consuming. In the following three chapters we demonstrate the usefulness of using statistical and machine learning methods for building models of workload, performance, and failures of large-scale systems. We hope that in the future, more of these techniques will be incorporated into tools that the datacenter operators use daily.

Chapter 2

Characterization and Synthesis of Workload Spikes in Stateful Systems

2.1 Introduction

A public-facing Internet-scale service available to tens of millions of users can experience very rapid and unexpected shifts in workload patterns. Handling such spikes is extremely challenging. Provisioning for them in advance is analogous to preparing for earthquakes: while designing for a magnitude-9 earthquake is theoretically possible, it is economically infeasible because such events are very rare. Instead, engineers use statistical characterizations of earthquakes—distributions of magnitude, frequency, and phase—to design buildings that can withstand most earthquakes. Similarly, while overprovisioning an Internet service to handle the largest possible spike is infeasible, pay-as-you-go cloud computing offers the option to quickly add capacity to deal with some types of spikes. However, exploiting this feature requires testing the service under such scenarios and understanding the nature of unexpected events: how fast they ramp up, how they peak, and so on.

Workload modeling and synthesis have always been important for stress-testing new production systems, as well as in academic research because of the difficulty of obtaining real commercial workload traces that might reveal confidential company information. The challenge of understanding and modeling spikes and the need to synthesize realistic "spiky" workloads motivate this work.

The term spike, or "volume spike," is commonly used to refer to an unexpected sustained increase in aggregate workload volume. In the case of stateless servers such as Web servers, such spikes can in principle be absorbed by a combination of adding more servers and using L7 switches, DNS, and geo-replication to redirect load to the new servers. Much work has focused on using both reactive and proactive approaches to automatically respond to spikes [Urgaonkar *et al.*, 2005b; Chase *et al.*, 2001; Kusic *et al.*, 2008; Tesauro *et al.*, 2006; Liu *et al.*, 2006; Stewart and Shen, 2005; Bennani and Menasce, 2005].

However, Internet-scale data-intensive sites—social networking (Facebook, Twitter), reference content (Wikipedia), and search engines (Google)—must also deal with "data spikes": a sudden increase in demand for certain objects, or more generally, a pronounced change in the distribution of object popularity on the site. Even data spikes that are not accompanied by an overall surge in volume may have severe effects on the system internally. For example, consider a partitioned database in which all objects are equally popular, but a sudden event causes 90% of all queries to go to one or two objects even though the total request volume does not increase.

Although volume spikes and data spikes can arise independently, it is becoming increasingly common for the two phenomena to occur together. For example, at the peak of the spike following Michael Jackson's death in 2009, 22% of all tweets on Twitter mentioned Michael Jackson [trendistic.com, 2010] (the largest Twitter spike in 2009), 15% of traffic to Wikipedia was directed to the article about Michael Jackson (see Section 2.3), and Google initially mistook the workload spike for an automated attack [Google, 2009]. The second and third most significant spikes on Twitter were Kanye West and the Oscars with 14.3% and 13.1% of all tweets, respectively [trendistic.com, 2010]. During the inauguration of President Obama, the number of tweets per second was five times higher than on a regular day [Twitter, 2009].

Workload spikes pose new challenges beyond those of modeling and stress-testing for two reasons. First, as mentioned previously, the effect of a data spike may be severe even if the aggregate workload volume changes little. Second, unlike stateless systems, simply adding more machines may not be enough to handle data spikes: data may need to be moved or copied onto the new machines. Deciding what to copy requires information not only about the distribution of object popularities during a data spike, but about the locality of accesses: if a group of objects becomes popular rather than a single object, are the popular objects grouped together on storage servers or scattered across the system?

Our approach consists of three steps:

1. An analysis of five spikes from four real workload traces, showing that the range of volume and data spike behaviors is more varied than one might expect (Sections 2.2 through 2.4);

2. A methodology for capturing both volume and data spikes with a simple seven-parameter statistical model (Sections 2.5 and 2.6);

3. A closed-loop workload generator that uses these models to synthesize realistic workload volume and data spikes (Section 2.7).

Analysis of five spikes from four real Web server traces. We identify seven important spike characteristics that suggest that volume spikes and data spikes are more varied than one might expect. The five spikes we analyzed represent a wide range of behaviors, including data spikes with little volume change, data spikes accompanied by volume spikes, and different distributions of object popularity and spatial locality during data spikes. The spike durations vary from hours to days, and the increases in workload volume vary from insignificant to factors of 5 or more. We also find that, in accordance with intuition, the onset shapes and peak values of volume spikes for "anticipated" events such as holidays or the Oscars differ substantially from those for unexpected events such as the death of Michael Jackson or the Balloon Boy hoax [CNN, 2009]. This suggests that "stress testing for spikes" (especially for stateful services) is more subtle and involves more degrees of freedom than previously thought.

A methodology for capturing volume and data spikes with a statistical model. We model the spikes in three steps. First, we model normal workload without spikes based on workload volume and baseline object popularity. Second, we add a workload volume spike with a particular steepness, duration, and magnitude. Finally, we select a set of hot objects with a particular spatial locality and increase their popularity. A particular novel aspect of our methodology is the identification and representation of *popularity distribution change* and *data locality* during a data spike:

(1) How does the distribution of object popularities change after spike onset and throughout the spike? What is the shape of this distribution—do the top N objects increase equally in popularity, or is there a significant tail?

(2) Given a simple representation of how data objects are mapped to storage nodes, what spatial locality is present in accesses to groups of popular objects during the spike? That is, when a group of objects increases in popularity, do accesses to the group tend to cluster on one or two servers, or do they require touching multiple servers across the installation?

Synthesis of realistic workloads from the model. We built a closed-loop workload generator that uses our models to synthesize realistic volume and data spikes. We show that by properly setting the model parameter values, we can generate any workload within the space defined by the seven spike characteristics we identified. In particular, we show that the total workload volume, workload volume seen by the hottest object, and workload variance generated by the model can be made to match those characteristics in the real traces we analyzed.

Our goal is not to create an intricate model that imitates every detail of the real traces, but a simple statistical model that is experimentally practical and captures the important characteristics of workload and data spikes. Note that the goal of the model is not to predict the *occurrence* of spikes, but to model the changes in workload volume and popularity of individual objects when spikes occur. Recalling the comparison to earthquakes, while we may not be able to predict individual earthquakes, it is useful to characterize general properties of earthquakes so as to provision and stress-test systems appropriately.

2.2 Methodology

Although obtaining real workload traces of public Internet sites is extremely difficult, we were able to obtain four traces and some partial data from a fifth source. Since our goal is to characterize data spikes as well as volume spikes, for each trace we identify the fundamental underlying *object* served by the site and extract per-object access information from the logs. For example, for a site such as Wikipedia or a photo sharing site, the object of interest can be represented by a static URL; for a site that serves dynamic content identified by embedding parameters in a common base URL, such as `http://www.site.com?merchant_id=99`, the object would be identified by the value of the relevant embedded parameter(s). Because we want to characterize sustained workload spikes lasting at least several minutes, we aggregate the traffic into five-minute intervals. Another reason for aggregation is low workload volume in some of the datasets.

In the rest of the chapter, *workload volume* represents the total workload rate during a five-minute interval. *Object popularity* represents the fraction of workload directed at a particular object. A *volume spike* is a large sustained increase in workload volume in a short period of time, while a *data spike* is a significant shift in popularity of individual objects. We provide no formal definition of volume and data spikes and identify the spikes manually by visual inspection. *Hotspots* or *hot objects* refer to objects in the system whose workload increases significantly; we provide a formal definition in Section 2.3.3.

2.2.1 Datasets

Table 2.1 summarizes our datasets and Table 2.2 summarizes the spikes.

World Cup 1998. Web server logs from week four of the World Cup 1998 [Arlitt and Jin, 1999]. The logs contain time of individual requests along with the file that was accessed. We use the path of the file as the name of the object.

UC Berkeley EECS Website. Web server logs from the UC Berkeley, EECS Department website hosting faculty, student, class web pages, technical reports and other content. The logs contain time of individual requests along with the file that was accessed. We use data from two events: increase in popularity of 50 photos on a student page and the Above the Clouds paper [Armbrust *et al.*, 2009a] being mentioned on Slashdot.org. We use the path of the file as the name of the object.

Ebates.com. Web server logs from Ebates.com used in [Bodík *et al.*, 2005]. The logs contain the time of individual requests along with the full URL. We use the value of the *merchant_id* URL parameter as the object. The merchants are represented as arbitrary integers.

Wikipedia.org. Wikipedia.org periodically publishes the hourly number of hits to individual Wikipedia articles [Wikipedia, 2007]. We use the individual articles as objects. While Wikipedia is one of the most popular Web sites, the disadvantage of this dataset is that it only contains hourly aggregates of workload instead of individual requests.

Partial sources of data from Twitter. We also use data from [Twitter, 2009], which contains plots of workload at Twitter during the inauguration of President Obama, and from [trendistic.com, 2010], which contains hourly popularity of the top 20 trends on Twitter in

Dataset	Granularity	Objects	# Objects	% Traffic to top 1% objects	% Traffic to top 10% objects
World Cup	requests	files	29,475	88%	99%
EECS website	requests	files	338,276	83%	93%
Ebates.com	requests	merchants	877	38%	82%
Wikipedia	hourly	articles	3,160,492	32%	67%
Twitter	aggr. workload	N/A	N/A	N/A	N/A
Twitter	top popularity	topics	N/A	N/A	N/A

Table 2.1: List of datasets and their various statistics

Dataset	Name of spike	Description
WorldCup	WorldCup	Spike at the beginning of a soccer match
EECS	Above-the-Clouds	Above the Clouds paper is mentioned on Slashdot.org
EECS	EECS-photos	Spike in traffic to about 50 photos on a student's page
Ebates	Ebates spike	Change in popularity of merchants in ad campaign
Wikipedia	Michael Jackson	Spike after the death of Michael Jackson
Twitter	inauguration	Inauguration of President Obama
Twitter	top 20 trends	Top 20 trends on Twitter according to trendistic.com

Table 2.2: List of spikes

2009. Because both of these are incomplete sources of data, we do not use them in our full analysis. However, they still demonstrate important properties of workload spikes.

2.3 Characterization of Real Spikes

In this section we characterize the important aspects of a workload spike. Since the change in workload volume and change in data popularity could occur independently, we analyze them separately.

2.3.1 Steepness of Volume Spikes

Figure 2.1 on the following page shows the increase in workload volume for each spike normalized to the beginning of the spikes.[1] We observe that the workload increase during the spikes varies significantly. The workload increased by a factor of almost four in the EECS-photos and WorldCup spikes, while it stayed almost flat in the Ebates, Above-the-Clouds, and Michael Jackson spikes. The steepness also varies; during the EECS-image and WorldCup spikes, the workload reached its peak in 20 and 60 minutes, respectively.

2.3.2 Static Object Popularity

Much has been written about objects' popularities[2] in the World Wide Web following a Zipf law [Wolman *et al.*, 1999; Padmanabhan and Qiu, 2000], which states that the popularity of the ith most popular object is proportional to $i^{-\alpha}$, where α is the power-law parameter. Power-law distributions are linear when object popularities are plotted against their ranks on log-log scale. Figure 2.2 on the next page shows the object popularity distribution for our four datasets along with a line fit to the data.[3]

If the data were from a power-law distribution, the dashed lines would coincide with the black curves. We observe that this occurs in none of the four datasets: either the most popular objects are not as popular as the power law predicts, or the tail of the distribution

[1] We do not have raw workload data for the Twitter spikes.

[2] Popularity of an object is the fraction of traffic directed at that particular object.

[3] We do not have popularity data for all the topics on Twitter.

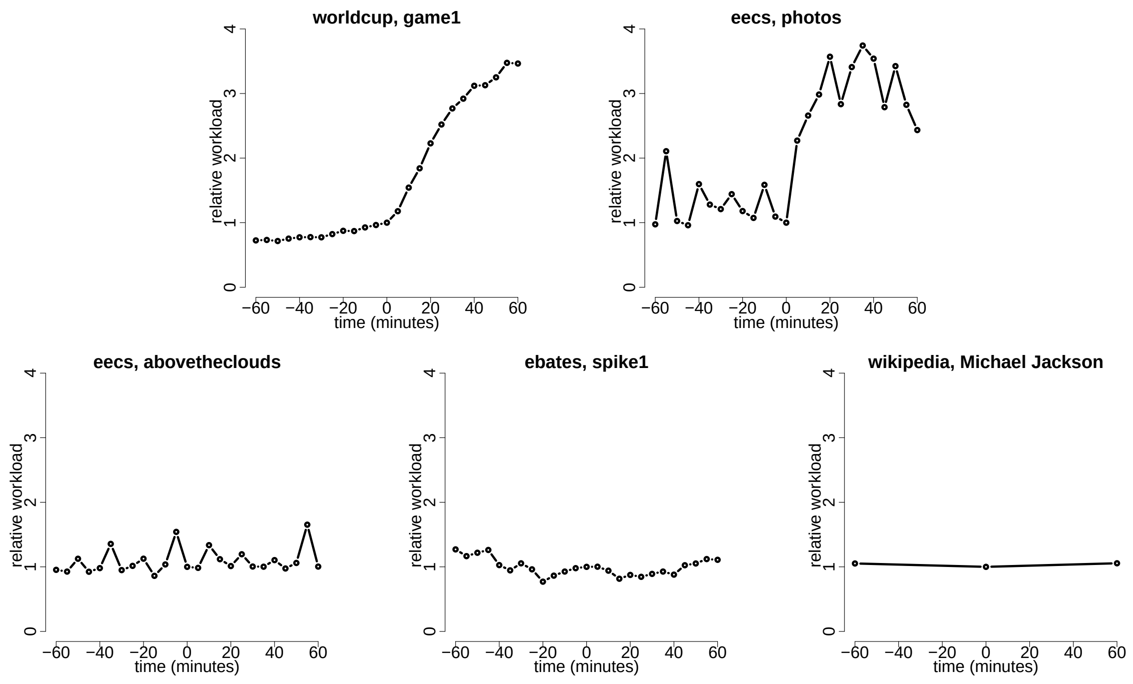


Figure 2.1: Workload volume profile of spikes normalized to the volume at start of spike at time 0. Notice that the increase in aggregate workload volume was very large in the two top spikes (up to 300%), while the workload remained almost flat in the three bottom spikes. Even though the aggregate workload did not change much during the bottom three spikes, there were several data hotspots with significant increase in workload (see Section 2.3.4).

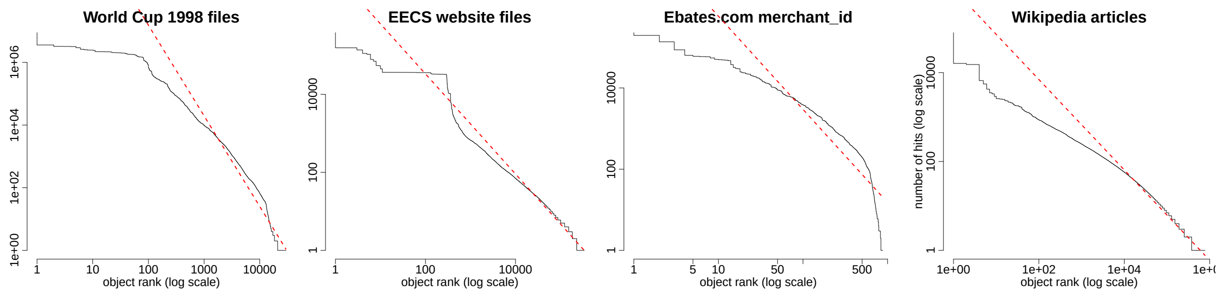


Figure 2.2: Object popularity distribution plotted on log-log scale along with the least-squares-fit Zipf curve (dashed line). Both axes are in log scale which means that there are many more points in the bottom-right corner of the graph. Those points thus significantly affect the slope of the line. X-axis represents the rank of an object, y-axis the number of hits to that object. The most popular objects receive significantly fewer hits than the Zipf curve would predict.

falls off much faster than the power law predicts. The EECS data[4] come closest, followed by the Wikipedia articles, while the Ebates and WorldCup datasets show the most significant

[4]The flat region in the EECS dataset corresponds to a set of faculty photos periodically displayed on a digital board; their popularities are thus almost identical.

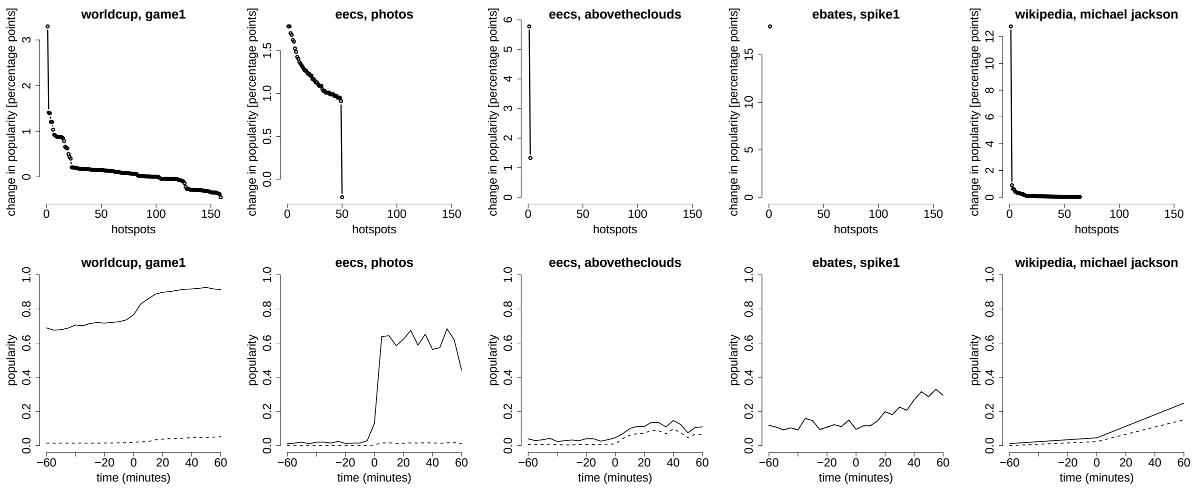


Figure 2.3: Top, difference in popularity of all hot objects. Bottom, popularity of the *single hottest* object (dashed) and all hot objects (solid) during spike.

differences.

Table 2.1 summarizes the fraction of traffic that corresponds to the most popular 1% and 10% of objects in all four datasets and shows significant differences among the datasets.

2.3.3 Detecting Data Spikes

To analyze the change in object popularity during a spike, we first define data *hotspots*. We consider an object to be a hotspot if the change of workload to this object during the first hour of the spike is significant compared to changes in workload during a *normal period* before the spike. We define the normal period to be one day before the spike, as it represents a typical interval during which various workload patterns repeat. We analyze the workload increase during the first hour of the spike since an hour is a long enough period to observe data hotspots and all of our spikes are longer than an hour.

More formally, let $\Delta_{i,t}$ be the change in workload to object i between time t and $t{+}1$ hour, and $\Delta_{max,t} = \max_i \Delta_{i,t}$ be the maximum change in workload to any object at time t. We define a threshold D as the median of $\Delta_{max,t}$ for times t during the normal period before the spike. Intuitively, D represents the typical value of maximum workload increase to a single object during the normal period. The median is a robust statistic that is not affected by outliers; using the median thus accounts for normal variability of the $\Delta_{max,t}$ statistic. We consider object i to be a hotspot if (and only if) $\Delta_{i,T_s} > D$, where T_s is the start of the spike. For each hotspot i, we define the change in popularity as the difference between the popularity of i one hour after the start of the spike and the popularity of i at the beginning of the spike. This definition implies that there could be data hotspots even during the normal period.

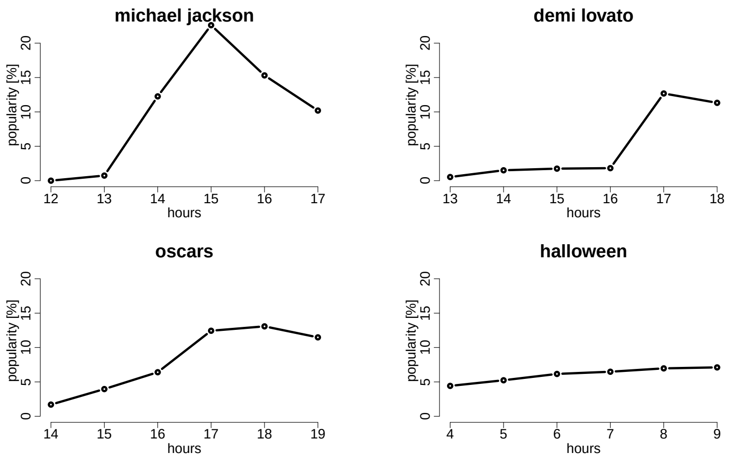


Figure 2.4: Topic popularity (percent) during 4 of the top 20 Twitter spikes. Spike steepness varies significantly.

2.3.4 Change in Object Popularity During Spikes

We now analyze the magnitude of changes in popularity of the hot objects during the spike and the temporal profile of these changes. The top row in Figure 2.3 shows the change in popularity of all the hot objects between the normal period before the spike started and during the spike. We observe a significant shift in popularity during the spikes. For example, the popularity of the Michael Jackson article increased by 13 percentage points during that spike. During the EECS-photo spike, the popularity of each of the 50 student photos increased by at least 1 percentage point.

We notice two important differences between the spikes. In the Ebates, Above-the-Clouds, and Michael Jackson spikes, the number of hot objects is very small, while the remaining two spikes have more hot objects. However, the number of hot objects is very small compared to the total number of objects in each dataset. Also, there are significant differences in the magnitude of change among the hot objects. In the Ebates and Above-the-Clouds spikes, a single hot object is responsible for most of the shift in popularity. However, in the EECS-photo and WorldCup spikes, the change in popularity is spread out more uniformly among the hot objects. Finally, the popularity of some of the hot objects actually decreased during a spike (WorldCup and EECS-photos spikes).

The bottom row in Figure 2.3 shows the popularity of the hottest object and the total popularity of all the hot objects. Figure 2.4 shows the popularity of the hottest topic in four of Twitter spikes.[5] Notice that the steepness of the popularity varies significantly.

2.3.5 Spatial Locality of Data Spikes

The response to data spikes is affected by the spatial locality of the hotspots. Some storage systems such as Amazon Dynamo [DeCandia *et al.*, 2007] handle data in blocks and can only replicate whole blocks. Other systems such as SCADS [Armbrust *et al.*, 2009b]

[5]We do not have popularity data for the inauguration spike.

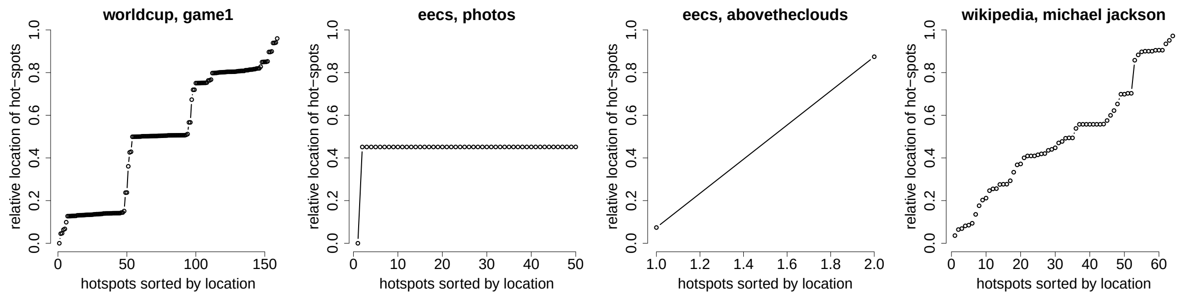


Figure 2.5: Spatial locality of hotspots. X-axis represents hotspots sorted lexicographically, y-axis shows their relative location. We do not show the Ebates spike which has only a single hotspot.

support range queries and thus store the objects in logical order. In such systems, if all the hot objects live on a single server or are logically close to each other, one could react to a spike by replicating a very small fraction of the data. However, if the hot objects are spread over many servers, one would have to copy data from multiple servers.

While we do not know how the individual objects are stored, to evaluate the spatial locality of data hotspots, we order them lexicographically by their name. This ordering preserves the logical structure of the objects; for example, files in the same directory will stay close to each other. Figure 2.5 shows the spatial locality of the hotspots. The x-axis represents the individual ordered hotspots and the y-axis represents the relative location of a hotspot in the range of all objects. Flat regions in these graphs thus represent hot objects located very close to each other.

We observe that the WorldCup and EECS-photos spikes have significant spatial locality. In the WorldCup spike, we notice three large clusters of hotspots. In the EECS-photos spike, all the objects except one form a contiguous range.

2.3.6 Spike Characterization Summary

Before presenting our proposed quantitative characterization of spikes, we summarize the key observations emerging from our analysis:

There is tremendous variation in volume and data spikes. The duration of the spikes varies from hours to days, the increase in workload volume ranges from insignificant increases to factors for five, and the slope of workload volume and popularity of hottest objects also varies drastically. This supports the claim that methods for automatic scaling of web applications cannot be tested on a single workload, but rather must be evaluated across a range of different and realistic spikes.

Object popularity is not necessarily Zipfian. The popularity distributions tend to be heavy-tailed, but there are significant differences from power-law distributions. In our workload model, we use Pitman-Yor stick-breaking process to generate popularity distributions with tails that "fall-off" faster than power-law tails (see Section 2.5.1 on page 16).

In all spikes, the number of hot objects is a small fraction of the total number of objects in the system. We believe this is because the workload and spikes are generated by human users whose actions are independent of the total size of the dataset. In other words, even if Wikipedia had ten times more articles, Michael Jackson's death would likely still result in Wikipedia users visiting just the single article, not ten times as many articles. Therefore, we characterize the number of hot objects as an absolute number instead of a fraction of all objects.

Anticipated spikes are both gentler and smaller than unanticipated spikes. In the top 20 Twitter trends, *surprising events* shoot up quickly from zero to peak, exhibiting 1-hour changes in popularity close to the maximum popularity. Examples include celebrity deaths (Michael Jackson, Patrick Swayze, Brittany Murphy, and Billy Mays) or other unexpected sensations (Nobel Prize for President Obama or the Balloon Boy). On the other hand, *anticipated events* such as holidays or other well known events (the Oscars, Eurovision) slowly build up their popularity.

2.4 Quantitative Characterization

In this section, we define a set of quantitative metrics to describe the important aspects of a workload and data spike based on observations in Section 2.3. We present the results in Tables 2.3 and 2.4. Here are the definitions of the metrics:

Number of hot objects represents the number of objects identified as hotspots (see Section 2.3.3).

Normalized entropy of the popularity of hot objects represents the skewness of the change in popularity of hot objects. Letting $c_1, \ldots, c_n$ denote the change in popularity of n hot objects (top row in Figure 2.3 on page 10), the normalized entropy H_0 is defined as follows:

$$H_0 = -\sum_{i=1}^{n} c_i \log_2 c_i / \log_2 n, \quad 0 \leq H_0 \leq 1.$$

H_0 close to 1 corresponds to distributions where all the hot objects have almost equal increase in popularity, such as in the EECS-photos spike (top row, column 2 in Figure 2.3). H_0 close to 0 corresponds to distributions where a single hot object has most of the popularity, such as the Michael Jackson spike (top row, column 5 in Figure 2.3).

Time to peak represents the time from the start to the peak of the spike. We first manually identify the start of the spike by visually inspecting the workload profile and changes in popularity of the hot objects. Then we find the peak time as time when the object with the most significant change in popularity (object h) reached the maximum popularity.

Duration of the spike is the time between the start of the spike and its end. We define the end of a spike to be the time when the popularity of object h returns to the level seen during the normal period before the spike. We do not use the workload volume to identify the peak and the end of the spike since in some cases the increase in workload volume is not large enough and using it would be unreliable. However, the popularity profile of object h is a clear signal of the peak and end of spikes.

Parameter	**World Cup**	**Above the Clouds**	**EECS photos**	**Ebates**	**Michael Jackson**	**inaugu-ration**
number of hot objects	159	2	50	1	64	N/A
0.1%-spatial locality	0.64	0.0	0.98	n/a	0.25	N/A
normalized entropy	0.76	0.69	0.99	0.0	0.48	N/A
time of peak [min]	105	40	55	130	60	N/A
duration [min]	190	800	1000	7000	>1560	N/A
relative increase in aggr.	4.97	0.97	2.43	1.13	1.05	≈ 5
max slope [%/min]	8.7	7.0	25.4	2.9	0.09	≈ 140

Table 2.3: Characterization of spikes using metrics described in Section 2.4. The metrics for the inauguration spike were estimated from a post on Twitter's blog.

Relative increase in workload volume represents the workload volume increase at the peak of the spike relative to the workload at the start of the spike.

Maximum slope captures the increase in workload volume during the steepest 5-minute period during the spike. We first normalize the workload to the start of the spike, find the steepest part, and represent its slope as change in relative workload per minute. Thus, a max slope of 7% per minute means that during the steepest part of the spike, the workload volume was increasing by 7% every minute (relative to the workload at the start of the spike). We note that this only captures the slope of the workload volume, not the slope of workload to the individual hot objects.

Spatial locality represents the degree to which the hot objects are clustered. Let $l_1, \ldots, l_n$ represent the relative location of hot objects in the logical address space of all objects sorted lexicographically. For a given value of δ, we consider two objects i and j to be *close* if $|l_i - l_j| < \delta$. Finally, let C_δ be the minimal number of clusters of hot objects, such that each cluster contains only objects that are close. Intuitively, if we need few clusters to cover all hot objects, the hot objects exhibit high spatial locality, and vice versa. We thus define spatial locality as

$$L_\delta = 1 - \frac{C_\delta - 1}{n - 1}, \quad \text{so } 0 \leq L \leq 1.$$

In Table 2.3, we report values for $\delta = 0.1\%$.

Change in popularity in one hour is the biggest change in popularity of the hottest object during one hour in percentage points (pp); i.e., a change from 2% to 10% is 8pp.

Maximum popularity represents the peak popularity of the hottest object achieved during the spike.

Together, these metrics capture the most important aspects of a spike. The number of hot objects, normalized entropy, change in popularity, and maximum popularity characterize the hot objects, while the rest of the parameters characterize the changes in workload volume. We present partial results for data from [Twitter, 2009; trendistic.com, 2010]. For the Obama inauguration spike we only have approximate values of the relative increase in workload volume, while for the top 20 trending topics we only have the changes in their

Spike	**Change in popularity in 1 hour [percentage points]**	**Maximum popularity [percent]**
WorldCup	1.14	1.53
EECS photos	1.47	1.48
Above the Clouds	5.77	6.42
Ebates	12.50	55.63
Michael Jackson	12.76	15.12
michael jackson (twitter)	11.52	22.61
kanye	12.64	14.39
oscars	6.03	13.08
balloon (boy)	11.76	12.89
demi lovato	10.86	12.68
thanksgiving	2.91	12.09
jonas brothers	10.42	12.06
fireworks (July 4th)	5.22	11.72
eurovision	5.36	9.93
happy easter	1.96	9.78
patrick swayze	9.67	9.67
aplusk	5.16	8.50
obama (nobel)	8.09	8.39
iphone	5.80	8.27
facebook	6.03	8.09
brittany murphy	7.60	8.01
lakers	4.47	7.56
yankees	4.62	7.14
halloween	0.92	7.11
billy mays	5.09	5.85

Table 2.4: Summary of change in popularity and maximum popularity of the single most popular object in each spike. The first five spikes correspond to spikes described in Table 2.2. The following 20 spikes are based on hourly data from the Top 20 Trending Topics of 2009 on Twitter.

popularity over time and not the workload volume.

2.5 Workload Model

In this section, we build on a baseline workload model for normal periods without any spikes or data hotspots and extend it to support workload spikes and changes in popularity of objects. Then, we extend the model to support a notion of data locality. We describe the parameters of the model, explain how they control the various spike characteristics (see Table 2.5), and show that our model can generate workload similar to the spikes described in Section 2.3.

We emphasize that it is not our goal to create a complex workload model that imitates every detail of the real traces analyzed in Section 2.3. Instead, our goal is a model that is simple to use and captures the important aspects of volume spikes and data spikes that emerged from our analysis. To this end, we use the Pitman-Yor stick-breaking process [Pitman and Yor, 1997] to generate popularity profiles, the Dirichlet distribution to model popularities of data hotspots, and the Chinese Restaurant Process [Aldous, 1985] to generate locations of hotspots (clusters). In the following sections we explain, justify, and validate these choices.

2.5.1 Model of Normal Workload

A model of normal workload consists of a workload volume time series $U = (u_1, \ldots, u_T)$ and a popularity profile for individual objects. We do not focus on the workload volume time series U, as this can be based on other studies of Web workloads, such as [Arlitt and Jin, 1999]. Instead we concentrate on the novel challenge that arises in data spike modeling: creating a realistic characterization of object popularity. We note at the outset that the popularities, while drawn from a distribution, are held constant during workload generation. The transformation from popularity profile to workload, however, is also stochastic, and this stochastic process is responsible for capturing the variance of the workloads of individual objects.

We showed in Section 2.3.2 that the popularity profiles in our datasets do not tend to follow the Zipf law often assumed for Web data. Therefore, we instead use *stick-breaking*, a probabilistic modeling method used to generate heavy-tailed distributions. The idea is straightforward—one starts with stick of unit length and breaks off a piece according to a draw from a beta distribution. This procedure is applied recursively to the remaining segment of the stick. The result is a set of increasingly small pieces of the stick whose lengths are taken to represent the popularity profile of our objects.

The particular form of stick-breaking we use is the *Pitman-Yor process* [Pitman and Yor, 1997]. This process, denoted $PY(d, \alpha)$, is parameterized by a *discount parameter* d, $0 \leq d < 1$, and a *concentration parameter* $\alpha > -d$. To generate the popularity profile of n objects, we first generate n beta random variables as follows:

$$\beta_i \sim \text{Beta}(1-d, \alpha + id), \quad i = 1, \ldots, n, \quad 0 \leq \beta_i \leq 1.$$

Next, we set the length of the first piece of stick as $\pi_1 = \beta_1$, and set the lengths of the remaining pieces as follows:

$$\pi_i = \beta_i \prod_{l=1}^{i-1} (1 - \beta_l), \quad i = 2, \ldots, n$$

Intuitively, the length π_i of the i'th piece is the length of the stick after breaking off the first $i-1$ pieces, multiplied by β_i. The lengths of the pieces generated by this procedure follow a power-law distribution with parameter $1/d$ in expectation. However, an individual

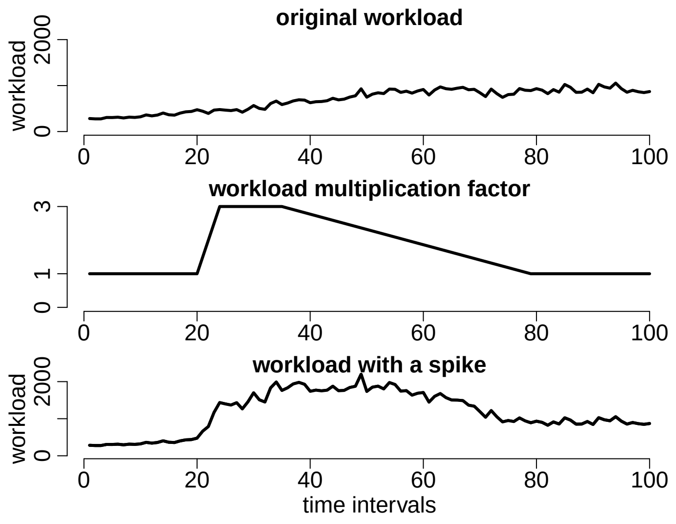


Figure 2.6: Top, an example of a normal workload profile without a spike. Center, workload multiplication factor c_t with the following parameters: $t_o = 20$ (onset), $t_p = 25$ (peak), $t_d = 35$ (decay), $t_e = 80$ (end), $M = 3.0$. Bottom, the workload profile during a spike is obtained by multiplying the normal profile (top) by c_t (center), point by point.

stick-breaking realization generates π_i's whose lengths drop off faster than a power law. This behavior more closely matches the behavior we observed in real workload traces in Figure 2.2 on page 9.

Our object popularity model is parameterized by a that controls the rate at which the object popularities drop off. This parameter is similar to the shape parameter of the Zipf distribution. To generate a popularity profile for n objects with parameter a, we first generate a vector $(\pi_1, \ldots, \pi_n)$ using the $PY(1/a, 0.5)$ process, and we then set the object popularities, $B = (b_1, \ldots, b_n)$, as a random permutation of the π_i's.

2.5.2 Modeling Volume Spikes and Data Spikes

To add a spike to a normal workload, we need to increase the workload volume during the spike and create a new object popularity distribution with a higher popularity for the hotspots. This process is parameterized by the following parameters: t_o (onset), t_p (peak), t_d (decay), and t_e (end) represent the times when the spike starts, when it reaches its peak, the end of peak period with flat workload, and the end of the spike, respectively. M represents the magnitude of the spike, or relative increase in workload volume. N represents the number of hotspots and V the variance of hotspot popularity. We first describe how we adjust the workload profile and then the object popularity during a spike.

The four t_i parameters and the magnitude parameter define the change in the workload profile. The workload profile is multiplied by a factor c_t to obtain the new workload profile during a spike. c_t is 1.0 for times t before t_o, increases linearly between t_o and t_p up to M, stays flat between t_o and t_d, decreases linearly to 1.0 between t_d and t_e, and is 1.0 after t_e (see Figure 2.6). We model the change in workload volume using a piece-wise linear factor to keep the model simple and minimize the number of parameters. Figure 2.1 on page 9 also justifies this decision as most of the workload profiles can be approximated well using a piece-wise linear function.

To generate object popularity during a spike, we adjust the baseline popularity $B = (b_1, \ldots, b_n)$ by putting more weight on the hotspots. In particular, we construct a vector $H = (h_1, \ldots, h_n)$ that represents the popularity of only the hotspots during a spike. The final object popularity profile at time t is $P_t = (1 - c_t^*)B + c_t^* H$—a weighted average of the baseline popularity and the hotspot popularity. P_t is an n-vector that represents object popularities at time time t. For times t before and after the spike, $c_t^* = 0$, so that the object popularity will be the baseline B. During the spike, c_t^* is adjusted in a piece-wise linear fashion (similar to c_t) such that at the peak of the spike $c_t^* = (M-1)/M$. This guarantees that as the workload volume increases by a factor of M, all the additional traffic will be directed towards the hotspots using distribution H. For example, for $M = 3$, 2/3 of the workload at the peak will be directed to hotspots.

We generate the popularity of hotspots H as follows. We first pick locations $l_1, \ldots, l_N$ of N hotspots uniformly at random; H has non-zero popularity only at these locations. The values of popularity of the hotspots, $h_{l_1}, \ldots, h_{l_N}$, are obtained as a sample from a Dirichlet distribution: $(h_{l_1}, \ldots, h_{l_N}) \sim \mathrm{Dir}(\alpha, \ldots, \alpha)$. A sample from a Dirichlet distribution results in N non-negative numbers that sum to 1 and thus represent valid object popularities. We use the Dirichlet distribution also because we can control the resulting normalized entropy with a single parameter.

As described in Section 2.3, different spikes have different values of normalized entropy of the hot object popularity distribution. In the workload model, we control the entropy of the hot object popularity distribution by adjusting the variance of the Dirichlet distribution.

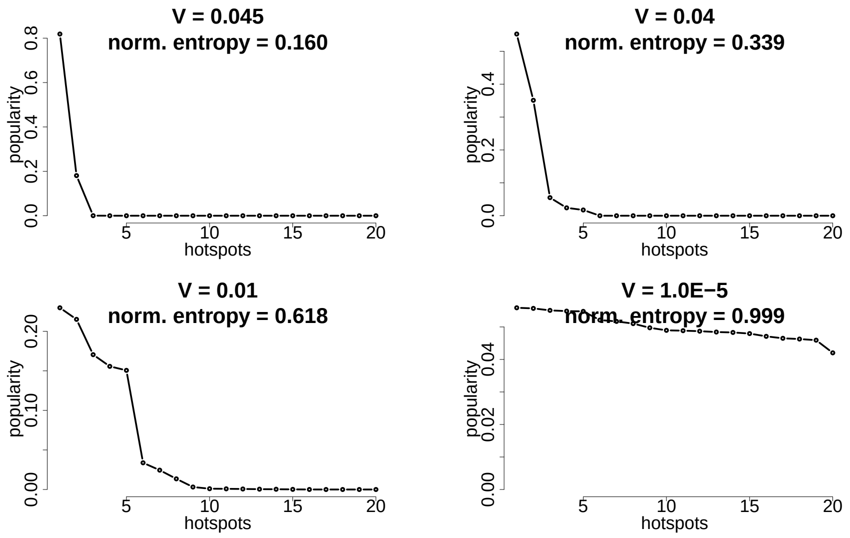


Figure 2.7: Popularity of $N = 20$ hot objects for different values of the variance parameter V resulting in different values of entropy.

We set the parameters of the Dirichlet distribution as follows:

$$\alpha = \frac{N - 1 - VN^2}{VN^3}.$$

With this setting of the α_i parameters, the expected value of all h_{l_i} is equal to $1/N$ and variance is equal to V; $\mathrm{E}[h_{l_i}] = 1/N, \mathrm{Var}[h_{l_i}] = V$. High variance V results in a distribution with high entropy, while low variance results in low entropy, which allows us to control the normalized entropy of the hot objects. Figure 2.7 on the previous page illustrates this process.

2.5.3 Adding Spatial Locality to Data Spikes

The model of a spike described in Section 2.5.2 picks the locations of the hot objects uniformly at random. However, as we demonstrated in Section 2.3.5, hot objects often exhibit significant spatial locality. Here we describe a simple two-step process that clusters the locations of the hotspots. The process is parameterized by L that controls the spatial locality of the resulting hotspots.

We first create clusters of hotspots using an iterative process known as the Chinese Restaurant Process (CRP), a statistical model of clustering that is parameterized by a single parameter $0 \leq L \leq 1$ [Aldous, 1985]. We start with a single cluster that contains the first hotspot. All the subsequent hotspots are either assigned to a new cluster, with probability proportional to $1/L - 1$, or pick an existing cluster i, with probability proportional to k_i, which is the number of hotspots in cluster i. The parameter L thus determines the number of clusters; in particular, large values of L imply a low probability of starting a new cluster

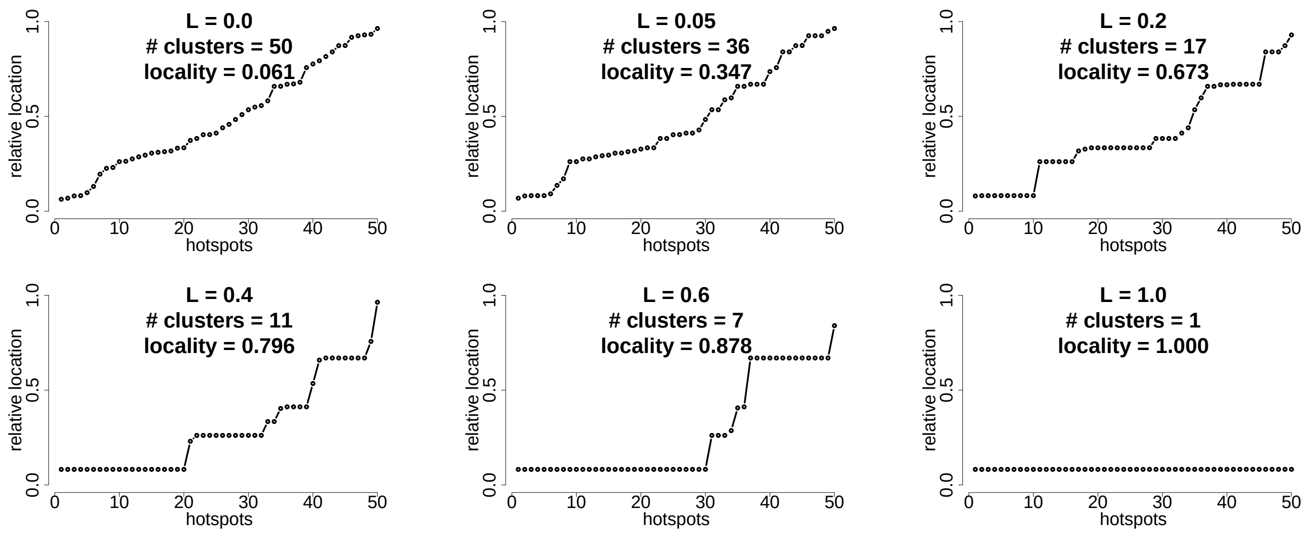


Figure 2.8: Locations of hot spots generated using a Chinese Restaurant Process. The title of each plot shows the L parameter used, number of clusters, and the resulting 0.1% spatial locality characteristic.

and thus a smaller number of clusters. Given N hotspots, the expected number of clusters grows logarithmically as $O((1/L - 1)\log N)$.

Second, we pick the location of cluster i, l_i, uniformly at random from all the available objects, and mark objects $l_i, \ldots, l_i + k_i - 1$ as hotspots. After selecting the locations of hot objects, we assign their probabilities as described in Section 2.5.2. Figure 2.8 on the preceding page shows sample locations of hot objects for different values of the locality parameter L. Since CRP is a stochastic process, the cluster sizes vary across multiple runs with the same L parameter. If we wish to control the variance of the cluster sizes we can run the CRP multiple times and average the sizes of clusters or simply use the expected values of the cluster sizes.

2.5.4 Workload Model Summary

To model a workload without spikes, we select the number of active users at each point in time, $U = (u_1, \ldots, u_T)$, from a section of an existing workload trace. The popularity of individual objects, b_i, is constructed using the Pitman-Yor stick-breaking process. We add a volume spike by multiplying the number of active users by a piece-wise linear function (Figure 2.6 on page 17). Finally, we add a data spike by selecting N hotspots with a particular spatial locality and entropy.

In Section 2.4 we presented seven important characteristics that define a space of spikes. We designed the model such that for any combination of the characteristics, there exist model parameters that generate a spike with such characteristics. Table 2.5 on the following page summarizes the model parameters. For most of the parameters, the effect on the various spike characteristics is straightforward. In particular, the magnitude and number of hot objects are directly the spike characteristics. The spike parameters $t_o, t_p, t_d,$ and t_e determine the maximum slope and duration. As for the normalized entropy and the spatial locality of the hotspots, these are controlled by the V and L parameters. As we show in Figures 2.7 and 2.8 on the previous page, there is a mapping between these quantities such that we can control the normalized entropy and spatial locality of the hotspots by the choice of V and L. In summary, for any values of the spike characteristics (see Table 2.3), we can find model parameters that generate such a spike.

2.6 Model Validation

We validate our workload model in two ways. First, we compare the variance of workload to individual objects in real traces with the variance obtained using our model. Second, we compare the workload volume and workload to the hottest object observed during the EECS-photos spike with workload generated from our model.

2.6.1 Short-term volatility of object popularity

As described above, the popularity of each object during the period before the spike is constant. One might be concerned that this implies that the actual workload received by

Parameter	Constraint	Description	Spike characteristic	Section
n	$n > 0$	number of objects	-	2.5.1
U	$u_i \geq 0$	workload profile	-	2.5.1
a	$a > 1$	power-law shape	-	2.5.1
N	$0 < N \leq n$	number of hotspots	number of hot objects	2.5.2
V	$0 < V < \frac{N-1}{N^2}$	popularity variance	normalized entropy	2.5.2
t_o, t_p, t_d, t_e	$t_o \leq t_p \leq t_d \leq t_e$	spike parameters	onset, peak, decay, end	2.5.2
M	$0 < M$	magnitude of spike	increase in workload	2.5.2
L	$0 \leq L \leq 1$	spatial locality	0.1%-spatial locality	2.5.3

Table 2.5: Parameters of the workload model, their description, the corresponding spike characteristics that they control, and the section that describes the parameter in more detail. The first three parameters model the *normal* workload and thus do not correspond to any spike characteristic. The spike model has seven main parameters (N, V, t_p, t_d, t_e, M, and L) and the start of the spike (t_o) that has no effect on spike characteristics.

object i over time will either be constant or have variance much smaller than observed in real traces. Here we demonstrate that the variance of workload generated by our model is comparable to variance in real workloads.

Generating k requests to n objects with popularities of $B = (b_1, \ldots, b_n)$ is equivalent to taking k samples from a multinomial distribution Mult(B). The expected value and variance of the number of hits to object i, w_i, can be expressed as follows: $\mathrm{E}[w_i] = kb_i$, $\mathrm{Var}[w_i] = kb_i(1 - b_i)$. In Figure 2.9 on the following page, we compare the variance observed during a normal workload period before the WorldCup spike with variance of a multinomial distribution. We see that the variance of a multinomial distribution closely matches the variance of most of the objects.

2.6.2 Comparing real and generated workload

To compare real and generated workload during the EECS-photos spike, we set the model parameters such that the spike characteristics match values presented in Table 2.3 and generate requests to objects based on the workload volume. In Figure 2.10 we compare the workload volume and the workload to the hotspot with the largest increase in workload during the spike. We note that while the generated curves do not exactly match the observed workload, the overall profile is very similar. We reiterate that our goal was not to replicate every detail of the real workloads, but only to capture the most important aspects of the spikes.

2.7 Workload Generation

We built a closed-loop workload generator [Schroeder *et al.*, 2006] in which a single user is simulated by a single client thread that creates and executes requests in a loop, a common design for workload generators [Faban, 2009; Rubis, 2008]. Each thread selects a request

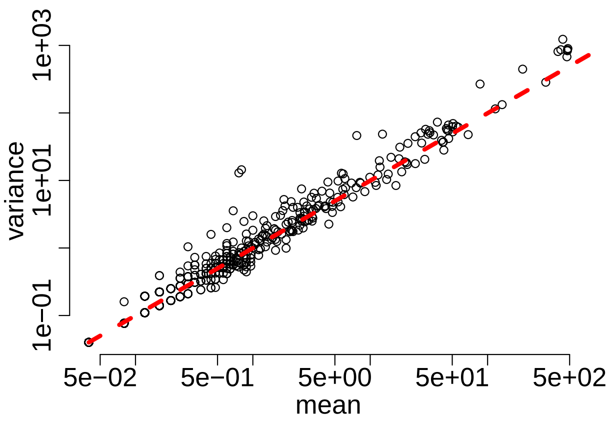


Figure 2.9: Mean vs. variance of workload to randomly selected 1000 objects in the World-Cup dataset during a two hour normal workload period in log-log scale. Each circle corresponds to a single object, the x-axis shows the mean workload to the object, the y-axis shows the variance of workload. The dashed (red) line represents the variance of these objects when they are sampled from a multinomial distribution.

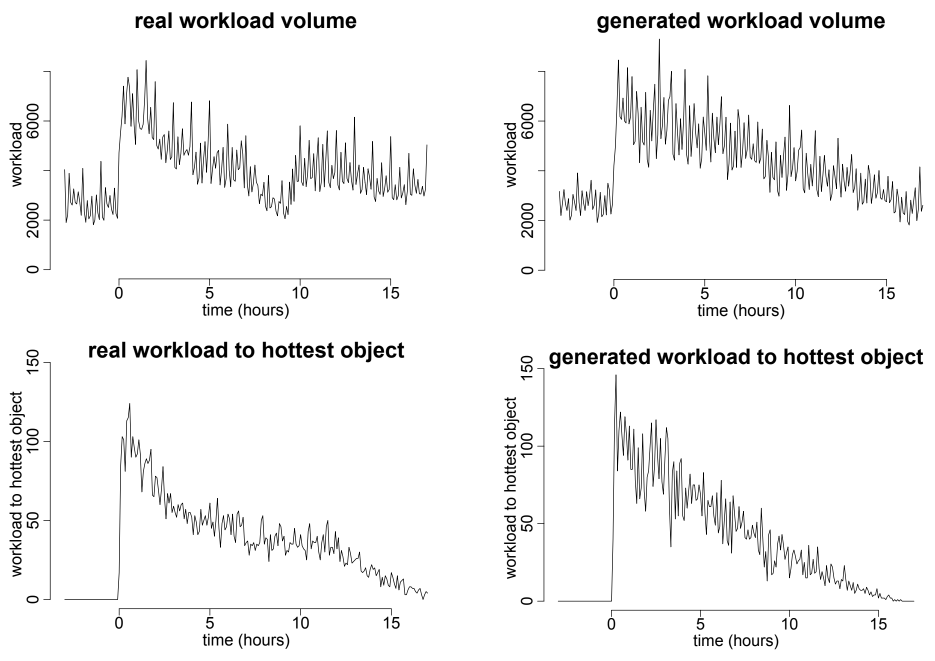


Figure 2.10: Comparison of real and generated workload for the EECS-photos spike. Top, actual and generated workload volume. Bottom, actual and generated workload to the hotspot with largest increase in workload.

type (such as read or write), selects request parameters using our model, sends the request to the system under test, waits for a response, and repeats.

We discretize time into T short intervals to create the *volume profile*, a time series $U = (u_1, \ldots, u_T)$ in which u_t is the number of users active during interval t. Each thread stores a copy of this profile; to simulate changes in the number of active users, a thread sleeps during interval t if its thread ID is larger than u_t. Otherwise, at time t, the workload generator selects object i with probability $p_{i,t}$.

We remark that while open-loop workload generation would allow specifying a particular workload rate (e.g., 2000 requests per second), a closed-loop workload generator only allows us to specify the number of active users; the resulting workload rate depends on the number of users, their think times, and the latency of the individual requests. For example, when 100 active users generate requests that take 100ms to execute, the workload rate would be 1000 requests per second. If the request latencies drop to 10 ms, the workload rate would increase to 10,000 requests per second.

Initializing the model for the experiment in Section 2.6 with 338,276 objects and 50 hotspots took 14.5 seconds. Generating 20 hours of workload took 2.1 seconds.

2.8 Related Work

Web server workloads have been thoroughly studied in many papers, such as [Arlitt and Jin, 1999; Chen and Zhang, 2003]. Most of these papers, however, analyze workload for stateless Web servers in context of caching, prefetching, or content distribution networks. While some papers study surges, spikes, or flash crowds [Schroeder and Harchol-Balter, 2006; Jung *et al.*, 2002], they concentrate only on the increase in workload volume to the Web server.

Most of the workload generators [Apache, 2010c; Barford and Crovella, 1998; Httperf, 2009; Faban, 2009; Rubis, 2008] used to evaluate computer systems or Web sites do not support either volume or data spikes. Httperf [Httperf, 2009] provides a simple tool for generating http workload for evaluating Web server performance. Httperf, however, generates requests only at a fixed rate specified by the user. Rubis [Rubis, 2008] is a Web application benchmark with a workload generator that uses a fixed number of clients and thus cannot generate a volume spike. Faban [Faban, 2009] is a flexible workload-generation framework in which volume and data spikes could be implemented, but are not supported out of the box.

Similar to our notion of data spikes is the concept of *temporal stability* of workload patterns in a Web server described in [Padmanabhan and Qiu, 2000]. The authors analyze Web server traces from a large commercial web site mostly in the context of caching. They define temporal stability as the overlap of the top N most popular pages during two days and find that the stability is reasonably high on the scale of days. While the idea of popularity of the top N pages is similar to data spikes, we are interested in changes on time scale of minutes, not days or weeks. Also, the authors do not provide a model or a workload generator to simulate such changes.

The authors of [Mi *et al.*, 2009] propose a methodology and a workload generator to introduce burstiness to a stateless Web server workload. The authors characterize burstiness using the *index of dispersion*—a metric used in network engineering. The workload-generating clients are in one of two states—high or low—with transitions between the two states governed by a simple Markov process. The clients in high state generate requests at high frequency and vice versa, thus introducing burstiness into the workload. However, the proposed model does not increase popularity of individual objects and one cannot easily change the steepness and magnitude of the bursts.

The papers [Gulati *et al.*, 2009; Kavalanekar *et al.*, 2008] characterize workload for storage systems at disk-level, but do not analyze volume or data spikes. Finally, many researchers in the networking community have focused on characterizing and modeling *self-similarity* in network traffic [Crovella and Bestavros, 1996] (burstiness over a wide range of time scales) or anomalies in network traffic volume [Lakhina *et al.*, 2004]. All of these efforts, however, concentrate on workload volume and not on data popularity.

2.9 Future Directions

In this chapter, we characterized several workload spikes and concluded that they differ significantly in many important characteristics. We also described a workload model that allows us to generate spikes with a wide range of characteristics and stress-test Web applications against such spikes. More work is required to understand typical spike patterns and their frequencies. Below, we present two conjectures based on initial analysis of spike magnitudes and frequencies, and also describe possible extensions to our spike model.

2.9.1 Two Conjectures

Below we present two conjectures based on initial analysis of spike magnitudes and frequencies. We studied the daily change in page popularities on Wikipedia during one month and treated spikes on individual pages during one day as separate events. The *spike magnitude* represents the increase in popularity of a particular page.

Spike magnitudes follow Zipf's law. Figure 2.11 on the next page shows that the spike magnitudes on Wikipedia during one month follow Zipf's law; there are many spikes with small magnitude and few spikes with large magnitude. By fitting the distribution (red line in Figure 2.11), we could predict the frequency of spikes of larger magnitudes. The accuracy of such predictions remains to be evaluated.

More popular objects are more likely to become hotspots. In Figure 2.12, we first bin Wikipedia pages based on their popularity one day before the spike and compute the empirical probability of observing a spike of certain magnitude. We see that the more popular pages have higher likelihood of getting a larger spike.

2.9.2 Extending the Model

Longer-term trends in object popularity. As demonstrated above, the variance in workload to individual objects on a time scale of five minutes is well modeled by a multinomial distribution. However, on longer time scales (hours), we notice slow upward and downward trends in workloads to many objects. We can simulate these trends by perturbing the object popularity using the Dirichlet distribution. Let $(b_1, \ldots, b_n)$ be the baseline popularity of all objects in the system and let $(p_{1,t}, \ldots, p_{n,t})$ be the actual popularity of objects used at time t. To simulate trends in popularity that change every T minutes, we update $(p_{1,t}, \ldots, p_{n,t})$ every T minutes by sampling from the following Dirichlet distribution: $\mathrm{Dir}(Kb_1, \ldots, Kb_n)$. For any value of K, the expected value of p_i is b_i, which means that the sampled values of $p_{i,t}$ will oscillate around the baseline popularity b_i. However, the value of K affects the variance of $p_{i,t}$; smaller values of K imply larger variance and thus more significant trends in object popularity and vice versa (see Appendix).

Other extensions. The Chinese Restaurant Process could be used to add spatial locality even to the baseline object popularity b_i by first creating some number of clusters and then assigning the objects with the largest popularity to these clusters. Second, the current model assumes that only the additional workload (with magnitude M) is directed at the hotspots. By increasing the values of c_t^* (see Section 2.5.2), we can redirect more workload at the hotspots. Finally, we can easily create mixed spikes by creating new hotspots while other spikes are still in progress.

2.10 Summary

As significant workload spikes are becoming the norm for popular Web sites such as Facebook and Twitter, it is important to evaluate computer systems using workloads that resemble these events. In many cases a surge in workload occurs together with the emergence of data hotspots, which has a crucial impact on stateful systems. Because it is very difficult to obtain realistic traces of such events, it is important to model and synthesize workloads that contain realistic spikes in workload volume and changes in data popularity.

In this chapter, we presented a methodology for characterizing the most important as-

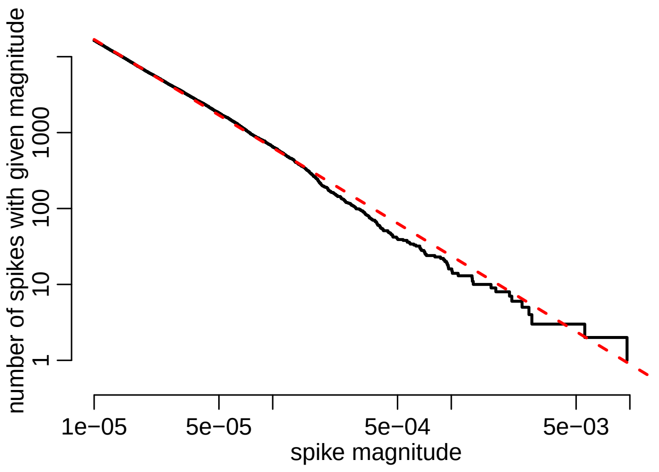


Figure 2.11: Spike magnitudes on log-log scale based on one month of Wikipedia data (solid line) and linear fit to the data (dashed line).

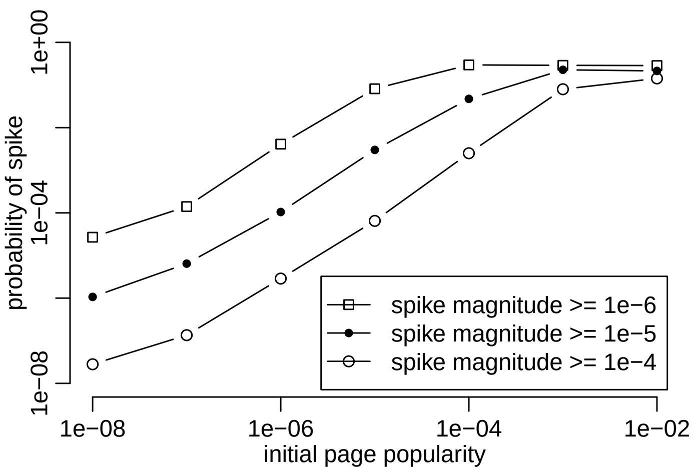


Figure 2.12: Probability of a spike of certain magnitude (10^{-6}, 10^{-5}, and 10^{-4}) for pages of different popularities on log-log scale.

pects of spikes in stateful systems. We characterize a spike in terms of changes in workload volume (maximum slope, relative increase, time to peak, and duration) and changes in data popularity (the number of hot objects, spatial locality, and normalized entropy).

We use this methodology to analyze five spikes in four datasets. We observed two important facts. First, the number of hotspots in all the spikes is a very small fraction of the total number of objects in the system. Second, the spikes differ dramatically in all of the other characteristics. This suggests that there is no “typical” workload spike and computer systems should be evaluated using a wide range of spiky workloads.

Our workload model uses a small number of parameters to capture the most important aspects of spikes in stateful systems using well-known probability distributions and generative processes. We first model a typical workload with no spikes using a workload profile and a distribution describing the object popularity. Second, we add a volume spike by increasing the workload and add a data spike by increasing popularity of a small number of objects. Finally, we add a spatial locality of hotspots using a simple clustering process. The workload model serves as input to a closed-loop workload generator, where simulated clients select objects based on the generated object popularity that evolves over time.

Workload spikes can overload a Web application and cause performance degradation or downtime. Cloud Computing enables dynamic resource allocation in response to changes in workload such as during spikes. In the next chapter, we propose a control framework that uses a performance model of an application to minimize the number of deployed servers and maintains a certain level of performance.

Chapter 3

Director: Framework for Dynamic Resource Allocation in Stateful Systems

3.1 Introduction

A key benefit of cloud computing is the ability to dynamically provision compute resources in response to changes in application workload. Such elasticity can avoid overprovisioning and thus significantly reduces server cost. While there has been a lot of work on dynamic resource policies for stateless systems [Chase *et al.*, 2001; Urgaonkar *et al.*, 2005b; Liu *et al.*, 2006; Kusic *et al.*, 2008; Hellerstein *et al.*, 2008; Tesauro *et al.*, 2006], such as web or application servers, little progress has been made for stateful systems.

However, an equally important goal is to maintain strict performance Service-Level Objective (SLO). SLO is specified by the site operators and typically requires a high quantile of the request latency to be lower than a specified threshold. An example SLO might require that 99^{th} percentile of latency during each 1-minute interval to be less than 100 ms. As described in the Dynamo paper [DeCandia *et al.*, 2007], Amazon uses even 99.9^{th} percentile in their SLO. Because popular Web applications want to guarantee good performance for all their users, specifying the performance SLO using averages, medians, or variance is not sufficient. While you can compute bounds on percentiles from the mean and variance using Chebychev's inequality, those bounds can be quite conservative.

Elasticity would reduce costs for most Web applications because they have diurnal workloads. However, stateful systems would benefit even more during unexpected workload spikes with data hotspots. While in stateless systems preparing for a 20% spike in workload means simply adding 20% more servers, in stateful systems the 20% increase can hit data items all stored on a single server, thus creating a hotspot. As we demonstrated in the previous chapter, workload spikes with significant data hotspots are relatively frequent. Since the hotspot location is unknown before the spike, statically overprovisioning for this situation would be extremely inefficient and expensive. Because most Web sites do not overprovision for large spikes, elastic storage systems would help avoid decreased performance or outages.

Automatic scaling of stateful systems presents additional challenges compared to stateless systems. First, in stateful systems it is not enough to simply add or remove servers; the control policy also has to decide how to partition or replicate data across the new servers. Second, copying data between servers both takes time and, because of the additional activity on the servers, increases request latency. A dynamic resource allocation policy for stateful systems has to address both of these challenges.

We make two contributions in this chapter:

- We formulate three basic principles of designing resource allocation frameworks for stateful systems that maintain performance SLO expressed using high quantiles of the latency distribution.
- We design and implement the Director – a modular control framework for automatic resource allocation in stateful systems based on Model-Predictive Control – and demonstrate that the same control policy can handle both periodic diurnal workloads and unexpected workload spikes with data hotspots.

We evaluate Director on the SCADS [Armbrust *et al.*, 2009b] storage system running on Amazon EC2. We show that it can quickly respond to unexpected workload spikes with data hotspots and still maintain strict performance SLO. During workloads with regular diurnal variation, Director reduced the server costs by 16% to 41% and was close to the optimal server allocation.

In the rest of the chapter, we first formulate the three design principles in Section 3.2 and then describe in detail our modular control framework in Section 3.3. Section 3.4 describes the statistical performance models used in our control framework. Section 3.5 presents the experimental setup and results.

3.2 Approach

Many storage systems, such as HBase [Apache, 2010b], Cassandra [Apache, 2010a], or PNUTS [Cooper *et al.*, 2008], can be dynamically reconfigured on the fly by adding more servers. However, they were not designed to respond quickly to changes in workload and maintain strict performance SLO during such scale up or scale down. When running in the cloud, application developers could reduce their server costs by coalescing the data to fewer servers at night and releasing the unused servers. During workload spikes, the system could be automatically scaled up by requesting more servers and partitioning and replicating data according to the current workload. Our goal is to design a general control framework that would enable such automatic scale up and scale down while maintaining strict performance SLO.

3.2.1 Design Principles of Control Policy

Any control policy for automatic resource allocation of stateful systems needs to make two main decisions: *when* to act and *how* to act. In particular, deciding *when* to act

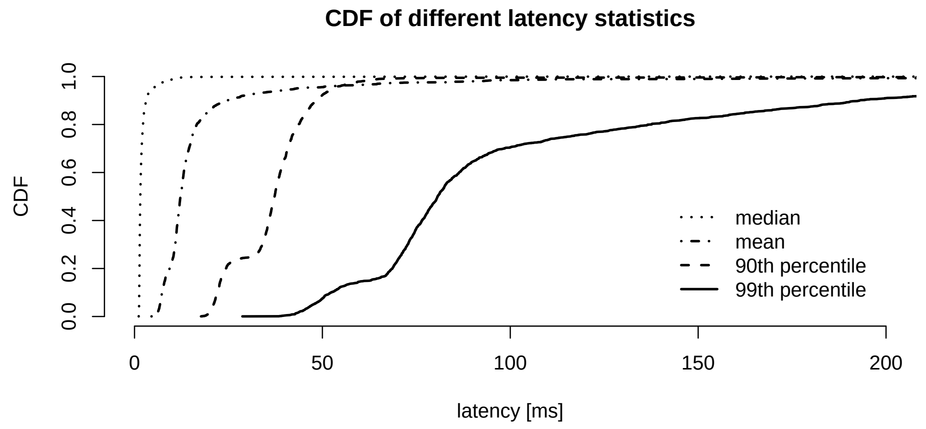


Figure 3.1: CDFs of different statistics of request latency based on a benchmark with flat workload against 32 SCADS storage servers. We obtained the data for the CDFs by computing the mean, median, 90^{th}, and 99^{th} percentile for each 20-second interval in the benchmark. Notice that the variance of the mean and median latency is much lower than the variance of the 99^{th} percentile.

corresponds to determining when the system is overloaded and needs to be scaled up or when it is underloaded and could be scaled down. Deciding *how* to act corresponds to determining how many servers should be added or removed and which data should be moved and where.

We formulate the following three principles for designing control frameworks to allocate resources in stateful systems automatically and maintain a performance SLO expressed using high quantiles of latency. Principles one and two concern when to act. They are principles of Control Theory applied to the constraints of resource allocation in stateful systems. The third principle concerns how to act:

1. Control policy cannot use the high quantile of latency as the only reference signal;
2. Control policy needs to include a feed-forward component and model the SLO violations as a function of the reference signal; and
3. Control policy needs to take into account the amount of data copied.

Principle 1: Control policy cannot use the high quantile of latency as the only reference signal

While closed-loop or feedback control is a popular control approach, there are two reasons why it is inadequate for resource allocation in stateful systems. First, as we demonstrate in Figure 3.1, high quantiles of latency have very high variance even for fixed workloads. If the policy used such a noisy signal, it would be constantly adding new servers only to remove them in the next iteration. It is a standard approach to smooth the noisy signal, however, smoothing leads to significant delays in control and late reactions by the policy and thus causes SLO violations.

Second, even if the latency signal were not noisy, it does not tell us *how much* overloaded or underloaded a server is. For example, if the latency threshold specified in the SLO is 100 ms and the current latency is 120 ms, we do not know if we need to decrease the workload by 5% or by 50% to bring the latency below 100 ms. Similarly, if two servers both have latency of 50 ms, could we merge their workloads and still get latency below 100 ms? We cannot tell based only on request latency. Therefore, the control policy needs to use a reference signal other than the high quantile of latency.

Principle 2: Control policy needs to include a feed-forward component and model the SLO violations as a function of the reference signal

We have given reasons to use a reference signal other than the high quantile of latency, however, the control loop still needs to maintain good performance and avoid SLO violations. Thus, the policy needs a model, or a transfer function, that predicts whether a particular value of the reference signal indicates an SLO violation. Such a model would also allow the policy to quantify how overloaded or underloaded each server is. Consequently, the policy could estimate how much data it has to move from an overloaded server or whether it can merge two underloaded servers.

Principle 3: Control policy needs to take into account the amount of data copied

In most large-scale stateful systems, it is infeasible to replicate all data on all storage servers. Instead, the data is partitioned across many servers and thus not all servers can handle every request. Therefore, when adding or removing servers in stateful systems, the policy also has to move data between servers.

Because copying data between servers is not free, the resource allocation policy needs to take into account the amount of data copied during scale up and scale down. Copying data is time intensive and has negative impact on performance of the storage system. We illustrate both in Sections 3.4.2 and 3.4.3.

Model-Predictive Control

We follow these principles and propose a control framework based on Model-Predictive Control (MPC) [Rossiter, 2003]. Our control policy uses workload as the reference signal and uses a performance model of the stateful system that predicts whether a server can handle a particular workload without violating the SLO. The advantage of using workload as the policy input is that we can directly control workload on each server by moving data between servers. We use Statistical Machine Learning (SML) to build an accurate performance model of the system based on data collected during a benchmark.

We assume that the underlying storage system is easy to reconfigure on-the-fly. In particular, the system needs to support `AddServer` and `RemoveServer` operations that add a new server to the system and remove an empty server from the system, respectively. Also,

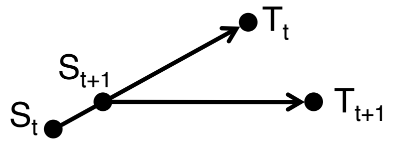


Figure 3.2: Illustration of the MPC approach. S_t and S_{t+1} correspond to the current states (i.e., input to the controller) at times t and $t+1$, respectively. T_t and T_{t+1} correspond to the target states (i.e., output of the controller) at times t and $t+1$, respectively. The controller executes one action to get from S_t to S_{t+1}, but then selects a new target state to adapt to changing conditions.

the system needs to store data in small chunks, or bins, and to support the ability to move, copy, and delete these bins. Specifically, the system needs to support three operations – `Move`, `Copy`, and `Delete` – that move a single bin between two servers, copy a bin from one server to another, and delete a bin from a server, respectively. The `Move` operations will be used to partition the data, the `Copy` and `Delete` operations will be used to create and remove additional bin replicas.

The control policy monitors workload to the individual data bins. As we demonstrated in the previous chapter, workload spikes on stateful system create data hotspots that could overload individual servers. By using the per-bin workload metrics, the policy could quickly identify the hotspot and move the corresponding data to a server with enough spare capacity. Using per-bin metrics also allows us to respond to a data hotspot with resources proportional to the magnitude of the spike, instead of resources proportional to the total data size. We discuss the size of the bins in Section 3.3.1.

If per-bin workload metrics were not available, the policy would have to use a *per-server* metric, such as CPU utilization or workload, to identify the hotspot. One option would be a split-in-half policy, where half the data of an overloaded server is moved to a new server. However, if using only per-server load metrics, the policy would not know how hot each of these halves is. Such policy might require several iterations to identify the hotspot and would thus take much longer to respond and would copy much more data than necessary.

In MPC, the controller uses a model of the system and its current state to compute the (near) optimal sequence of actions of the system that maintain the desired constraints. To simplify the computation of these actions, MPC only considers a short receding time horizon. The controller then executes only the first action in the sequence and then uses the new current state to compute a new sequence of actions. Thus, in each iteration, the controller reevaluates the system state, computes a new target state and starts moving towards it to adjust to changing conditions. We illustrate MPC in Figure 3.2.

Like MPC, our policy considers the current workload and data layout and computes a target data layout that does not violate the SLO and minimizes the number of allocated servers. The policy outputs a set of actions that change the layout from the current one to the target one, and starts executing them. However, if some actions do not finish within a short period of time, the policy cancels them and uses the most recent data layout and workload measurements to compute a new layout target.

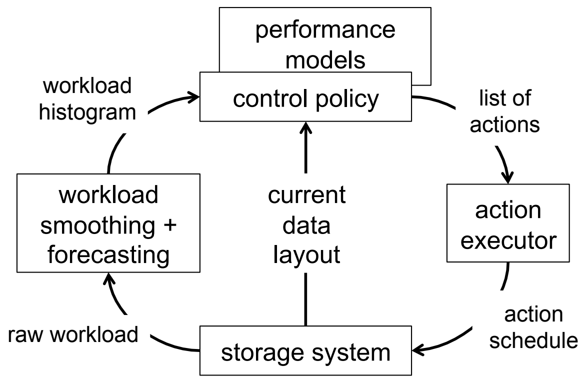


Figure 3.3: The Director control framework consists of four main modules: workload forecasting, control policy, performance model, and action scheduler.

3.3 Director Control Framework

Our control framework based on MPC consists of four main modules: workload forecasting, control policy, performance model, and action scheduler (see Figure 3.3). The *workload forecasting* module monitors the workload to the system, smoothes it, and uses the current and historic workload data to make predictions about future workload. *Control policy* uses the *performance model* to detect overloaded servers and moves data away to decrease workload on these servers. Finally, the *action scheduler* schedules the data copy actions to minimize their impact on performance. We describe the performance model in Section 3.4 and the remaining modules in the following subsections.

3.3.1 Workload Monitoring and Forecasting

The goal of workload monitoring and forecasting is to provide the policy with detailed information on workload for different parts of the full data range. The control policy uses this information together with the performance model to determine which servers are overloaded or underloaded, which bins of data to move and where to move them. This detailed workload information is necessary to reduce data copying, as stated in the third principle. The workload is represented by a *workload histogram* that contains workload request rates for individual request types for each data bin.

The total number of bins is a parameter of the control framework. Setting the value too low or too high has its drawbacks. With too few data bins, the control policy does not have much flexibility in how it moves data from overloaded servers and might have to copy more data than necessary. Having too many data bins increases the load on the monitoring system and running the control policy might take longer since it would have to consider more options. In practice, we found that having on average five to ten bins per server provides a good balance.

Because storage systems often handle very high workload rates – in our case, 7,000 requests per second per server – to further reduce the impact of workload monitoring, we

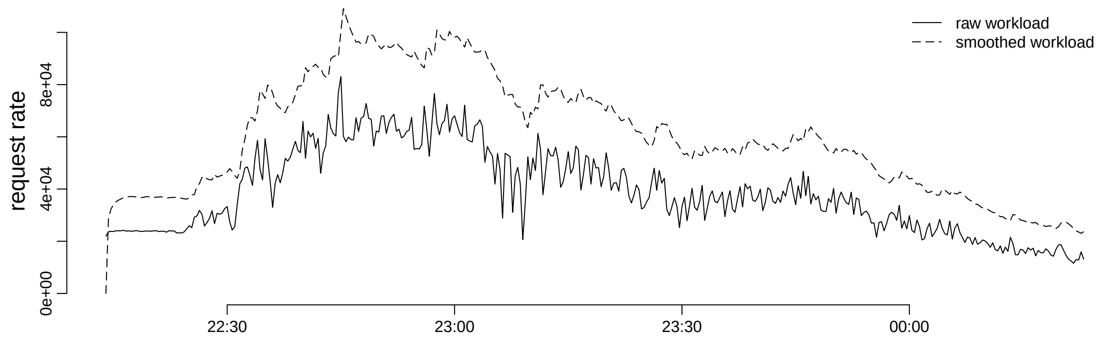


Figure 3.4: The actual raw workload volume during an experiment from Section 3.5 and workload smoothed with parameters $\alpha_{up} = 0.9$, $\alpha_{down} = 0.1$ and safety buffer of 0.3. For example, notice the spike in raw workload at 22:45. Because of high α_{up} value, the smoothed workload quickly increased, but decreased more slowly because α_{down} has a low value.

only monitor a small fraction, W_S, of all requests. For each sampled request, we monitor the key it accessed and its latency. We use Chukwa [Rabkin and Katz, 2010] to collect the monitoring data and compute the workload on each bin every T_{data} seconds in a central location. The output of this processing step is a *raw workload histogram* that represents the workload rates of different request types in each bin during the most recent time interval.

We feed the raw workload histogram into the workload forecasting module which processes the sequence of raw histograms and applies smoothing, historic forecasting, or workload prediction. In our implementation of the workload forecasting module, we concentrate mostly on smoothing the workload in each bin so the control policy does not react to tiny spikes in workload. To smooth the workload, we use hysteresis as is standard practice in control systems. Because we want to respond quickly to workload spikes and then slowly scale down, we apply hysteresis with two parameters: α_{up} and α_{down}. If the workload increases compared to the previous value of the smoothed workload, we smooth with parameter α_{up}, otherwise we smooth with α_{down}. We perform smoothing independently in each bin and also add a safety buffer by multiplying the workload by a constant greater than 1. You can see example of the smoothed workload in Figure 3.4.

We note that the variance of workload is much smaller than the variance of the 99th percentile of latency. For example, we obtained the CDF of the 99th percentile of latency on Figure 3.1 on page 29 from a benchmark with flat workload with no variance. We use hysteresis mainly to prevent the system from scaling down very quickly.

3.3.2 Control Policy

The goal of the control policy is to change the configuration of the storage system to minimize the number of servers in use while maintaining the performance SLO. The policy considers the smoothed workload histogram and the current data layout on the storage system and outputs a set of actions that change the system configuration. When handling

```
Run every $T_{policy}$ seconds:

  1. if some actions are still running
      (a) if less than $T_{block}$ seconds since we started the actions, stop
      (b) else cancel all unscheduled actions, wait until all running actions complete and
          continue
  2. estimate workload on each server and classify servers as overloaded or underloaded
  3. for each overloaded server S
      (a) if S only stores one range, add more replicas of that range
      (b) else if a single server cannot handle workload on the hottest bin B on S:
            i. move B to an empty server
           ii. add more replicas of B
      (c) else move bins B from server S (starting with the hottest bin) to the most-loaded
          underloaded server that can still accept bin B until S is not violating SLO
  4. for each underloaded server S that is not scheduled to receive any new data (from
     step 3), sorted by increasing workload
      (a) if S contains only a single bin replica, remove the bin if no longer necessary
      (b) else for each bin B on S
            i. move B to most-loaded underloaded server with workload higher than S that
               can still accept B
           ii. if cannot move B, leave it on S
  5. add or remove servers as necessary, based on previous actions
```

Figure 3.5: Control policy pseudocode.

an overloaded server, we first consider partitioning data on the server and only create replicas if the server has a single bin that cannot be split. This general policy minimizes the amount of copied data and avoids creating unnecessary replicas.

Given the information on the workload in each data bin, the number of servers would be minimized by solving a bin-packing problem – packing the data bins into servers – which is an NP-complete problem. While approximate algorithms exist, they typically do not consider the current locations of the bins and could thus completely reshuffle the data on the servers, which would be a costly operation. Instead, our policy is based on a greedy heuristic that moves data away from the overloaded servers and attempts to coalesce the underloaded servers.

The policy uses *steady-state* and *copy-duration* performance models to make its decisions (see the second principle in Section 3.2.1 on page 28). The steady-state model predicts whether a storage server can handle a particular workload without violating the performance SLO. The policy uses this model to compute which servers are overloaded and to find servers that have enough spare capacity to accept additional workload.

To simplify the reasoning about the state of the system, the policy does not run until all the data copy actions have completed. However, if actions took many minutes to finish, they could block the policy from running and prevent it from responding to sudden changes in workload. Therefore, the policy uses the copy-duration model to estimate the duration of the actions and splits long-running actions into shorter ones. If some of the actions do not complete within a short time period, T_{block}, the policy cancels them so it can respond to changes in workload.

Finally, to avoid waiting for new servers to boot up during a sudden workload spike, the policy maintains a pool of hot-standby servers of size H. These servers are already running the storage system software, but are not serving any data. Hot-standbys are particularly useful for handling data hotspots when replicas of a bin require a new empty server. If no hot-standbys were available, to respond quickly to a data hotspot would require moving all the cold bins from the overloaded server instead of moving just the hot bin to an empty server. Such operation would require moving a lot of data and would thus conflict with the principle that the control policy has to minimize the amount of copied data.

Details of Control Policy Implementation

The policy proceeds in five steps: 1) checks if there are still any actions running, 2) estimates the workload on each server and classifies each server as overloaded or underloaded, 3) fixes overloaded servers using data partitioning and replication, 4) merges underloaded servers by reducing replication and coalescing data bins on the underloaded servers, and 5) adds or removes servers based on previous steps. We describe these steps in more detail below and summarize them in Figure 3.5 on the preceding page.

First, the policy decides whether to proceed based on the status of the previously-scheduled actions. If no actions are running, the policy starts immediately. If some actions are running, but it has been less than a minute since the last time the policy was run, the policy waits. After one minute, the policy cancels any unscheduled actions, waits for any unfinished actions to complete, and then proceeds.

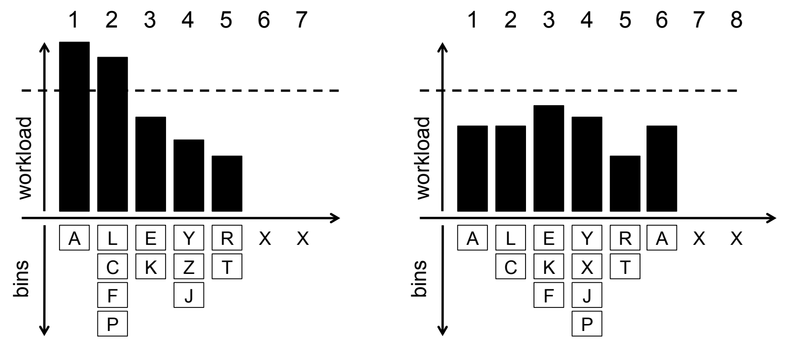


Figure 3.6: Example of the policy response to two overloaded servers. In both figures, the columns represent the storage servers, the top bar graphs represent the workload on each server and the dashed line represents the workload threshold obtained from the performance model (see Section 3.4). The boxes on the bottom represent bins stored on the servers and the two last servers marked with an "X" are the hot-standbys with no data and no workload. In the left graph, servers 1 and 2 are overloaded. Server 1 only has one bin, which cannot be split and thus has to be replicated to a new server – server 6 in the right graph. Server 2 has four bins. Two of them, F and P, are moved to servers 3 and 4. Because one of the hot-standbys was allocated to the replica of A, the policy also requests a new hot-standby server, 8.

Second, the policy uses the smoothed workload histogram and current data layout to classify servers as overloaded or underloaded. The workload histogram represents the workload on the individual bins, and the data layout maps the data bins to servers that store that bin. By combining the histogram and data layout, we can estimate the workload rate of get and put operations on each server. We use the steady-state performance model to classify all servers that cannot support their predicted workload as overloaded. The remaining servers are classified as underloaded.

Third, we handle overloaded servers by using data replication and partitioning. If an overloaded server S contains a single bin B with workload so high that B cannot be supported on a single server, the policy places B on a new, empty server. We use the steady-state model to compute how many replicas are necessary to handle workload to this bin and replicate the data accordingly. If S does not contain such a hot bin, we move some bins from this server to less loaded servers. In particular, we start with the most-loaded bin on the server and move it to the most-loaded underloaded server. We continue moving bins until server S can handle its own workload. Figure 3.6 illustrates this step.

Next, the policy attempts to reduce data replication and coalesce data on the underloaded servers. If a server S contains only a single bin replica, we remove the replica if it is no longer needed, according to the performance model. If S contains multiple bins, we try to move these bins to another underloaded server with load *higher than server S*. This heuristic aims to move as much data as possible from the least loaded servers so that these servers can be removed.

Finally, based on the previous steps, the policy adjusts the number of servers to maintain the desired number of hot-standbys. If the workload increased and the policy used up one or more of the hot-standbys, it requests additional servers. On the other hand, if the workload decreased and more servers became available, the policy releases some servers. We note that the policy does not immediately execute the data copy actions. Instead, it only creates descriptions of these actions and feeds them to the action scheduler.

3.3.3 Action Scheduler

On most storage systems, copying data between servers has negative impact on the interactive workload. Section 3.4 shows that in SCADS benchmarks, the copy operation affects the target server significantly, but the source server is largely unaffected. Therefore, executing all data copy actions concurrently might overwhelm the system and reduce performance. On the other hand, executing the actions sequentially minimizes the impact on performance, but is very slow.

Just as data replication can improve the steady-state performance of storage systems, replication is also beneficial during data copy. We use an action scheduling policy that executes actions as concurrently as possible, subject to the constraint that each data bin has at least one replica on a server that is not affected by data copy. When an action completes, we go through all the remaining, unscheduled actions and schedule those that do not violate any constraint. This ensures that each bin has at least one replica with good performance.

3.3.4 Setting Values of the Control Policy Parameters

Table 3.1 on the following page summarizes the parameters of the control framework. Here we describe how we set the values of the parameters, the trade-offs associated with each parameter, and how one could tune the parameter values to improve the control policy.

Workload sampling fraction controls the fraction of the requests that are used for estimating the workload on the individual servers. Using too few requests would result in inaccurate and noisy estimates of the workload and might thus cause oscillations in the policy actions. On the other hand, using too many requests could overload the monitoring system or have impact on the interactive requests. In practice, we found that 2% sampling provided good estimates of the workload without decreasing performance noticeably.

We point out that the monitoring framework we used, Chukwa [Rabkin and Katz, 2010], processed the individual requests in a central location, thus creating a bottleneck. In our experience, increasing the sampling fraction would overload the central server, which caused delays in the delivery of monitoring data. Aggregating the requests on the individual storage servers would enable us to monitor all the requests with little impact on performance.

Monitoring period controls the frequency of receiving workload updates and is mostly constrained by the properties of the monitoring framework. Because our control framework uses hysteresis to aggregate workload information over time, it could use arbitrarily short monitoring periods. In real deployments, web site operators typically monitor their datacenters at one-minute intervals. We set our monitoring framework to report workload every

Parameter	Description	Value used in experiments
	Workload smoothing	
W_S	workload sampling fraction	2%
	number of data bins per server	5 - 10
T_{data}	monitoring period	20 seconds
α_{up}	smoothing up	0.9
α_{down}	smoothing down	0.1
	safety buffer	0.1
	Policy	
T_{policy}	policy period	20 seconds
T_{block}	max policy blocking time	1 minute
	max action duration	1 minute
H	number of hot-standbys	2

Table 3.1: Control framework parameters.

20 seconds to allow the policy to quickly respond to changes in the workload.

Average number of data bins per server adjusts the granularity of workload monitoring as well as the granularity of data copy and move actions. Using too few data bins would not allow the policy to isolate small data hotspots. This would lead to unnecessary data copying in conflict with our third principle. Using too many bins would increase the amount of monitoring data. In our experiments, we found that using more than five or ten bins per server did not further reduce the amount of data copied.

Smoothing parameters and the **safety buffer** control the hysteresis of the workload rate on each data bin and overprovisioning of the system, respectively. Values of these parameters significantly affect the number of SLO violations and the server cost. For example, setting α_{down} too high provides little workload smoothing when the workload drops, which forces the policy to scale the system down quickly and can lead to SLO violations later. Setting the safety buffer too high leads to too much headroom on each server and thus wastes resources.

We used a control policy simulator, such as the one described in [Bodík *et al.*, 2009b], to tune the values of these parameters. The simulator takes as input a workload trace and the values of various policy parameters, and estimates the server cost and number of SLO violations for these parameters. The simulator uses the performance model to estimate the latency on each server during the simulation. Local optimization methods, such as gradient descent, could be used to find the optimal values of these parameters. Control policy simulation is a general framework that could be used to understand the effects of arbitrary policy parameters, such as the policy blocking time and the number of hot-standby servers described below.

Policy period controls how often the policy reacts to changes in workload. We use policy period equal to the monitoring period to allow the policy to respond to each new workload update.

Policy blocking time and **maximum action duration** control the amount of time the policy waits for actions to complete before making a new decision. Both of these should be set to at least the duration of the *policy period*. However, we set their values to one minute so that the actions are not interrupted by the policy too often, while letting the policy to react to new workload at least once a minute.

The safety buffer parameter described above controls the overprovisioning of all storage servers and thus helps to absorb small workload spikes on the data bins. **Number of hot-standby servers** controls overprovisioning at the level of the whole storage cluster by setting aside empty servers that can be added instantly to the cluster. For our policy, hot-standbys are particularly useful when creating additional replicas for a data hotspot. Because new replicas require empty servers, using hot-standbys avoids waiting for new servers to boot up.

The actual number of hot-standbys could be computed based on the workload spike that the website should be able to handle. For example, if the throughput of a single server is 7,000 requests per second, using two standbys allows the policy to quickly respond to a hotspot with 14,000 requests per second on one data bin.

3.3.5 Summary of Control Policy

Our approach to dynamic resource allocation in stateful systems is based on MPC. The proposed model-based control framework consists of four basic modules: workload smoothing and forecasting, control policy, performance model, and action scheduler. While our particular implementation of these modules is not necessarily the most suitable one for all storage systems, each module could be replaced without affecting the other modules.

Compared to closed-loop, model-free approaches, model-based control offers two main advantages. First, our policy does not exhibit oscillatory behavior because it does not respond to request latency, which could be affected by its previous actions. Second, we use the system performance model to measure the load on each server relative to the specified performance SLO. This, in turn, enables us to estimate how much data we have to move from an overloaded server or how much capacity is left on each server.

When applying this framework to a different storage system, the individual components could be tweaked based on the performance of that system and the type of workload it has to handle. The workload forecasting module could detect trends in the workload and make predictions about future workload volume. Or, if the target workload has a strong periodic component, historic forecast could be incorporated to workload forecasting which would enable the policy to prepare for the workload spikes ahead of time. To take advantage of heterogeneous computing resources offered by Cloud Computing providers, one could create different performance models for each server type and augment the policy to take advantage of all the available server types. Finally, depending on the impact of data copy on performance, available network bandwidth, and the amount of data on the servers, the action scheduler could be replaced with one that copies the data at different rates and with different level of concurrency.

Our model considers workload per server to be the main factor affecting system performance. However, there are other bottlenecks or constraints in storage systems, such as the amount of data and the network throughput, and the policy could consider these as additional reference signals. As stated in the second principle, one would have to build a model that predicts SLO violations based on these features. For example, how much network activity can the servers and the switches handle without affecting the high quantiles of request latency?

3.4 Performance Models

MPC reacts directly to changes in workload, instead of latency, but still needs to maintain SLO expressed using the high quantiles of request latency. Therefore, the policy needs a model that accurately predicts whether a server can handle a particular workload without violating the performance SLO (see the second principle in Section 3.2.1 on page 28). Using the predicted workload and the model, the policy determines which servers are overloaded and moves data from them to less loaded servers. The policy also uses a model of duration of the data copy operations to create short copy actions.

An MPC control policy using an inaccurate performance model could allocate either too many or too few servers for the storage system. If the model is too optimistic and overestimates the workload the system can handle, the control policy would underprovision the system which would lead to increased request latency or even failed requests. If the model is too pessimistic and underestimates the workload the system can handle, the policy would overprovision and thus waste computing resources. While using the safety buffer and hot-standbys also overprovisions the system, we control exactly how much it is overprovisioned. An incorrect model could potentially over- or underprovision by an order of magnitude. Because there is no explicit feedback loop in our MPC control framework, an accurate performance model is a necessity.

One of the standard approaches to performance modeling is using analytical models based on networks of queues [Urgaonkar *et al.*, 2005a]. These models, however, require detailed understanding of the system and often make strong assumptions about the request arrival and service time distributions. Consequently, analytical models are difficult to construct and their predictions might not match the performance of the system in production environments.

Instead, we use statistical and machine learning methods to create accurate models of system performance. We benchmark both the steady-state performance of the system as well as duration of copy operations and their impact on latency and we use the measured data to fit statistical models.

It is well known that performance of the system in an offline benchmark often does not match the performance in a production environment [Allspaw, 2008]. This disparity is mostly caused by difficulty to reproduce the production workload in a test deployment of the system. Building SML models from such offline benchmarks might lead to incorrect predictions and thus to incorrect decisions of the control policy. While in this section we use offline benchmarks to collect training data, an alternative approach would be to use *performance exploration*, which probes the system in a production environment and thus collects accurate performance measurements. We demonstrated such an approach in [Bodík *et al.*, 2009a].

3.4.1 Steady-state Performance Model

The goal of the steady-state performance model is to predict whether a server can handle a particular workload without violating the performance SLO. The policy uses this model to detect overloaded servers and decide where to move data from overloaded or underloaded servers. The performance SLO that we consider in this chapter is of the following form: k^{th} percentile of request latency over L-minute intervals should be less than X milliseconds. The procedure for building the performance model is parameterized by k and X. For example, decreasing the value of X would result in a model that allows less workload on each server. In this section we use $k = 99$ and $X = 100$, which are typical values used in SLOs.

The model uses the request rates of both get and put operations as input. In SCADS, all the data is stored in memory, which means we do not have to model accesses to disk or the disk cache. While the performance of SCADS could be affected by network utilization or activity of other virtual machines running on the same physical servers, we do not consider them in the model. In Cloud Computing environments, we do not have control over the network or activity of other users.

We collected the data for this model in a benchmark on four SCADS storage servers. We increased the workload linearly from 2,000 to 10,000 requests per second per server and used four different workload mixes of get and put operations: 50% gets/50% puts, 80%/20%, 90%/10%, and 95%/5%. These workload mixes are comparable to workloads used in Yahoo's storage benchmark [Cooper *et al.*, 2010]. For each workload mix, we collected a dataset $R = \{(w_i, s_i)\}$, where w_i represents the workload rate on one server during a 20-second time interval and s_i represents the fraction of requests slower than 100 ms during the same interval.

To visualize the dependency between workload on the server and fraction of slow requests, we group w_i's into buckets of width of approximately 200 requests per second and compute the average and standard deviation of s_i for each bucket. The top two graphs on Figure 3.7 show the performance of SCADS for the 80% and 95% workload mixes[1]. Notice that performance degrades significantly around 6,000 and 7,000 requests per second. However, the variance is high even for much lower workloads and is often above the threshold of 1%.

In stateful systems, data replication is often used to achieve better performance. For example, with two replicas of each data range, a user request would be forwarded to both replicas and the faster response would be sent back to the user. Using additional replicas thus reduces the probability that the user request will be slow [Dean, 2010].

We use a sampling procedure that simulates the execution of user requests with 2 or 3 replicas to create new datasets R_2 and R_3 from R. For N replicas, we repeatedly sample N points from a single workload bucket in dataset R; $p_i = (w_i, s_i)$ for $i \in \{1, \ldots, N\}$. The user request would be slow only if all the requests to the N replicas are slow. Assuming that all the servers behave independently, the probability of the user request being slow is

[1] "Mix of N%" means N% of gets and (100-N)% of puts.

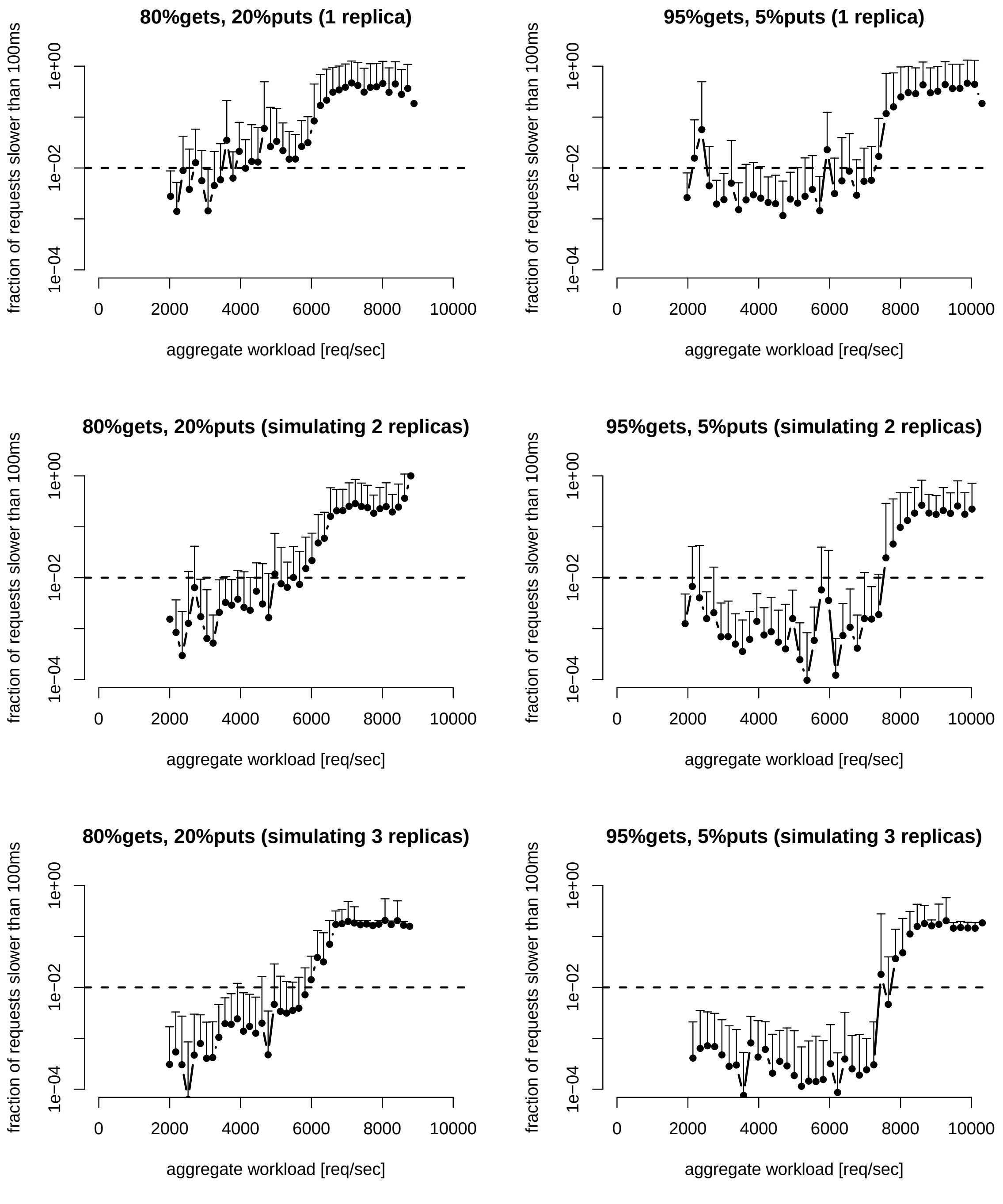


Figure 3.7: The x-axis represents the workload in requests per second, the y-axis represents the fraction of requests slower than 100 ms on a log scale, and the dashed horizontal line represents the 1% threshold performance SLO. For each workload bin, the average fraction of slow requests is represented by a dot and the error bar represents two standard deviations. The graphs on the left and right show data for 80% and 95% workload mixes, respectively. The three rows show different numbers of replicas.

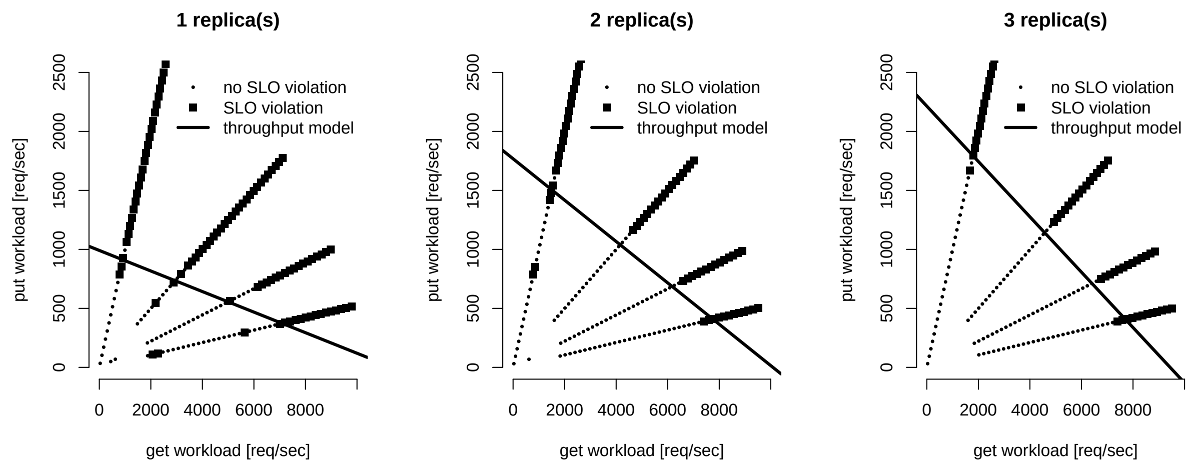


Figure 3.8: The training data and throughput model for one, two, and three replicas. On the each graph, the x- and y-axes represent the request rates of get and put operations, and the small dots and large squares represent workloads that the server can and cannot handle, respectively. The four "rays" in each figure represent four different workload mixes: from top-left to bottom-right, 50%, 80%, 90%, and 95%. The diagonal lines represent the boundary between acceptable and unacceptable performance, according to the fitted model.

$\prod_i s_i$. We thus add point $(1/N \sum_i w_i, \prod_i s_i)$ to the R_N dataset. Performance of SCADS estimated using this sampling technique is presented in rows two and three in Figure 3.7. Notice that the fraction of slow requests drops significantly, well below the threshold of 1%.

We use datasets R, R_2, and R_3 as training data for logistic regression to create a linear classification model of SCADS performance. The model uses two features: the workload rate of get and put requests. We obtain the binary class variable by discretizing the fraction of slow requests in each workload bucket into *high* or *low* based on the threshold specified in the performance SLO (dashed, horizontal line in Figure 3.7). The resulting binary datasets and the fitted linear logistic regression models are presented in Figure 3.8; notice that with more replicas the servers can support higher workload and still maintain the SLO.

3.4.2 Effect of Data Copy on Performance

While our control policy does not directly consider the effects of data copy on system performance during real-time decisions, we considered these effects when designing the policy and the action execution modules. Here we present the results of a benchmark where we measured these effects.

In the benchmark, we varied the workload mix, workload rate, and the copy rate of the SCADS copy operation. We used two servers and each trial consisted of copying 10MB of data between them. While the copy operation was in progress, we also ran a workload against both servers with rates of 4,000, 6,000, and 8,000 requests per second per server. We used workload mixes of 50%, 80%, 90%, and 95% and copy rates of 100 kB, 250 kB, 500 kB,

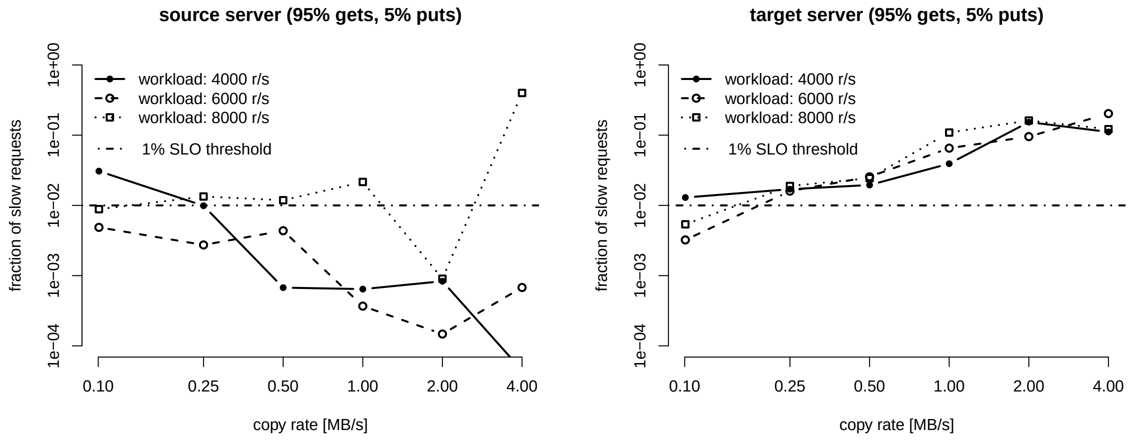


Figure 3.9: Performance of gets during a copy operation on the source and target servers (left and right, respectively). The x-axes represent the copy rate (in log scale), y-axes represent the fraction of requests slower than 100 ms (in log scale), and the horizontal line represents the 1% SLO threshold. The three lines show the average fraction of slow requests across the five iterations for different workloads on the servers.

1 MB, 2 MB, and 4 MB per second. For each combination of workload rate, workload mix, and copy rate, we ran five iterations of each trial and recorded the duration of the copy operation and performance on both the source and target servers.

Figure 3.9 presents the performance data for 95% workload mix. Notice that the effect on latency on the source server is small because it only performs data reads, which are cheaper than writes. On the target server, there is a clear trend – faster copy rates affect the latency more.

3.4.3 Model of Copy Duration

To avoid creating long copy actions which would take many minutes to complete and to react rapidly to changes in workload, our control policy creates short actions. While the copy operation in SCADS has a *target copy rate* parameter, the actual copy rate is often lower because of activity on both servers. To allow the policy to estimate the duration of a copy action, we build a model that predicts the *copy rate factor* – the ratio of actual and target copy rates. A copy rate factor of 0.8 means that the copy operation achieved only 80% of the target copy rate. Given the target copy rate and estimated copy rate factor, we compute the actual copy rate and thus the duration of the copy operation.

We use data from a benchmark described in the previous section to fit the model of the copy rate factor. Figure 3.10 illustrates that the target copy rate, workload rate and workload mix all have impact on the copy rate factor. We model the copy rate factor using linear regression with features linear and quadratic in the target copy rate and get and put request rates. As in the steady-state performance model, we ignore the network utilization and activity of other VMs.

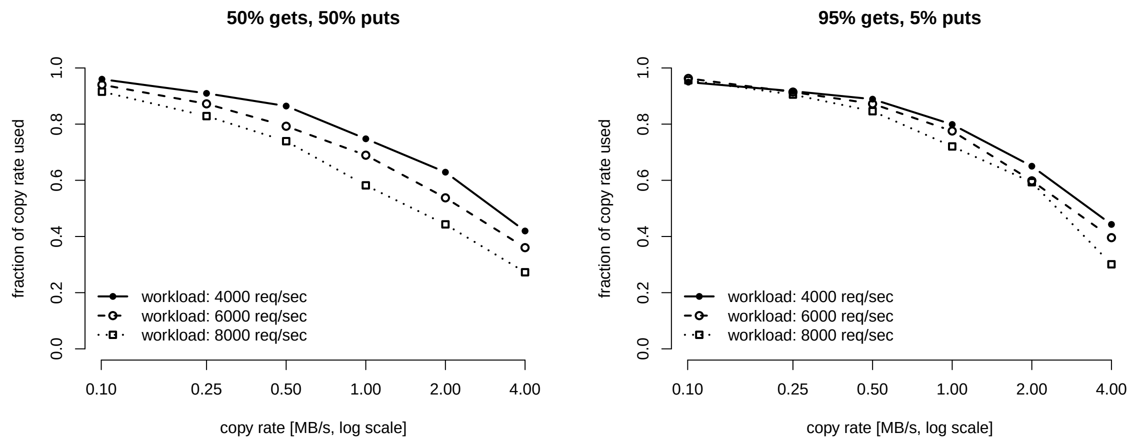


Figure 3.10: Actual copy rates obtained by benchmarking the SCADS storage system. X-axes represent the target copy rate.

3.4.4 Summary of Performance Models

In summary, we created two models: steady-state and copy-duration. The steady-state model uses the get and put workload rates on a single server as features, and predicts whether the server can handle that workload without SLO violations. The copy-duration model takes workload, target copy rate, and data size as input, and predicts the duration of the copy operation. Both of these models are inputs to the policy.

3.5 Experimental Results

In this section we demonstrate that our proposed control framework can respond to unexpected workload spikes as well as to diurnal workload changes by dynamically adjusting the configuration of a stateful system while maintaining a strict performance SLO. We chose these two workloads because they represent the two main scenarios where our proposed control framework could be applied.

Unexpected workload spikes with data hotspots are difficult to handle in stateful systems because the location of the hotspot is unknown before the spike. Therefore, statically overprovisioning for such spikes would be very expensive. Instead, we demonstrate that our control framework can quickly respond to such spikes and maintain good performance even during the spike ramp up.

As we demonstrated in the previous chapter (Section 2.3 on page 8), most of the workload during a spike in a stateful system is concentrated in a few hotspots. It is relatively easy for our control policy to react to such spikes because only a very small fraction of the data has to be replicated. We can thus handle a spike with data hotspots with resources proportional to the magnitude of the spike, not proportional to the size of the full dataset.

Our spike workload is based on the statistics of the “9/11 spike” experienced by CNN.com as a result of terrorist attacks on September 11, 2001 [LeFebvre, 2001], where the workload increased by an order of magnitude in 15 minutes, which corresponds to about 100% increase in 5 minutes. We simulate such workload by using a flat, one-hour long period of the Ebates.com workload [Bodík *et al.*, 2005] to which we add a workload spike with a single hotspot. During a five minute period, the aggregate workload volume increases linearly by a factor of two, but all the additional workload is directed at a single object in the system. Using the terminology of the previous chapter, this spike has a single hotspot, time of peak of 5 minutes, and relative increase in aggregate workload of 2.0. To handle this spike, our control policy dynamically creates eight additional replicas of the hottest data bin.

We further demonstrate that the same control policy can reduce the server cost during diurnal workloads. In our experiment, the request rate profile of the diurnal workload is based on a 24-hour period from the Ebates.com workload [Bodík *et al.*, 2005]. The popularity of the individual objects is sampled from the Zipf distribution and remains constant during the experiment. To reduce the duration of the experiment, we replay the 24-hour period in two hours.

To compensate for the 12x faster workload, we reduce the server boot-up time, server charge interval, and total data size also by a factor of 12 while keeping the copy rate the same. Because we cannot shorten the monitoring interval and the policy will take the same time to execute, we still run the policy with a period of 20 seconds, which would correspond to a 4-minute period in real workload.

We replay both workloads against SCADS running on 20 m1.small[2] instances on Amazon EC2 [Amazon.com, 2010] and use 60 additional servers to generate load. We split the full data range into 200 bins and use replication factor of two for improved performance; each get request is sent to both replicas and we count the faster response as its latency. We do not consider the latency of put requests in this work. Even though the benchmarks in

[2]An m1.small instance has 1.7 GB of memory, 1 EC2 Compute Unit, and 160 GB of storage.

Parameter	**Value for spike**	**Value for diurnal workload**
server boot-up time	3 minutes	15 seconds
server charge interval	60 minutes	5 minutes
server capacity	800 MB	66.7 MB
size of 1 key	256 B	256 B
total number of keys	4.8 million	400 thousand
number of replicas	2	2
total data size	2.2 GB	196 MB
copy rate	4 MB/s	4 MB/s
policy execution period	20 seconds	20 seconds

Table 3.2: Various parameters for the spike and diurnal workload experiments. We replay the diurnal workload with a speed-up factor of 12 and thus also reduce the server boot-up and charge intervals and the data size by a factor of 12.

Section 3.4.2 on page 43 showed that copying faster than 100kB/sec increases the probability of SLO violations, in our experiments we use a copy rate of 4MB/sec. The availability of two replicas of each bin, combined with our action execution policy, reduce the impact of data copying. Our performance SLO is based on latency threshold of 100 ms and we report SLO violations for different quantiles and time intervals. Table 3.2 summarizes the experiment parameters.

3.5.1 Workload Spikes with Data Hotspots

Figure 3.11 on the following page summarizes the workload, 99th percentile of latency, and number of servers during the spike workload. The policy first takes advantage of the two hot-standbys and moves the two existing replicas of the hot bin to these servers. As the workload increases even more, it requests three extra servers and creates three more replicas. Eventually, the policy creates total of eight additional replicas of the hot bin.

Because our performance SLO is parameterized by the latency threshold, latency quantile, and duration of the SLO interval, we present the performance of the system for different combinations of these parameters. We keep the latency threshold fixed at 100 ms and vary the quantile and the interval (see Figure 3.12 on the next page). We make two observations. First, as expected, higher latency quantiles result in more violations. Second, it is more difficult to maintain an SLO with shorter time intervals.

If we wanted to overprovision SCADS to handle such a spike with a data hotspot on an arbitrary data bin, we would have to create eight additional replicas of all bins, which would be too expensive. Because our policy monitors workload on each bin, it can respond to spikes with data hotspots using resources proportional to the magnitude of the spike, instead of proportional to the size of the full dataset. As demonstrated in the previous chapter, workload spikes have very few hotspots and the policy thus needs to create replicas of only a few bins.

3.5.2 Diurnal Workload

We performed experiments with the diurnal workload using two different safety buffer parameters (see Section 3.3.1 on workload smoothing). With a safety buffer of 0.3, the smoothed workload is multiplied by a factor of 1.3, which offers more headroom and the system can thus better absorb small spikes in the workload. A safety buffer of 0.1 provides less headroom and thus higher savings at the cost of more SLO violations.

We compare the results of our experiments with the optimal resource allocation and two fixed allocations. In the optimal allocation, we assume that we know the exact workload in each of the 200 data bins during the whole experiment and compute the minimum number of servers we would need to support this workload for each 5-minute interval. The optimal policy assumes that moving data is free and thus provides the lower bound on the number of compute resources required to handle this workload without SLO violations.

The fixed-100% and fixed-70% allocations use a constant number of servers during the whole experiment. Fixed-100% assumes that we know the exact peak of the workload and computes the number of servers based on the peak workload and the maximum throughput of each server (7,000 requests per second, see Section 3.4.1). The number of servers of the

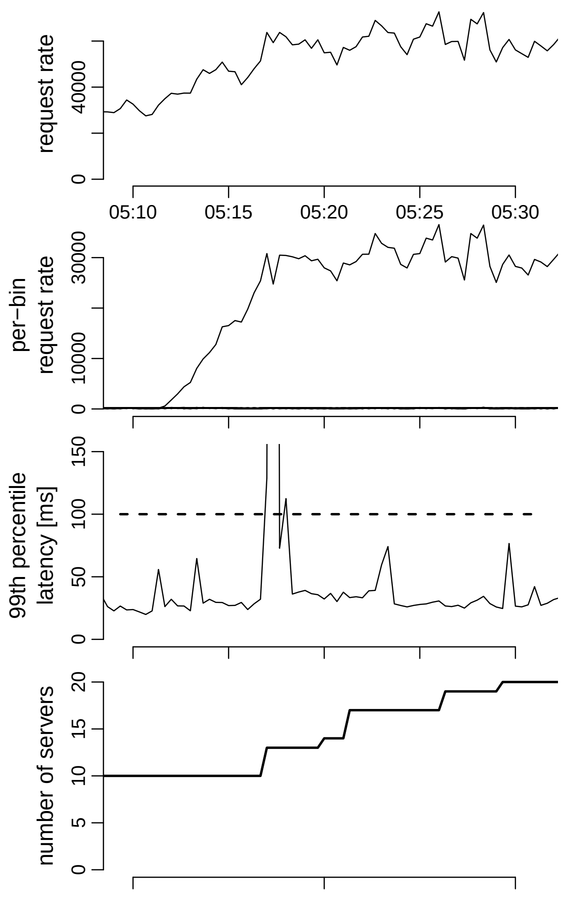
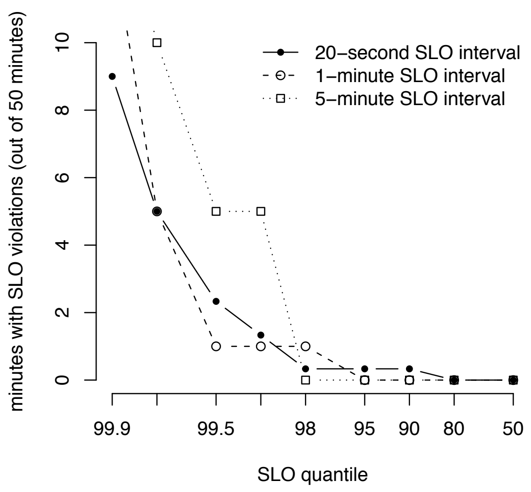


Figure 3.11: Summary of workload and performance during the experiment with the spike workload. First row: aggregate request rate during the spike. The rate doubled between 5:12 and 5:17. Second row: request rate for all 200 data bins; the rate for the hot bin increased to approximately 30,000 requests per second. Third row: 99th percentile of latency during the experiment along with the 100 ms threshold (dashed line). Bottom row: number of servers over time. The control policy keeps up with the spike during the first few minutes, then latency increases above the threshold, but the system quickly recovers.



Interval	Max quantile
20 seconds	80
1 minute	95
5 minutes	98

Figure 3.12: The top figure shows performance of SCADS during the spike workload for different SLOs. We keep the SLO threshold constant at 100 ms and vary the SLO interval length (20 seconds and 1 and 5 minutes) and the SLO quantile (from 50 up to 99.9). The y-axis represents the number of minutes with SLO violations (not the number of SLO violations). For example, both one 5-minute violation and 15 20-second violations add to 5 minutes of SLO violations. We present the maximum quantile of a compliant SLO configuration for each interval in the bottom table. Notice that we can support higher latency quantiles for longer time intervals.

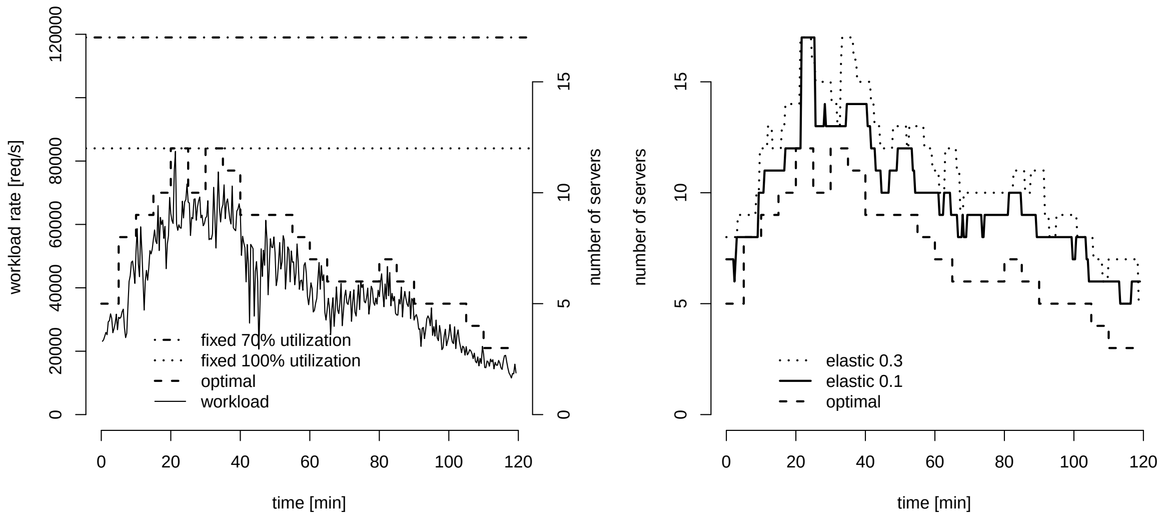


Figure 3.13: Left, workload (solid line) and number of servers assuming the optimal server allocation and two fixed allocations during the diurnal workload experiment. Right, optimal server allocation and two elastic Director allocations with safety buffers of 0.3 and 0.1.

fixed-100% allocation is identical to the maximum number of servers used by the optimal allocation. Fixed-70% also uses the exact peak of the workload, but runs the servers at only 70% of their capacity (i.e., $7,000 * 0.7 = 4,900$ requests per second). Fixed-100% is the optimal fixed allocation, but in practice, datacenter operators often add more headroom to absorb unexpected spikes.

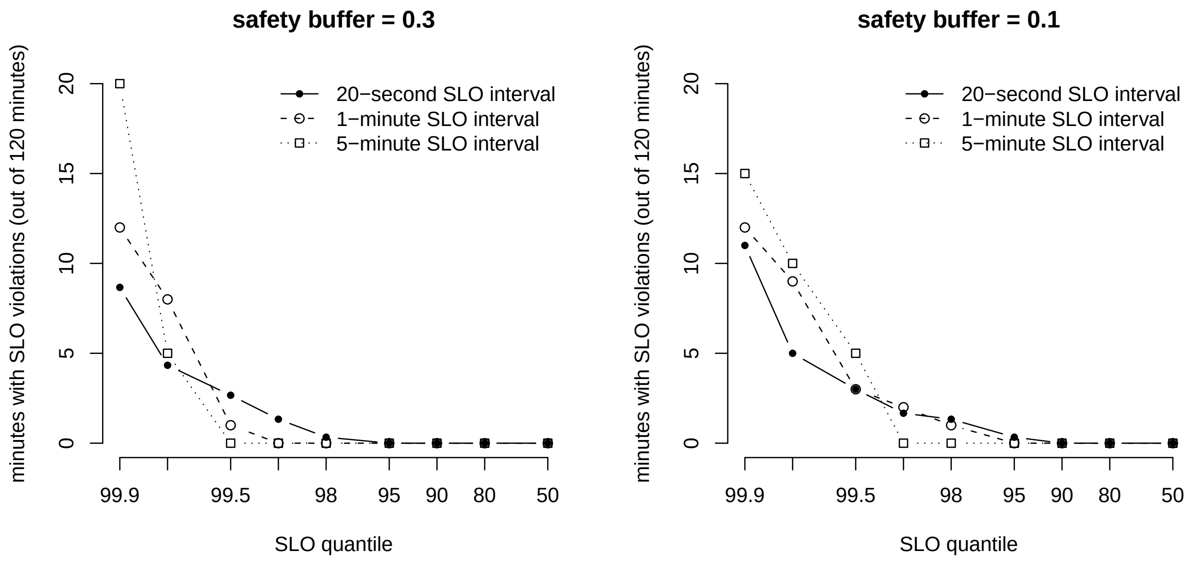
Figure 3.13 shows the actual workload profile and the number of servers used by the different allocation policies: optimal, fixed-100%, fixed-70%, and our elastic policy with safety buffers of 0.3 and 0.1. Figure 3.14 summarizes performance and SLO violations. The control policy with a safety buffer of 0.1 achieves savings of 16% and 41% compared to the fixed-100% and fixed-70% allocations, respectively.

The optimal resource allocation uses 175 server units, while our policy with a safety buffer of 0.1 uses 241 server units. However, recall that our control policy maintains two empty hot-standby servers to quickly respond to data hotspots that require replication. The actual number of server units used by our policy for serving data is thus 191[3], which is within 10% of the optimal allocation.

3.5.3 Summary of Results

In this section, we demonstrated that our control framework can respond to both workload spikes with data hotspots and workloads with regular diurnal variation. In the first experiment, our policy quickly responded to a spike comparable to one of the worst recent

[3]The experiment has a total of 25 5-minute server-charging intervals which yields 50 server units used by the hot-standbys.



Interval	Max quantile for SLO compliance	
	safety buffer = 0.3	safety buffer = 0.1
20 seconds	95	90
1 minute	99	95
5 minutes	99.5	99

Figure 3.14: Summary of SLO violations and maximum latency quantile supported with no SLO violations during the diurnal workload with two different safety buffer parameters; 0.3 (top left) and 0.1 (top right). See Figure 3.12 on page 48 for explanation.

spikes. It added eight additional replicas of the hot data bin and thus restored the performance of the system within two minutes. In the second experiment, the policy scaled the system up and down in response to a regular diurnal workload and would thus reduce the cost by 16% to 41%. We note that we used the same mechanism and the same policy to handle both workload scenarios.

3.6 Related Work

Much has been published on dynamic resource allocation for stateless systems such as Web servers or application servers [Chase *et al.*, 2001; Urgaonkar *et al.*, 2005b; Liu *et al.*, 2006; Kusic *et al.*, 2008; Hellerstein *et al.*, 2008; Tesauro *et al.*, 2006]. However, most of that work does not directly apply to stateful systems since the control polices for stateless systems do not have to consider data partitioning and replication. Also, copying data between servers takes time and negatively impacts performance of the system.

Recently, there has been work on power-proportional storage systems – such as Sierra [Thereska *et al.*, 2009] and Rabbit [Amur *et al.*, 2010] – which consume power that is proportional to the workload. Both papers target large file systems, such as HDFS or GFS, which store terabytes of data per server and thus cannot migrate all data from a single server on-the-fly. The approach that both papers take is to first provision the system for the peak load with multiple replicas of all data and then turn off servers when the workload decreases. These system thus cannot respond to workload spikes taller than the provisioned capacity or to data hotspots that affect individual servers. If the workload significantly exceeds the provisioned capacity, the data layout of these systems has to be changed offline. While both papers evaluate the performance of the system under the power-proportional control policy (Sierra uses the 99th percentile of latency), their goal is not to quickly respond to unexpected and large workload spikes.

In Everest [Narayanan *et al.*, 2008], the authors propose a *write off-loading* technique that allows them to absorb short burst of writes to a large-scale storage system. Everest is targeting large-scale storage servers with terabytes of data on each machine, similar to Sierra and Rabbit, and thus cannot handle a sustained workload spike or data hotspot because the data layout cannot change on-the-fly. Instead, Everest detects an overloaded server by monitoring request latency and redirects the writes to less loaded servers. When the workload decreases, the data written to the other servers is copied slowly back to their actual target server. While the authors measure the improvement in 99th percentile of latency during the 30 minute experiments, their goal is not to maintain strict SLO on short time intervals.

[Chen *et al.*, 2006] and [Soundararajan *et al.*, 2006] propose a database replication policy for automatic scale up and scale down. [Soundararajan *et al.*, 2006] uses a reactive, feedback controller which monitors the request latency and adds additional full replicas of the database. [Chen *et al.*, 2006] adds a proactive controller which uses a performance model of the system to add replicas before the latency increases.

These papers differ from our work in two main aspects. First, the performance SLO is based on mean request latency instead of the stricter high quantile of latency. Using mean latency hides latency spikes that would otherwise be visible. Second, both papers assume that the full dataset fits on a single server and only consider adding a full replica when scaling up (instead of partitioning). This implies that the system has to copy much more data than necessary. In our approach, we split the full data range into bins, monitor workload on each bin and move only the overloaded bins. We can thus handle a spike with data hotspots with resources proportional to the spike, not proportional to the total size of the data.

Distributed Hash Tables (DHTs) are decentralized distributed systems that provide primitives for insertion, deletion, and lookup of key-value pairs [Balakrishnan *et al.*, 2003]. Most DHTs were designed to withstand churn in the server population without affecting the availability and durability of the data. DHTs thus contain mechanisms to automatically partition data over new servers and to create additional replicas if servers disappear. However, quickly adapting to changes in user workload and maintaining a strict performance SLO during such adaptation were never design goals of the DHTs. Amazon's Dynamo [DeCandia *et al.*, 2007] is an example of a DHT that provides an SLO on the 99.9th percentile of request latency, but the authors mention that during a busy holiday season it took almost a day to copy data to a new server.

3.7 Summary

In this chapter, we considered the problem of dynamic resource allocation in stateful systems. Stateful systems that could be dynamically scaled up and down would be beneficial because they could automatically add resources during workload spikes to maintain the performance SLO and release resources at night to reduce costs. While much has been published on dynamic resource allocation for stateless systems, such as Web or application servers, these approaches do not apply directly to stateful systems.

We propose three principles of designing control frameworks for this problem. The first principle states that the control policy should not use high quantile of latency as the only reference signal. This principle is based on the observation that the 99$^{\text{th}}$ percentile of latency is very noisy and using it as the only reference signal would lead to oscillations. The policy might add more servers, only to remove them in the next iteration, or unnecessarily copy too much data back and forth. While smoothing the latency would reduce the noise, it would also result in delays in the controller.

As described in the first principle, the control policy should not use latency as the only reference signal, but it still has to avoid SLO violations specified using request latency. Therefore, the second principle states that the control policy should model SLO violations as a function of the reference signal.

Following these two principles, we designed Director – a control framework based on Model-Predictive Control. We use workload as the reference signal and a performance model of the system to determine whether each server can handle its workload without SLO violations. The performance model also helps us estimate how much data has to be moved from an overloaded server and how much spare capacity the underloaded servers have. We use SML techniques to fit an accurate performance model of the storage system based on data obtained using benchmarking.

Finally, the third principle states that the control policy has to consider data movement, because in typical storage systems, copying data between servers affects performance of the interactive requests. In our control framework, we monitor workload on each data bin in the system and move the hottest bins from the overloaded servers. Fine-grained workload monitoring allows us to respond to workload spikes with data hotspots with resources proportional to the magnitude of the spike, instead of proportional to the total size of the data.

In experiments using the SCADS key-value store, we demonstrate that the Director control framework can maintain a strict performance SLO while responding to unexpected workload spikes with data hotspots and adjusting system resources during diurnal workload patterns. In an experiment with a data hotspot, the control policy quickly requests additional servers and creates replicas of the hotspot. While the latency briefly exceeds the specified threshold, the system quickly recovers. In an experiment with a diurnal workload, the same policy reduces the cost by 16% to 41% and performs close to the optimal allocation policy.

Dynamic resource allocation policy can quickly add resources in case of a spike, but steep spikes can still overload an application and cause performance problems. Besides workload spikes, performance degradation and downtime can be caused by software bugs, hardware failures, or configuration errors. Web site operators have to quickly identify the problem and resolve it. In the next chapter, we propose a methodology for automatic resolution of performance problems that helps operators reduce time to recovery.

Chapter 4

Fingerprinting the Datacenter: Automated Classification of Performance Crises

4.1 Introduction

A datacenter *performance crisis* occurs when availability or responsiveness goals are compromised by inevitable hardware and software problems [Patterson *et al.*, 2002]. These crises can often be automatically and quickly *detected* through increase in request latency or anomalies in other standard performance metrics. However, it is usually very difficult and time consuming for the datacenter operators to *diagnose and resolve* these crises due to complexity of the underlying systems. Because the users expect uninterrupted availability of modern web sites, quick resolution of performance crises is critical to business and web site reputation.

Web application operators are on-call 24 hours a day, carry pagers, and must respond within 15 minutes to performance crises [Bodík *et al.*, 2006]. Their highest priority is to restore the web application to normal operating conditions. The operators typically diagnose the crisis by inspecting various system metrics, application logs, and alarms, and then perform a set of resolution steps which range from restarting processes and rebooting servers to redirecting traffic to a different datacenter.

However, as noted in [Bodík *et al.*, 2006], crisis resolution does not usually fix the root cause of the problem. Due to complexity of the system, repairing the bug and pushing the new code to production can often take several days or even weeks. In the meantime, the same crisis can recur many times. The same can happen if the fix is based on a misunderstanding of the root cause [Glerum *et al.*, 2009] or because of emergent misbehaviors due to large scale and high system utilization[1].

Crisis resolution can often be accelerated by automatically identifying the current crisis and proposing resolution steps that resolved the same crisis in the past [Bodík *et al.*, 2006; Cohen *et al.*, 2005]. A correct identification also avoids escalation of the crisis to higher

[1]Jeff Dean, Google Fellow, keynote at LADIS 2009 workshop

management and allows the root cause analysis to be performed offline. This motivates our crisis-identification methodology that achieves high accuracy and can be easily deployed to large-scale datacenters. We envision that by the time the operator responds to the alarm, she might already have a message in her inbox: "The current crisis is similar to a crisis that occurred two weeks ago. In the former crisis, redirecting traffic to a different datacenter resolved the problem." If the determination of similarity were correct, the operator could avoid tens of minutes of downtime by initiating the same recovery action.

We present a methodology based on statistical machine learning (SML) that exploits two important properties of Web applications. First, the state and the behavior of the system is captured by various performance metrics such as workload rate, request latency, resource utilization, or other application-level metrics. Because many metrics are collected from large number of servers, it is often difficult and time-consuming for human operators to process them, and statistical methods are required to analyze the data. Second, most systems behave normally most of the time, which allows us to create statistical models of *normal system behavior* and detect deviations from this model.

The main idea of our crisis-identification methodology is to automatically capture the patterns of all past performance crises using *crisis fingerprints* and identify the current crisis by comparing its fingerprint to past crises. A crisis fingerprint is created by first selecting a set of metrics relevant to this crisis and then comparing their typical values during normal periods to values during the crisis. The fingerprint thus efficiently captures the state of the datacenter during the crisis and uniquely identifies the crisis by representing the change in values of a small set of metrics. We find past crises that are similar to the current one and thus retrieve information about how this crisis was resolved in the past. If no crisis is similar to the current one, we label it as "new type of crisis"; this information is also useful to operators as they can immediately concentrate on debugging the new crisis. Our process for creating and comparing fingerprints scales to very large clusters of thousands of servers.

We use stringent accuracy criteria to validate the approach on four months of data from a production datacenter with hundreds of servers. Our results clearly establish that:

1. When used in a fully-operational setting, our approach correctly identifies 80% of previously seen crises and, on average, identifies the crises ten minutes after they were detected. In contrast, the operators of the datacenter have informed us that automatic identification is still useful up to one hour after a crisis begins, so in 80% of the cases our approach could have reduced the crisis duration by as much as 50 minutes.

2. When given access to all future data, the fingerprints almost optimally discriminate among the different types of crises. These experiments validate that a fingerprint based on collected performance metrics is an effective and compact representation of the datacenter state.

3. The subset of metrics automatically selected and summarized by our approach identifies crises better than competing approaches representative of both current industry practice and the most closely related work [Cohen *et al.*, 2005] which uses a different statistical selection method.

4. Our approach clearly quantifies tradeoffs among false positives, accuracy of identification, and time to identification.

To the best of our knowledge, this is the first time such an approach has been applied to a large-scale production installation with rigorous validation using hand-labeled data.

4.2 Problem and Approach

A typical datacenter-scale user-facing application runs simultaneously on hundreds to thousands of machines. In order to detect performance problems and perform postmortem analysis after such problems, several *performance metrics* are usually collected on each machine every *epoch* (i.e., a short time interval ranging from one to several minutes). Wide variation exists in what is collected and at what granularity; packages such as HP OpenView [HP, 2010], Ganglia [Massie, 2004], and others provide off-the-shelf starting points. Since large collections of servers execute the same code, under normal load balancing conditions the values of these metrics should come from the same distribution; as we will show, we use this intuition to capture the state of each metric and identify unusual behavior.

A small subset of the collected metrics may be *key performance indicators* (KPI's) whose values form part of the definition of a contractual service-level objective (SLO) for the application. An SLO typically specifies a threshold value for each KPI and the minimum fraction of machines that have to satisfy the requirement over a particular time interval. For example, an SLO might require that the end-to-end interactive response time be below a certain threshold value for 99.9% of all requests in any 15-minute interval on at least 90% of all servers.

A performance *crisis* is defined as a prolonged violation of one or more specified SLO's. Recovery from the crisis involves taking the necessary actions to return the system to an SLO-compliant state. If the operators can recognize that the crisis is of a previously-seen type, a known remedy can be applied, reducing overall recovery time. Conversely, if the operators can quickly determine that the crisis does not correspond to any previously seen incident, they can immediately focus on diagnosis and resolution steps, and record the result in case a similar crisis recurs.

Our goal is to automate the *crisis identification* process by capturing and concisely summarizing the subset of the collected metrics that best discriminate among different crises. We next describe our process for doing this, called *fingerprinting* the datacenter, and define a similarity metric between two fingerprints to identify recurring problems.

4.2.1 Fingerprint-based Recognition

A crisis fingerprint is a vector representing the performance state of a datacenter application that characterizes important aspects of a performance crisis. It is based on values of performance metrics and, intuitively, it characterizes which metrics' values have significantly increased or decreased on a large fraction of the application servers compared to their normal values. There are five steps to our fingerprint-based recognition and identification technique.

1. We summarize the values of each performance metric in a particular epoch across all the application servers by computing one or more *quantiles* of the measured values (such as the median of CPU utilization on all servers). We call these quantiles the *metric quantiles*. As we discuss in Section 4.2.2, this summarization scales well with the number of servers.

2. Based on the past values of each metric quantile, we characterize its current value as *hot*, *normal*, or *cold*, representing abnormally high, normal, or abnormally low value, respectively. This process results in a *summary vector* containing one element per tracked quantile per metric, indicating whether the value of that quantile is hot, normal, or cold during that epoch. We discuss this step and the choice of hot and cold thresholds in Section 4.2.2.

3. We identify *relevant metrics* whose quantile behavior distinguishes normal performance from the performance crises defined by the SLO's. The metric selection process is described in Section 4.2.3. This subset of the summary vector for a given epoch is the *epoch fingerprint*. We show that using our subset of metrics outperforms using all metrics and using the three KPI's selected by the operators.

4. Since most crises span multiple epochs, we show how to combine consecutive epoch fingerprints into a *crisis fingerprint*. We use Euclidean distance to determine whether fingerprints of two crises correspond to the same underlying problem. These steps are described in Section 4.2.4.

5. Finally, we observe that in a real operational setting, crises appear sequentially and identification of a crisis can be based only on information obtained from previous crises. We hypothesize that crisis identification will be improved through *adaptation* — updating identification parameters each time a correctly-labeled crisis is added to the dataset. The adaptation procedure is described in Section 4.2.5.

4.2.2 Hot and Cold Metric Quantiles

In the first step, we compactly represent the values of each metric on all servers during a particular epoch. Because servers of the same application typically run the same code, we summarize the metric values across all servers using several quantiles of the observed empirical cumulative distribution over an epoch. Figure 4.1 on the following page illustrates using the median of the metrics. We refer to these quantiles as *metric quantiles*.

We use quantiles instead of other statistics such as mean and variance because quantiles are less influenced by outliers. This summarization of the state of the metrics does not grow as the number of machines increases, so the size of the fingerprint is only proportional to the number of metrics being collected. In addition, there are well known algorithms for estimating quantiles with bounded error using online sampling [Guha and McGregor, 2009],

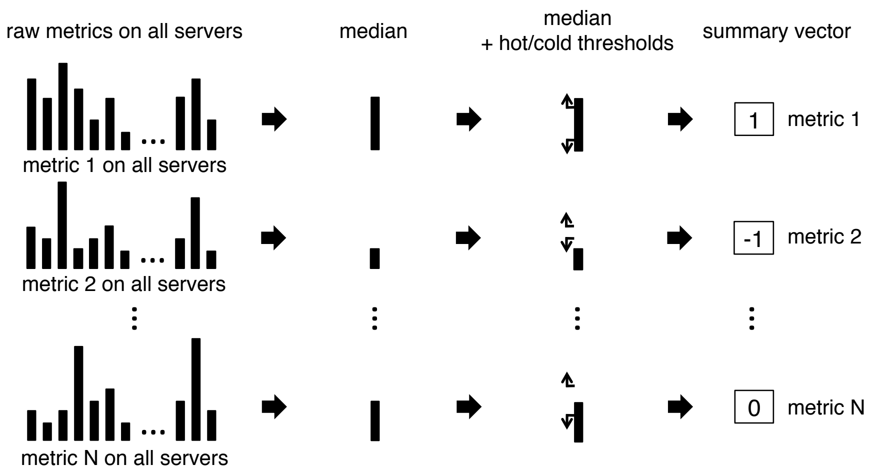


Figure 4.1: The summary vector of a particular epoch is created in two steps. First, the values of each metric are summarized using one or more quantiles; here we use the median. Second, each metric quantile is discretized into a *hot*, *normal*, or *cold* state based on its hot/cold thresholds represented by the arrows. Metric value above the hot threshold – arrow pointing up – is represented as 1, value below the cold threshold – arrow pointing down – is represented as -1, and value between the two thresholds is represented as 0.

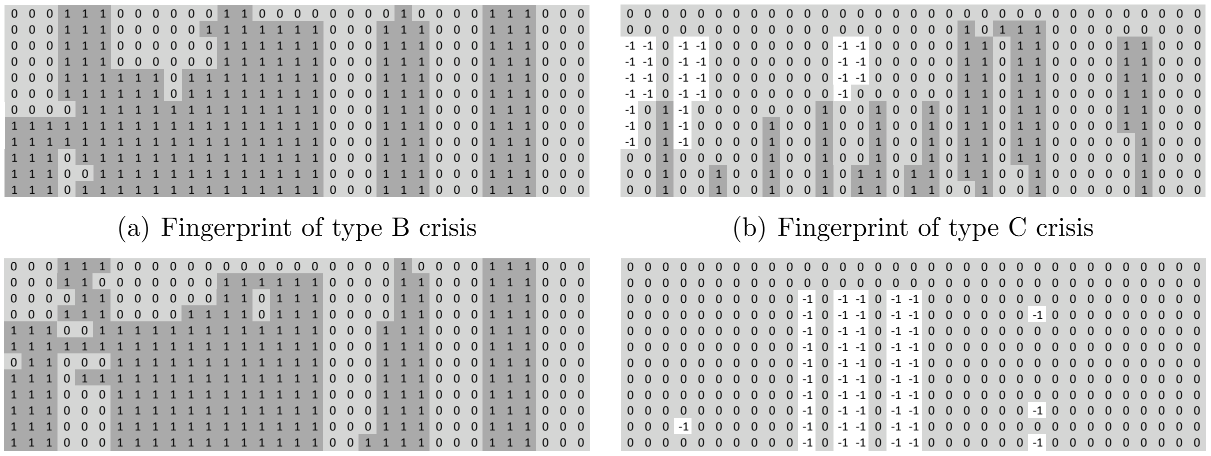
which guarantee that the entries in the fingerprints can be computed efficiently. In our case study, which involved several hundred machines, we computed the values of the quantiles exactly.

We observe that the main factor that differentiates among different types of crises are the different metric quantiles that take extreme values during the crisis. In other words, compared to values of a metric quantile during normal periods with no crises, the values observed during a crisis are either too high or too low.[2] We capture this fact in the fingerprint by encoding which metric quantiles increased or decreased significantly on a large number of servers during the crisis. In particular, we discretize the value of each metric quantile to one of three states: extremely high (*hot*), normal, or extremely low (*cold*) relative to its past values. For example, if the median of a metric is *hot*, the most recent value for that quantile is higher than normal.

The computation of hot and cold thresholds is parameterized by p – the percentage of past values of a metric quantile that are considered extremely low or high during normal system operation. The cold threshold of a particular metric quantile m (such as median of CPU utilization) is computed as the $p/2^{\text{th}}$ percentile of values of m in the past W days that exclude epochs with SLO violations. The hot threshold of m is computed as $(100 - p/2)^{\text{th}}$ percentile over the same period. For example, for $p = 4\%$ we use the 2^{nd} and 98^{th} percentiles. In Sections 4.4 and 4.5 we discuss the choice of quantiles, tuning constants p and W and examine the sensitivity of our approach to their values.

We now define a *summary vector* for one epoch: it is a vector of $Q \times M$ elements,

[2]Because most crisis types are relatively infrequent, we cannot build a robust model of metric quantiles *during* a crisis.



(a) Fingerprint of type B crisis

(b) Fingerprint of type C crisis

(c) Fingerprint of type B crisis

(d) Fingerprint of type D crisis

Figure 4.2: Fingerprints of three types of crises; crisis B repeated several times (see Table 4.1 on page 63). Each row is an epoch and each column represents the state of a particular metric quantile, with dark, light, white corresponding to hot, normal, cold $(+1, 0, -1)$ respectively in the fingerprint. The fingerprints are composed of 11 metrics, where each three adjacent columns represent three quantiles (25th, 50th, and 95th). Notice that the patterns of metric quantiles in crises B are very similar, while very different from crises C and D.

where M is the number of metrics and each group of Q elements corresponds to the metric quantiles of individual metrics. An element's value is -1 if the quantile's value is below the *cold* threshold during the epoch, $+1$ if the quantile's value is above the *hot* threshold, and 0 otherwise (see Figure 4.1 on the previous page).

4.2.3 Selecting the Relevant Metrics

As we will show in our experiments in Section 4.4.1, achieving robust discrimination and high identification accuracy requires selecting a subset of the metrics, namely the *relevant* metrics for building the fingerprints. We determine which metrics are relevant in two steps. We first select metrics that correlate well with the occurrence of each individual crisis by borrowing techniques from machine learning, specifically *feature selection and classification* on data surrounding each crisis. Second, we use metrics most frequently selected in the previous step as the relevant metrics used for building all fingerprints. The summary vector is converted into an *epoch fingerprint* by selecting only the relevant metrics.

Feature selection and classification is a technique from statistical machine learning that first induces a classification function between a set of features (the metrics in our case) and a class (crisis or no crisis) and tries to find a small subset of the available features that yields an accurate function. Let $X_{m,t}$ be the vector of metrics collected on machine m at time t and $Y_{m,t}$ be 1 if m violated an SLO at time t, or 0 otherwise. A classifier is a function that predicts the performance state of a machine, Y, given the collected metrics X as input. The feature selection component picks a subset of X that still renders this prediction accurate.

In our approach, we use logistic regression with L1 regularization[3] as the statistical machine learning method. The idea behind regularized logistic regression is to augment the model fitting to minimize both the prediction error and the sum of the absolute values of model coefficients, which in turn pulls estimates of irrelevant parameters to go to zero, effectively performing feature selection. It has been empirically shown in various settings that this method is effective even in cases where the number of samples is comparable to the number of parameters in the original model [Koh *et al.*, 2007], as is the case in our scenario in which the number of possible features (over 100 per server for several hundred servers) exceeds the number of classification samples. Note that the crises do not need to be labeled when performing the metric selection, so there is no burden on the operator; this step is completely automated.

4.2.4 Matching Similar Crises

In the final step we summarize epoch fingerprints of a single crisis into a *crisis fingerprint* and compare crisis fingerprints using the Euclidean distance metric. Because crises usually last for more than one epoch[4], we create a crisis fingerprint by averaging the corresponding epoch fingerprints, thus summarizing them across time. For example, Figure 4.2 on the preceding page shows epoch fingerprints of four crises. Each row represents an epoch, each column represents a metric quantile, and white, light, and dark represent the values -1, 0, and 1 of the cold, normal, and hot state respectively. The three left-most columns of the top-right crisis would be summarized as $\{\frac{-7}{12}, \frac{-4}{12}, \frac{6}{12}\}$; there are 12 epochs in the crisis and the column sums are -7, -4, and 6. Different types of crises have different fingerprints. For example, in the case of crisis of type B, "overloaded back-end" (see Table 4.1 on page 63), different processing queues, workload, and CPU utilization are abnormally high. In contrast, in crisis of type D, "configuration error 1", most of the workload and CPU utilization metrics are normal, yet some of the internal queues are abnormally low. Notice also that the quantiles often do not move in the same direction; for example, see the left-most three columns in the top-right crisis in Figure 4.2 on the preceding page. Direction of movement of metric quantiles distinguishes different crises.

If the Euclidean distance between the fingerprints of a pair of crises is less than the *identification threshold* T, our method considers the crises identical, otherwise they are considered different. Intuitively, if T is too low, some identical crises would be classified as different (false negative), while if T is too high, different crises would be classified as identical (false positive). We define the false positive rate α as the number of pairs of different crises that are incorrectly classified as identical divided by the number of pairs of crises that are different. We ask the operator to specify an acceptable bound on α, and we set T to the maximum identification threshold that respects that bound. An ROC curve (Receiver Operating Characteristic), such as the one in Figure 4.5 on page 66, is particularly useful for visualizing this. In our experiments, we set α close to zero, essentially guaranteeing no false positives in practice. We illustrate the effects of increasing α on the identification process

[3]See [Young and Hastie, 2006] for more details on Logistic regression.

[4]Duration of an epoch is typically determined by operator's setting of collection interval.

```
Problem identification workflow
When a crisis is detected based on KPIs:
        update hot/cold thresholds (Section 4.2.2)
        update past crises' fingerprint entries (4.2.1)
During first $K$ epochs of crisis (we use $K = 5$, see Sec. 4.3.3):
        update crisis fingerprint with new data (4.2.1)
        find most similar past crisis $P$ (4.2.4)
        if similarity within identification threshold (4.2.5)
          emit label $P$, else emit label $X$
When crisis is over:
        operators verify label of crisis
        update set of relevant metrics (4.2.3)
        update identification threshold $T$ (4.2.5)
```

Figure 4.3: Problem identification workflow. Each line refers to a section that describes that step in detail.

in Figure 4.6.

4.2.5 Adaptation

To identify performance crises as soon as possible in an "online" operational setting, we execute the operations indicated in Figure 4.3 when a crisis is detected through an SLO violation. Since the hot and cold thresholds represent a range of values of a metric quantile during time intervals without SLO violations, they are updated when a crisis is *detected* based on values of metric quantiles in the past W days so as to incorporate the most recent metric values (Section 4.2.2). Because the fingerprints of past crises are determined by these thresholds, we also update the fingerprints. After each crisis, we automatically update the set of relevant metrics based on the most recent C crises and the identification threshold T based on all the past labeled crises to achieve expected false positive rate of α. We discuss the sensitivity of our approach to these parameters in Section 4.5 on page 70. We note that the final verification of the crisis label is performed manually and offline by operators.

4.3 Evaluation

Using the ground truth labels provided by human operators of a production system described in Section 4.3.1, we evaluate our approach and compare it to three alternatives: a) one that relies only on the operator-identified key performance indicators for crisis identification, b) one that creates crisis fingerprints based on *all* available metrics, and c) one that models crises using the *signatures* approach described in [Cohen *et al.*, 2005]. Our evaluation consists of three parts. First, we evaluate the *discriminative power* of our approach and compare it to the other approaches, using the entire available dataset. That is, we

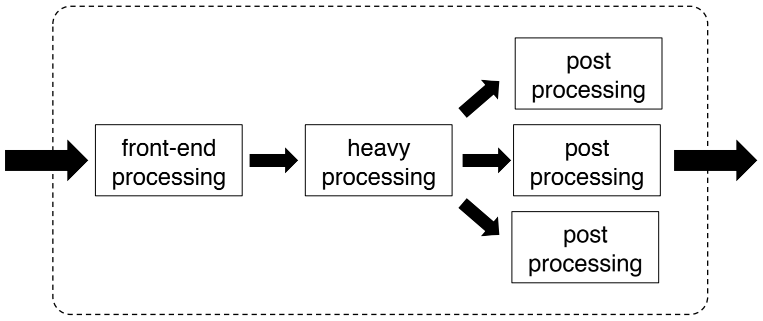


Figure 4.4: Processing on machines in the datacenter under study.

quantify how accurately each approach classifies two crises as identical or not. This part of the evaluation, described in Section 4.3.2, establishes an upper bound on identification accuracy for each approach.

Next, we simulate the operational setting in which our approach is designed to be used, in which crises appear sequentially and identification of a crisis is based only on information obtained from the previous crises. In these experiments we use adaptation described in Section 4.2.5. Section 4.3.3 describes how we evaluate the accuracy and time-to-identification of our technique and quantify the loss of accuracy resulting from the use of only partial information.

Finally, in Section 4.3.4 we compare our approach to the other techniques, again in an operational setting. However, since each approach uses different techniques for adaptation, to make the comparison meaningful we remove the need for adaptation by providing access to the entire dataset for training. We refer to this as operational setting with an *oracle*.

4.3.1 System Under Study

We evaluate our approach on data from a commercial datacenter running a 24×7 enterprise-class user-facing application. It is one of four datacenters worldwide running this application, each containing hundreds of machines, serving several thousand enterprise customers, and processing a few billion transactions per day.[5] Figure 4.4 shows the structure of the application running on most machines. The incoming workload is processed on the machine in three stages: light processing in the front-end, core of the execution in the second stage, followed by some back-end processing. The requests are then distributed to the clients or to another datacenter for archival and further processing. We do not have access to any other data on the clients or servers in the other datacenters.

For each server, we measure about 100 metrics each averaged over a period of 15 minutes. We call these periods *epochs*. The 15-minute averaging window is established practice in this datacenter, and we had no choice on this matter; similarly, we have no access to any other performance metrics or to information that would allow us to reconstruct the actual path of each request through the server. The metrics include counts of alerts set up by

[5]The exact numbers are considered confidential by the company that operates the datacenter.

type of crisis	# of instances	label
A	2	overloaded front-end
B	9	overloaded back-end
C	1	database configuration error
D	1	configuration error 1
E	1	configuration error 2
F	1	performance issue
G	1	middle-tier issue
H	1	request routing error
I	1	whole DC turned off and on
J	1	workload spike

Table 4.1: List of identified performance crises. Names are indicative of the root cause. We stress that almost all these crises manifested themselves through multiple metrics, and that there is overlap between the metrics of the different crises.

the operators, queue lengths, latencies on intermediate processing steps, summaries of CPU utilization, and various application-specific metrics.

The operators of the site designate three key performance indicators corresponding to the average processing time in the front end, the second stage, and one of the post-processing stages. Each KPI has an associated SLO threshold determined as a matter of business policy. A performance crisis is declared when 10% of the machines violate *any* KPI SLO's.

We use four months of production data from January to April 2008. During this period, the datacenter operators manually diagnosed and labeled 19 crises, ranging from configuration problems to unexpected workloads to backlogs caused by a connection to another datacenter. These crises were labeled after an exhaustive investigation according to the determined underlying cause. Attached to the labels are logs providing details about the investigation, and most importantly, the set of remedial actions taken and their effectiveness.

The value of the fingerprinting technique is to accurately identify future crises so that the appropriate information about remedial actions can be readily retrieved and used.[6] In Table 4.1 we provide descriptive labels of the crises and the number of times each type occurred during the period of study. The labels we provide are not the actual ones attached to the operators report; for obvious reasons we are not able to publish these. Yet, as explained above, the actual label is irrelevant to the fingerprinting technique and does not affect the results. Still, the labels in Table 4.1 give a sense of the wide range of problems that are under study and that can be successfully identified with the proposed techniques. We also had access to collected metrics and 20 unlabeled crises that occurred between September and December 2007 which we used to simulate online deployment, but not to evaluate identification accuracy.

[6]Note that issues such as the granularity of the diagnosis and the appropriate mapping to the set of available remedial actions are outside the scope of the fingerprinting technique.

4.3.2 Evaluating Discrimination

Discrimination measures how accurately a particular method classifies two crises as the same or distinct. We compare our method to three others, in each case using the entire dataset to give each method the maximum information possible. This establishes the baseline ability to capture the differences between different crises for each method. As is standard, we compare the different approaches using ROC curves [Lachiche and Flach, 2003], which represent the tradeoff between the false positive rate (incorrectly classifying two different crises as identical) and recall (correctly classifying two identical crises) over the whole range of the identification threshold T. It is standard practice to represent this comparison numerically by computing the area under the curve (AUC). The optimal approach will have an AUC of 1, indicating that there is no tradeoff between detection and false positives. By using an ROC curve for comparison, we take into account all possible cost-based scenarios in terms of the tradeoff between missing identical crises versus considering different crises to be identical.

4.3.3 Evaluating Identification Accuracy and Stability

Identification accuracy measures how accurately our approach labels the crises. A human operator has hand-labeled each crisis in our dataset, but in an operational setting the crises are observed sequentially. Each crisis is labeled *unknown* if the Euclidean distance between its fingerprint and all fingerprints of past crises is greater than the identification threshold T; otherwise it is labeled as being identical to the closest crisis.

Since many crises (indeed, all those in our dataset) last longer than a single 15-minute epoch, we must also define *identification stability*—the likelihood that once our approach has labeled a crisis as known, it will not change the label later while the crisis is still in progress. In each 15-minute epoch, the identification algorithm emits either the label of a known crisis, or the label `x` for *unknown*. A sequence of K identifications is *stable* if it consists of $n \geq 0$ consecutive `x`'s followed $K - n$ consecutive *identical* labels. Since the operators of this application informed us that identification information is useful up to one hour into a crisis, we use $K = 5$. For example, if A and B are labels of known crises, the sequences `xxAAA`, `BBBBB`, and `xxxxx` are all stable, whereas `xxAxA`. `xxAAB`, `AAAAB` are all unstable. Given this stability criterion, a sequence is *accurate* if it is stable *and* the labeling is correct; that is, either all labels are `x`'s and the crisis is indeed new, or the unique non-`x` label matches that of a previously-seen crisis that is identical to the current crisis. Further, for a previously-seen crisis we can define *time to identification* as the first epoch after crisis onset during which the (correct) non-`x` label is emitted.

We emphasize that from the point of view of Recovery-Oriented Computing [Patterson *et al.*, 2002], stability is essential because the system operator's goal is to initiate appropriate recovery actions as soon as the crisis has been identified. Unstable identification could lead the operator to initiate one set of actions only to have the identification procedure "change its mind" later and apply a different label to the crisis, which might have implied different recovery operations. There is an inherent tradeoff between time to identification and stability of identification; we quantify this tradeoff in Section 4.4 and show how a system operator can control the tradeoff by setting a single parameter in our algorithm.

4.3.4 Comparing to Other Approaches

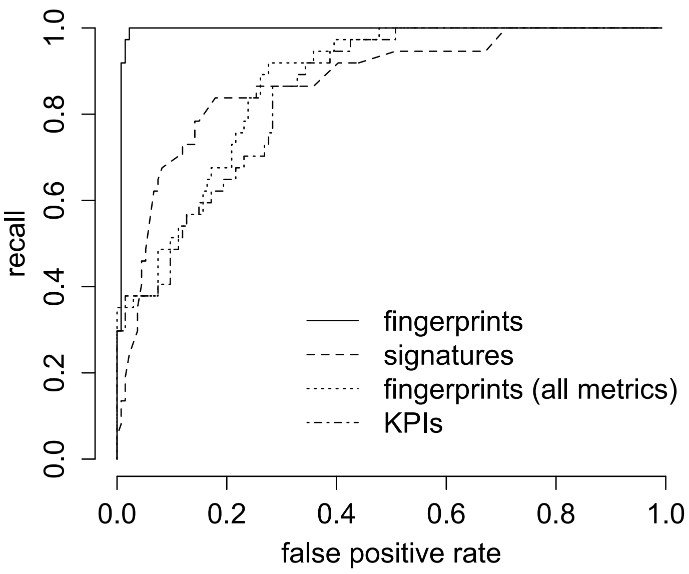
To make meaningful comparison of identification accuracy of all considered approaches, we first eliminate the adaptation described in Section 4.2.5. With adaptation in place, any comparison would also have to compare the relative loss of accuracy of each method when only partial information is used to make decisions. Instead of adaptation, we use an *oracle* to set the best parameters for each method, allowing us to compute the decrease in accuracy caused by estimating the method's parameters online.

To remove adaptation from the fingerprinting approach, we compute the identification threshold T based on an ROC curve over all labeled data, we select a single set of relevant metrics using models induced on all the labelled crises, and we compute hot and cold thresholds based on the whole dataset. We use this set of parameters throughout the experiment.

To explain how we remove adaptation from the signatures approach in [Cohen *et al.*, 2005], we first briefly review the adaptation it usually performs. The signatures approach builds a classifier for each crisis that predicts whether the current system state will result in an SLO violation. The SLO state serves as ground truth for the classifier, and the *signature* captures the subset of metrics that form the features used by the classifier that achieves the highest accuracy. Crisis recognition consists of first selecting a subset of the models with highest prediction accuracy on the current crisis, and then building a signature based on the most relevant metrics. This entire procedure requires setting many parameters, including the number of epochs on which the model is evaluated and the number of models being selected. Adaptation consists of periodically merging similar models and deleting inactive/obsolete models [Zhang *et al.*, 2005]; these processes depend on additional free parameters.

To remove adaptation from the signatures approach, we allow it to always select the optimal model for each crisis, which gives this approach a model management technique that is omniscient and clairvoyant. In addition, since our system consists of hundreds of servers rather than the handful of servers used for evaluation in [Cohen *et al.*, 2005], rather than assigning a model to each server we assign a model to the datacenter and summarize the metrics using quantiles. Finally, in place of the naive Bayes models used in [Cohen *et al.*, 2005], we use logistic regression with L1 regularization for feature selection; since the logistic regression models were more accurate in our setting than those using naive Bayes, this gives the signatures approach another advantage.

We point out that our criteria for accuracy are much more stringent and more realistic than those used in [Cohen *et al.*, 2005]. In that work, an identification was considered successful as long as the actual crisis was among the k most similar crises selected by the algorithm, according to a distance metric. In contrast, our identification is successful if the single correct label is produced. In case the crisis is a previously-unseen crisis, identification is successful only if the algorithm consistently reports "unknown crisis." Furthermore, a prerequisite to accuracy of our approach is the stability criterion, which has no analogue in [Cohen *et al.*, 2005].



type of fingerprint	AUC
fingerprints	0.994
fingerprints with all metrics	0.873
KPIs	0.854
signatures	0.876

Figure 4.5: ROC curves for crisis discrimination and the area under the curves (AUC) for the fingerprinting and alternative approaches.

4.4 Results

The main results reported in this section used the following settings. The fingerprints were built using three quantiles for each metric: median, and the 25th and 95th percentiles, to capture the variance. In the selection of the relevant metrics we used classifier models containing ten metrics (the balanced accuracy was high enough and the standard deviation in the cross-validation was low), and we used the most frequently selected 30 metrics over the past 20 crises as the metrics for the fingerprints (see Section 4.2.3). Finally we used a moving window of 240 days to set the hot/cold thresholds using $p = 4\%$. Section 4.5 on page 70 describes the sensitivity analysis and suggests how to set all these parameters in a realistic setting.

4.4.1 Discriminative Power

As discussed in Section 4.3.2, we start our evaluation of the fingerprint approach by examining its ability to classify two crises as the same or distinct, and compare it to alternative approaches. The graph in Figure 4.5 shows the ROC curves and AUC for each approach. The fingerprint approach exhibits an AUC of 0.994, which means that in terms of discrimination, this approach is able to maximize the detection rate with extremely few false positives. Comparing to the alternative approaches, using the KPIs gets an AUC of 0.854 and fingerprints based on all the metrics achieve 0.874. Furthermore looking at the shape of the curves it is clear that neither is able to discriminate at the same level as the fingerprints. Using the KPIs simply does not provide enough power to discriminate between the different types, and using all the metrics obfuscates the discriminative signal by introducing noise.[7] Finally, the signatures approach performs better than both these two approaches but still well below the fingerprinting approach.

4.4.2 Fully Operational Setting

In the following experiments we simulate the conditions under which the approach would be used for identification of crises in an operational setting. In this setting, the crises arrive one at a time and we update fingerprinting parameters as we observe them. Because the results could be affected by the order of the crises, we perform experiments using 20 random permutations of the crises, one of which is the actual chronological order, and report the average accuracy across all runs. We update the fingerprint parameters as described in Section 4.2.5 on page 61.

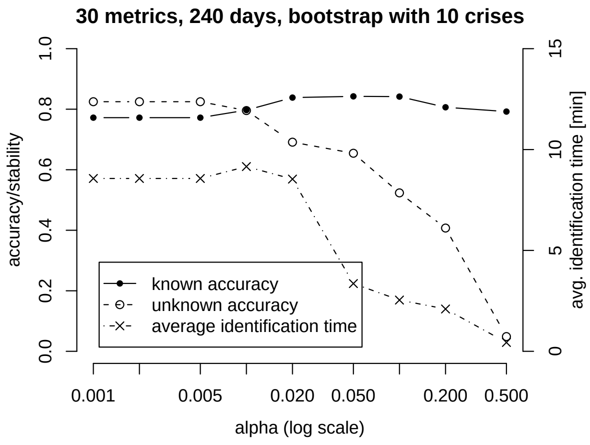
Recall from Section 4.3.3 that a new crisis C is said to be identified accurately if the identification is *stable* over five epochs (i.e., one hour) and if the label is correct. A crisis C is previously known if the set of crises that occurred before C contains crisis D identical to C. The correct label for C would thus be the label of crisis D. If the past set of crises contains no crisis identical to C, the correct label for C would be *unknown*. We use three evaluation metrics: *known accuracy* (fraction of correct identifications for previously seen crises), the *unknown accuracy* (fraction of correct identifications for previously unseen crises), and also the *time to identification* (the average time between crisis detection and its correct identification).

To evaluate the performance of our method with adaptation, we run three sets of experiments. In each experiment we use a different number of labeled crises to bootstrap the fingerprinting process (i.e., select the initial metrics and identification threshold). When starting with two labeled crises, we achieve known and unknown accuracy of 78% and 74%[8]. Experiments that start with five and ten crises both yield accuracies of approximately 76% and 83% (see Table 4.2 and Figure 4.6 on the following page).

Besides accuracy, the time at which the method makes a decision regarding the identity of the crisis is important to the operator. We illustrate the dependency among the three evaluation metrics in Figure 4.6 on the next page. We note that our method is able to make the identification within ten minutes of crisis detection, even in a fully operational setting where the relevant metrics and identification threshold are adapted in an online fashion. Operators of this web application mentioned that correct identification is useful even one hour after the crisis was detected. Our hypothesis is that using performance metrics

[7]Metrics that are not correlated with the crises may take extreme values during both crises and normal intervals.

[8]We report accuracies for $\alpha = 0.001$ – conservative value that guarantees almost no false positives.



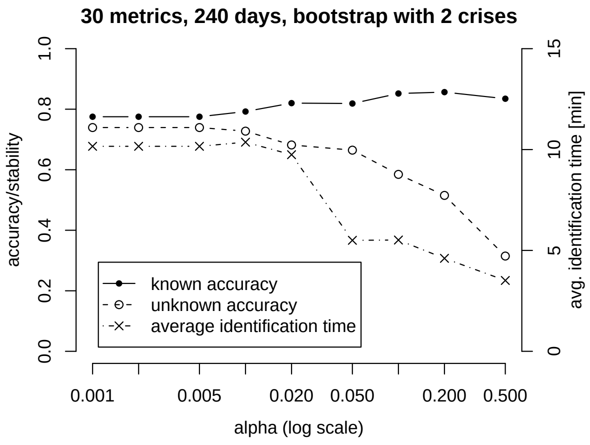


Figure 4.6: Known accuracy, unknown accuracy, and time of identification results in **full operational setting** when using 30 metrics in fingerprints, 240 days of moving window, and bootstrapping with ten and two labeled crises (top). X-axes represent different values of the false positive rate parameter specified by the operator. We report results for $\alpha = 0.001$.

Operational setting	Known acc.	Unknown acc.
oracle	98%	93%
adaptation, bootstrap w/ 10	77%	82%
adaptation, bootstrap w/ 5	76%	83%
adaptation, bootstrap w/ 2	78%	74%

Table 4.2: Summary of the results for different settings.

with finer granularity (e.g. one minute instead of 15 minutes) would result in even faster identification, but we are unable to verify this hypothesis without additional data.

4.4.3 Operational Setting with an Oracle

To compare our approach to the three alternative approaches, we eliminate the adaptation as described in Section 4.3.4. Instead, each of the four methods uses the best settings of its parameters based on the whole dataset as if provided by an oracle and we do not update the parameters for new crises. For example, we select the metrics for fingerprints based on all available crises and set the identification threshold to best discriminate among them. Setting the parameters based on all available data allows us to quantify the loss of accuracy from online estimation of these parameters in the operational setting. Note that in contrast to the fully operational setting, in this setting it does not make sense to compare approaches in terms of average time to identification.

Because the signatures approach uses the optimal model, it will not change its decision during the evolution of each crisis. In full operational setting, the signatures method may change its initial assessment of the crisis as it evaluates the ensemble of models [Zhang *et*

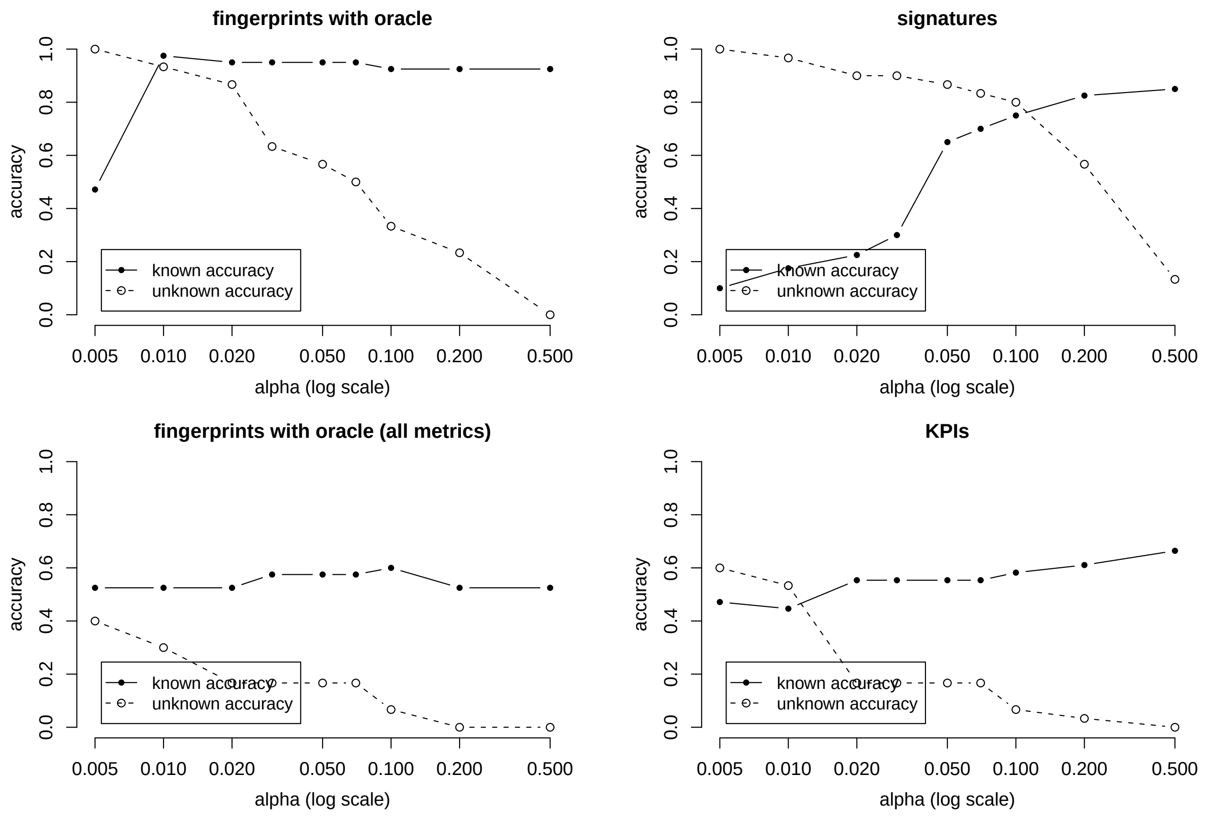


Figure 4.7: Known accuracy, unknown accuracy, and time of identification for different approaches when using an **oracle**: fingerprints (top left), signatures (top right), fingerprints using all the available metrics (bottom left), and KPIs (bottom right). X-axes represents different values of the false positive rate parameter specified by the operator.

al., 2005], yet when we provide the optimal model we are sidestepping this issue.[9]

For each approach we executed five runs with different initial set of five crises and performed identification on the remaining 14 crises. The initial set of crises always contained two crises of type “B”, one of type “A”, and two other crises that varied in each run. Because crises “A” and “B” are the only ones that repeat, we use them in each initial set to estimate the identification accuracy of known crises. We report the average of all the evaluation metrics across the five runs in Figure 4.7.

Fingerprinting achieves very high known and unknown accuracies of 97.5% and 93.3%. The approaches based on using all the metrics and the KPIs achieve accuracies of approximately 50% and 55%, respectively. The signatures approach [Cohen *et al.*, 2005] performs better than the baselines and achieves accuracies of 75% and 80%, but it is still worse than fingerprinting.

[9]We emphasize that using the optimal model actually favors the evaluation of the signature technique, as it is not guaranteed that the ensemble of models will find the optimal model.

4.4.4 Summary of Empirical Results

Based on the discrimination and identification experiments, we conclude that:

1. Maintaining only *relevant* metrics for distinguishing crises from normal operation allows the fingerprinting approach to attain much higher identification accuracy.

2. The KPIs provide insufficient information to distinguish or identify different types of crises.

3. The fingerprinting approach, based on keeping concise representations of the datacenter state, performs significantly better than the signatures approach of [Cohen *et al.*, 2005].

4. In the realistic, fully operational setting, we correctly identify 80% of the crises with an average time between crisis detection and its identification of ten minutes.

4.5 Fingerprinting in Practice

As with any approach based on collecting and analyzing data, applying the approach in an operational setting requires setting some parameters and understanding the sensitivity of the technique to these parameters. As described in Figure 4.3 on page 61 and in Section 4.2.5, we automatically update fingerprinting parameters when a new crisis is detected and after it is over. These updates are based on a set of tuning constants that may not require any changes, but should be reviewed periodically or when a major change occurs in the datacenter. These tuning constants include the number of metrics used in the fingerprints, number of days W and percentage p of metric values considered extreme when computing hot and cold thresholds, and α used when computing identification threshold T.

The effects of the tuning constants on the identification accuracy and time to detection could be estimated offline by running identification experiments using past labeled crises. The operators would use ROC curves and graphs such as the ones in Figure 4.6 on page 68, to select a suitable operating point of datacenter fingerprinting. Producing these graphs took only a few minutes for the 19 crises we used in our experiments. After selecting the tuning constants, the adaptation of the fingerprinting parameters and the identification algorithm proceed automatically with little intervention by the operators. To illustrate the approach, we report on some of the experiments we performed in order to set the tuning constants and study their effect on the results.

In our reported results we use $p = 4\%$ as the percentage of metric values considered extreme when computing the hot and cold thresholds. When experimenting with values of 2%, 10% and 20% we observe that the area under the ROC curve (as in Section 4.4.1) decreased from 0.99 to 0.96 – a small change and still far better than the competing approaches.

Instead of using all three quantiles when summarizing metrics, we also tried using just the median, which reduced the identification accuracy by 2 to 3 points in the fully operational setting. In the oracle setting, the accuracy decreased by 5 points which is still better than competing approaches as reported in Sections 4.4.1 and 4.4.3 and illustrated in Figures 4.5

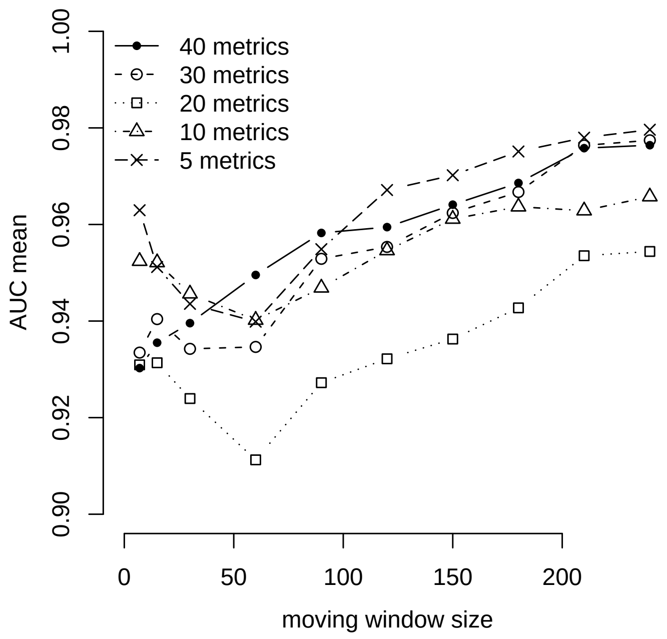


Figure 4.8: Area under the ROC curve for different combinations of the moving window size and the number of metrics used in fingerprints.

and 4.7 on page 69. Our intuition is that some pairs of crises are distinguished by three quantiles that do not all move in the same direction, and observing the movement of only a single quantile would necessarily fail to capture such differences.

We summarize the affects of changing the number of metrics and size of the moving window in Figure 4.8. Note that reducing the window size also reduces the AUC for fingerprints with 20 or more metrics. Using five or ten metrics and small window size allows the fingerprints to quickly adapt to change in the metric values which results in increase in AUC.

Also, as mentioned in Section 4.2.4, we update the identification threshold T to avoid false positives (α set to 0.001). In Figure 4.6 on page 68 we show the effects of increasing the false positive rate α (which in turn affects T). In our dataset, increase in α results in only small increase in known accuracy, but significant decrease of unknown accuracy for $\alpha > 2\%$.

Finally, when comparing two crises, we first compute the crisis fingerprints by averaging the corresponding epoch fingerprints. In all the experiments in Section 4.4, we average across epochs -30 minutes, $\ldots$, 60 minutes, relative to the start of the crisis (the limit of 60 minutes was set by the datacenter operators). Figure 4.9 on the next page shows that time intervals that start at least 30 minutes before the beginning of the crisis quickly achieve high levels of discrimination.

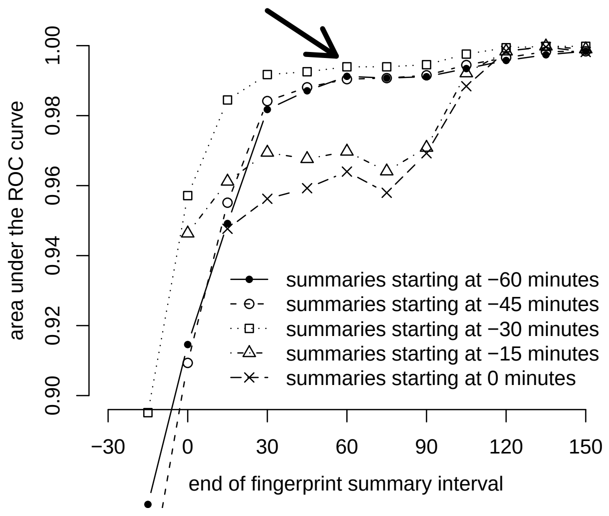


Figure 4.9: Area under the ROC curve (discriminative power) of fingerprints when summarized over different time intervals. Each line on the graph represents intervals that start at the same epoch (relative to the start of the crisis), while the x-axis represents the end of the interval. The arrow points to AUC corresponding to the interval ⟨−30minutes, +60minutes⟩ used in all our experiments.

4.6 Related Work

4.6.1 Crisis Signatures

By far, the work closest in spirit to our own is the signatures approach to identifying and retrieving the essential system state corresponding to previously-seen crises [Cohen *et al.*, 2005]. The authors propose a methodology for constructing signatures of server performance problems by first using machine learning techniques to identify the performance metrics most relevant to a particular crisis; second, using the induced models for online identification; and third, relying on similarity search to recognize a previously recorded instance of a particular incident. They showed their approach to be successful in a small transactional system on a handful of performance problems.

We view our methodology as a direct descendant of the signatures approach but with important differences that lead to several crucial improvements. Our fingerprinting approach is based on treating the problem as an online clustering based on the behavior of the metrics during a crisis. The signatures approach is based on maintaining multiple models (one per crisis), which are then managed by computing a fitness score to decide which of the models is likely to provide the best identification of the current crisis. The selected models are then used to construct the signature of the crisis and identify it. As our results in Section 4.4 demonstrate, these differences lead to a substantial improvement in accuracy.

Our fingerprinting method also provides additional advantages. First, fingerprint size scales linearly, rather than exponentially, with the number of metrics considered. Second, the fingerprint size is independent of the number of machines and fingerprinting can thus be applied to very large deployments. Finally, because we do not use multiple models for crisis identification, we avoid two related sources of potential error and their corresponding free parameters. In particular, we do not need policies for maintaining and validating multiple models nor a method for selecting the best models and combining their outputs.

Although the findings of [Cohen *et al.*, 2005] were encouraging, the method was evaluated on modest workloads running on few servers and using generous criteria for identification accuracy. In reality, today's applications run on hundreds up to tens of thousands of machines in a datacenter, and since the goal of problem identification is to provide actionable information for initiating recovery, the evaluation criteria should be stringent.

4.6.2 Using Machine Learning to Identify Operational Problems

As early as 2003, [Redstone *et al.*, 2003] proposed the use of compute-intensive modeling techniques to automatically diagnose application failures. Since then, researchers have tried to identify operational problems by analyzing performance metrics using machine learning [Cohen *et al.*, 2004; Cohen *et al.*, 2005; Zhang *et al.*, 2005; Bodík *et al.*, 2008; Yuan *et al.*, 2006; Pertet *et al.*, 2007; Duan and Babu, 2008], by identifying unusual or noteworthy sequences of events that might be indicators of unexpected behavior [Reynolds *et al.*, 2005; Chen *et al.*, 2004], and by manually instrumenting the system [Barham *et al.*, 2004] and creating libraries of possible faults and their consequences [Yemini *et al.*, 1996]. Others have laid out general methodological challenges in using computers to diagnose computer problems [Goldszmidt *et al.*, 2005; Cook *et al.*, 2007].

The results of the HiLighter tool [Bodík *et al.*, 2008] showed that the use of regularized logistic regression [Young and Hastie, 2006] as a classifier results in a metric selection process that is more robust to noise than the naïve Bayes classifier used in the signatures approach. In particular, HiLighter avoids the heuristic search for the relevant features of each model which was used in the signatures approach. However, like in signatures, HiLighter proposes a representation of metric state that grows exponentially with the number of metrics

Our fingerprinting methodology assumes that the operator is able to correctly label a crisis after its resolution. In [Woodard and Goldszmidt, 2009], the authors pose the crisis identification as an unsupervised online clustering problem that does not require labeled crises. They first model the evolution of each crisis using a Markov chain and then cluster the crises using a model based on the Dirichlet Process.

In an earlier project [Bodík *et al.*, 2005], we applied Statistical Machine Learning to *detection* of application failures. We built a tool that used the Chi-squared statistical test to detect anomalies in user workload to a Web site and would identify the pages with the most significant change in traffic. Moreover, the tool provided an intuitive visualization of the traffic patterns that allowed the operator to quickly verify the alert generated by our anomaly detection algorithm. The visualization of failure fingerprints (see Figure 4.2 on page 59) was equally helpful as it allowed the operators to rapidly understand the evolution of the crisis on hundreds of servers. This visualization addresses a key question of autonomic computing of how to win operators' confidence so that these new tools can be embraced.

4.7 Summary

In this chapter we described a methodology for constructing datacenter fingerprints using statistical techniques. The goal of the fingerprints is to provide the basis for automatic classification and identification of performance crises in a datacenter. Different from root-cause diagnosis, identification facilitates rapid online repair of service disruptions, allowing potentially time-consuming diagnosis to occur offline later. The goal of rapid recovery is consistent with statements by leading Web application operators that today's 24x7 Web and cloud computing services require total downtime to be limited to 50 minutes per year[10].

Under realistic conditions and using real data and very stringent accuracy criteria, our approach provides operators the information necessary to initiate recovery actions with 80% accuracy in an average of 10 minutes, which is 50 minutes earlier than the deadline provided to us by the operators. Indeed, our criteria may be more stringent that required in practice, since operators may just want to see a list of candidate crises most similar to the current one. We conjecture that our technique also directly applies to "virtual clusters" in cloud computing environments. If the same physical cloud is shared by many operators through a combination of physical and virtual isolation, the technique applies to each operator's "subcluster", but we have no information on whether it works across operators' applications. We hope that our results will inspire researchers to test and improve on these techniques for multi-tenant environments.

Furthermore, the visualizations of the fingerprints themselves are readily interpretable by human operators: when we showed a few of the fingerprints to the application operators, they quickly recognized most of the corresponding crises, even though they are not experts in machine learning. Interpretability and gaining operator trust are important for any machine learning technique that will be used in an advisory mode in a production environment.

[10]Marvin Theimer, senior principal engineer, Amazon Web Services; keynote at LADIS 2009 workshop.

Chapter 5

Future Work

5.1 Workload Spike Modeling and Synthesis

Our work on workload spikes was limited mainly by the availability of data. Our analysis should be repeated with data from many more spikes to verify our findings and understand the frequency and magnitude of spikes. The same analysis should also be performed at different timescales – from milliseconds up to hours and days.

The current approach to stress-testing systems, in industry and academia, is to replay a previously-captured workload trace or to run a simple synthetic benchmark. Parameterized workload models, such as the spike model we proposed in Chapter 2, suggest a different approach where the model is used to generate more and more difficult workloads and to find where the system breaks. For example, when stress-testing against spikes, one could iteratively increase spike steepness, magnitude, or the number of hotspots.

5.2 Director

Director for Multi-tier Applications

Our Director framework could be easily applied to any system for which we can model the impact of the available actions on performance of the system. In our case, creating a model of SCADS was relatively simple because we only had to consider workloads on individual servers. However, more complex applications are composed of multiple tiers and depend on many other software services which could also be shared with other applications in the datacenter. Building a single performance model for such complex applications might be infeasible because the model would have to predict end-to-end request latency based on workloads on each tier and service.

Another option would be to decompose the end-to-end performance SLO into SLOs for individual tiers and services. We could then create a much simpler model for each tier and service and use separate Director control loops. Such a decomposition could be based on request execution paths observed by using path-tracing tools such as X-Trace [Fonseca *et al.*, 2007].

Director for MapReduce Frameworks

MapReduce and Hadoop frameworks [Borthakur, 2007; Dean and Ghemawat, 2004] could also benefit from model-driven resource allocation policy. The state-of-the-art batch job schedulers try to achieve fairness or improve data locality [Isard *et al.*, 2009; Zaharia *et al.*, 2010]. Ideally, however, the scheduler would directly minimize the duration of each job or of all the submitted jobs.

Modeling could thus be utilized at two levels: for individual jobs and for the whole scheduler. The duration of each job is determined by parameters such as number of mappers and reducers, the amount of RAM, the size of input data and others. By capturing the effects of the parameters on job duration in a statistical model, we could optimize over all the parameters to minimize the duration of the individual jobs. Early work on modeling Hadoop jobs is presented in [Ganapathi, 2009].

Most MapReduce schedulers execute many jobs concurrently from a long queue of submitted jobs. By using a per-job model, the framework could find the best schedule and resource allocation for individual jobs. For example, the scheduler could answer questions like: is it better to run jobs A, B, and C concurrently, or run two of them concurrently and the third one later? Or, if running all three jobs concurrently, what is the best way to divide the cluster resources among the three jobs to minimize the duration of the slowest job? Such models would also enable the users to specify per-job deadlines and other SLOs.

Optimizing Director using Policy Simulation

Most resource allocation frameworks have parameters that significantly affect their performance. For example, an operator using our Director framework has to specify two workload smoothing parameters and an overprovisioning parameter. Extreme values of these parameters usually lead to poor system performance or wasted resources. Therefore, it is important to tune these parameters to improve performance and reduce resource usage.

Because model-based resource allocation frameworks already have a performance model of the system, it is possible to construct a control policy simulator which could be used to find the optimal values of the parameters. The policy simulator would take workload and parameter values as input, execute the actual control policy, and estimate the performance of the system using the model. Optimization techniques like hill-climbing could be used to search over the parameter space and optimize the policy. We demonstrated this technique in [Bodík *et al.*, 2009b].

5.3 Fingerprinting

Using More Metrics

When constructing fingerprints, our current approach uses only metrics already captured by the monitoring system. Using more detailed system metrics could lead to better discrimination among crises. More metrics could be added by automatically parsing application log files or by adding path-tracing to the application.

Application logs are frequently used for debugging Web applications, but because of extremely large size of these logs, automated processing is often necessary to extract useful information on the behavior of the system. As was demonstrated in [Xu *et al.*, 2009], with a little effort, free-text log files can be parsed and used to automatically detect anomalies in computer systems. These methods could be adapted to create additional performance metrics which could be very useful for crisis identification.

In most applications, the collected performance metrics only measure the performance of each software service independently of the rest of the system. Cross-layer path tracing methods, such as X-Trace [Fonseca *et al.*, 2007], could be used to trace the execution of user requests across the whole software stack. Such tracing would make it possible to collect metrics that capture the behavior of requests *between* services. These metrics could significantly improve the understanding of the system performance during a crisis.

Extending Fingerprints to Multi-service and Distributed Applications

Current datacenters run thousands of applications which often share hundreds of software services in the datacenter and also share hardware and network equipment. Moreover, a single application could run in tens of globally-distributed datacenters. While some failures only affect a single application in one datacenter, other failures could affect the same application in many datacenters, or different applications in the same datacenter. For example, a failed network switch could affect all servers connected to that switch and a simple configuration error could impact the same application in all datacenters.

In such a complex environment with many dependencies, it is very difficult to identify and diagnose performance problems. Fingerprinting could be applied in two ways: we could build a single fingerprint for the whole datacenter (or multiple datacenters), or we could build fingerprints for individual software services running in the datacenters.

While creating a single fingerprint for one or multiple datacenters would capture the state of the whole system during a crisis, it might not result in accurate crisis identification for two reasons. First, during most crises, only a small part of the whole datacenter is affected by the crisis and thus only a small part of the fingerprint would be relevant for identification. During identification, the irrelevant parts of the fingerprints could thus significantly obscure the important differences. Second, most datacenters evolve very quickly by adding more hardware and more services, or rewriting older services. Therefore, these fingerprints could get stale and unusable very quickly.

Creating fingerprints for individual software services would be more feasible. During a crisis, SLO violations would be used to determine which services are affected by the current crisis. To identify the crisis, only fingerprints of the affected services would be compared.

Automatically Suggesting Fixes to Common Problems

Many Web applications are built using the same building blocks, such as the Ruby on Rails framework or Amazon AWS services, and thus potentially share the same types of performance problems. In many cases, a developer might have to spend a considerable amount of time trying to resolve an issue that many others have experienced before. It would thus be beneficial to create a public “recommendation” service which would point to the source of the problem or recommend a solution. Such service would be similar to Microsoft Windows recommending fixes after application or driver crashes.

While fingerprinting could be used as the foundation for such a service, there are many open questions. For example, how do we identify the set of performance problems that are shared among the different applications? Or, how do we create a fingerprint that captures only metrics shared among all the applications and does not use any information specific to a particular application?

Chapter 6

Lessons Learned and Conclusions

In our everyday lives, we use increasingly complex Web applications distributed across many datacenters. However, only a few experts understand the workloads, performance, and failures of these systems. In this thesis, we used Statistical Machine Learning (SML) to address three problems of datacenter operations: characterization and synthesis of workload spikes in stateful systems, dynamic resource allocation in stateful systems, and automatic identification of recurring performance crises. In this concluding chapter, we summarize our contributions and the lessons learned along the way, and offer suggestions for future applications of SML to systems problems.

6.1 Summary of Contributions

In Chapter 2, we addressed the problem of characterization and synthesis of workload spikes and data hotspots in stateful systems. We analyzed traces of five real spikes and concluded that these spikes vary significantly along many important dimensions. Based on this analysis, we proposed a statistical model that captures most of the variance we observed in these five spikes. This model could thus be used to synthesize realistic workload and stress-test systems against such spikes.

In Chapter 3, we focused on designing control policies for elastic stateful systems. The policies enable automatic scale up and scale down in response to changes in application workload. We first formulated three principles of designing such dynamic resource allocation frameworks that have to maintain strict performance Service-Level Objective expressed using high quantiles of request latency. Second, we designed and implemented Director – a modular control framework based on Model-Predictive Control – which uses a statistical performance model of the system. Finally, we demonstrated that this framework can respond to unexpected workload spikes with data hotspots and can also handle periodic diurnal workloads.

In Chapter 4, we addressed the problem of identification of recurring performance crises in large-scale Web applications. We proposed a methodology for constructing datacenter fingerprints that succinctly capture the state of a system during a failure and could thus be used to quickly identify a crisis in progress. In our evaluation on data from 19 real crises in a large-scale Microsoft application, the fingerprints correctly identified 80% of the crises on average ten minutes after crisis detection. One of the most important components of the fingerprinting process was a statistical method for automatic selection of metrics captured in a fingerprint.

6.2 Lessons Learned

6.2.1 Guidelines for Applying SML to Systems

Based on our experience of applying SML to problems in computer systems, we formulate a guideline that should be followed when applying SML to systems, consisting of the following four steps. First, make sure that SML is an appropriate method for solving this problem. Second, translate the problem into the SML domain. Third, collect the data used to fit the SML model. Finally, select a particular SML method and its implementation. In the following paragraphs, we describe these steps in more detail.

Step 1: Is SML appropriate for this problem?

Before applying SML, one should consider other approaches to solving the problem. While using SML is advantageous for extracting patterns from data, finding anomalies, or identifying clusters of crises, it might not be suitable in all circumstances. Answering the following questions helps clarify if SML is the appropriate method to use. In general, models created using SML are not 100% accurate; will the system be able to deal with the inaccuracy? Could the inaccuracy hurt the system? How will the data for model fitting be collected? Instead of predicting a particular quantity, could it be directly measured in the system? Could the knowledge of the system be used to create an analytic model instead of fitting a black-box SML model?

For example, for the crisis identification problem, both the previous work [Cohen *et al.*, 2005] and our work in Chapter 2 showed that "naive," non-SML-based methods result in low crisis identification accuracy. On the other hand, SML methods that can detect patterns in performance metrics are far more accurate.

Step 2: Translating the problem into SML domain

After one has decided on using SML, the next step is to translate the systems problem into the SML domain. In other words, do we want to solve the systems problem using regression, clustering, classification, anomaly detection, or some other technique? It is important to point out that one systems problem could be translated in different ways, each having its own tradeoffs. We illustrate this process on the performance modeling problem in the Director framework.

The goal of the Director performance model is to predict SLO violations based on the system workload. Because the SLO could be defined using different latency quantiles (for example, 90^{th}, 99^{th}, or 99.9^{th}), our first approach was to fit the full latency distribution using a parametric model and relate its parameters to the system workload. However, the empirical latency distributions are not an exact match to most of the parametric distributions and thus achieving an accurate prediction at all quantiles would be difficult. Our second approach was to create multiple models, one for each SLO quantile, and use quantile regression. Instead of fitting the mean of the observed data, quantile regression directly fits the specified quantile. While this method exactly meets our goal, because it is not used in practice very often, this choice severely limits the number of available open-source packages we could use. The third approach was based on binning the data into short time intervals, computing the specified quantile for each interval, and using a mean regression to directly fit the quantile as a function of the workload during each interval. Such approach had two advantages: there are many more methods for mean regression than for quantile regression and the regression has to fit much less data (instead of using latency of all measured requests, we only use one datapoint per time interval). However, it turned out to be difficult to find an accurate and robust latency model that monotonically increased with workload and increased to "infinity" after reaching the maximum throughput of a server. Because the Director only needs to determine whether a particular server is overloaded, our final approach in Chapter 3 used binary classification. Instead of fitting the specified percentile of latency, we labeled each time interval as 0 or 1 based on whether the SLO was violated or not during that interval, and used a linear model to classify the data. Compared to the previous approach, this simpler model eliminated the need to accurately fit the latency, but provided the same benefits in the Director framework. This example demonstrates the different trade-offs one has to consider when translating a systems problem into the SML domain.

Step 3: Collecting data

The datacenter applications and their workloads frequently change, therefore the system will have to continuously collect data to refresh the various models. Because data collection has costs, there are various trade-offs associated with it. For example, collecting data too frequently could have a negative impact on system performance, but having more data would allow us to update the model more frequently or have a more accurate model. Instead of collecting data from a production system, we could measure the system in a benchmark, but the behavior of a system in a benchmark might not match the production version.

Often, the metrics being collected from a system do not suffice to model system performance and additional system instrumentation is necessary. For example, SCADS, the storage system used in evaluation of the Director framework, originally collected only workload and latency metrics for high-level queries. After additional instrumentation, we were able to collect more detailed metrics of the lower-level SCADS operations which allowed us to fit a more accurate performance model of SCADS. In cases like this, the burden falls on the SML expert to convince the system designer that the benefits of the additional instrumentation outweigh the cost of implementation and the performance overhead.

Step 4: Selecting a particular SML method

After translating the problem to an SML domain, we have to pick the actual method and its implementation. For example, we translated the performance modeling problem to a classification problem and now we have to decide which of the many classification algorithms to use. There exist a large number of different methods for regression, classification, or clustering, and their characteristics have significant impact on the particular systems problem. Here we briefly summarize the different characteristics and trade-offs one has to consider before settling on a particular method.

First, we need to consider the **assumptions** the model is making about the data. For example, linear regression assumes that the relationship between the response variable and the feature variables is approximately linear, while non-parametric regression methods do not make such strong assumptions. If the data do not satisfy the assumptions of the model, the resulting model could be very inaccurate. One could use non-linear features in linear regression to fit more complex models.

The different SML methods also differ in the **amount of data necessary** to fit an accurate model. While models with stricter assumptions might be more difficult to fit to a particular dataset, in general, they require fewer data to fit an accurate model. In general, one needs little data to fit a linear model, but much more data to fit an accurate non-parametric model. The amount of necessary data also grows exponentially with the number of features. When the amount of data available for training is low, one should consider methods that incorporate regularization into the fitting process. For example, in the HiLighter project [Bodík *et al.*, 2008], the precursor of fingerprints, we used logistic regression with L1 regularization to fit accurate linear models based on a dataset with 2000 features but only 86 datapoints.

One also has to consider the **computational cost** of fitting the model, updating the model, and making predictions using the model. The time complexity of model fitting of different SML methods ranges from linear up to cubic or slower. For some methods, updating the model with new data can require fitting a new model from scratch, while other methods provide much faster updates. Similarly, some methods could make a prediction in constant time, while others might require a sampling technique that results in much slower predictions. To select the appropriate method, one has to consider whether the model will be used online or offline, how often it will need to be updated, and whether the improvement in model accuracy is worth the increased computational cost.

Another property that is important in systems applications is **interpretability** of the model. For example, we created crisis fingerprints using a dimensionality reduction technique. Had we used PCA, a common technique for dimensionality reduction, the resulting features in the fingerprint would have been linear combinations of the original performance metrics. Such features would be very difficult for the datacenter operators to interpret. Instead, we used feature selection, which simply discards some of the metrics. This led to fingerprints based on the original performance metrics that are intuitive for the operators.

There are many other properties of statistical models that have to be considered when selecting a particular SML method. For example, is the model sensitive to **outliers** in the data? Does the model accept **non-numeric features**, such as text or graphs? How easy is it to incorporate **prior information** about the data into the model, such as monotonicity? The success of applying SML to systems depends on understanding these trade-offs and their implications on the original systems problem.

Consider SML as one component of the overall system

It is important to keep in mind that SML is usually only one part of the solution of the original problem. For example, while the accuracy of the Director performance model is important, the actual measures of success are the number of SLO violations and the server cost savings achieved by using the Director. Based on our own experience, as described in step 2 above, it is very easy to spend too much time improving the accuracy of the model and ignoring the actual systems problem.

Therefore, our suggestion is to first use the simplest possible SML method, incorporate the resulting model into the rest of the solution and then compute the actual measure of success. If the system is performing poorly, it is important to make sure that it is in fact the SML model that is causing the poor performance and not some other component in the system. Only after we identify the model as the culprit, should we start working on improving the accuracy of the model.

Because the measure of success of the original problem depends on the accuracy of the model, it is important to understand the best possible case. This can be achieved by replacing the model trained on data with an always-accurate model – an oracle – and running the rest of the evaluation as usual. For example, in the fingerprinting project, we constructed fingerprints based on perfect future knowledge of the performance metrics. Such experiments estimate the best possible case and can be used to decide whether it is worth tuning the model further.

6.2.2 Trusting SML in Datacenter Operations

In the computer systems community, SML is still sometimes considered too empirical and “black magic” and there are concerns that the datacenter operators might not trust tools based on SML. Here we highlight three reasons for this attitude and later outline how they could be addressed.

First, datacenter operators have a lot of expertise in management of applications, servers, and other datacenter components which they gained over the years. To them, it is difficult to understand that a tool based on SML could do a better job of detecting and identifying performance problems, finding anomalies in the system, or estimating performance of the system for unseen workloads.

Second, for tools that automatically adjust the system configuration, a small error could actually turn into a disaster. For example, in Chapter 3 we demonstrated that the Director framework can change the resource allocation in response to changes in workload. However, a workload or performance anomaly, or a bug in the control policy logic, could result in the policy allocating an order of magnitude more or less resources to the application. Either of these outcomes would be disastrous for the datacenter.

Third, using SML methods often requires setting various parameters, also called "voodoo constants." For example, to define a fingerprint, we have to specify the number of metrics to use, hot and cold thresholds, and other parameters. While there exist methods that also automatically adjust the values of these parameters, they are more difficult to use and more computationally intensive.

These three issues could be addressed in various ways. First, when making a prediction, the model should return both the prediction and the error bars on the prediction which the operators could use to adjust their response. For example, our crisis identification tool should not return only the most similar crisis, but maybe the top three crises along with a measure of certainty (see [Woodard and Goldszmidt, 2009]), which would help the operators select the best repair actions.

Another approach would be to provide an intuitive visualization of the model to help the operators understand what the model represents. We have used this technique in two projects: in crisis detection tool [Bodík *et al.*, 2005], where the visualization provided insight into which parts of the website were anomalous, and in crisis identification (see Chapter 4), where a visualization of the fingerprints allowed the operators to understand which metrics are represented in the fingerprint. In both cases, the visualization helped the operators better understand how the model works and consequently they trusted the tool more.

Yet another technique would be to build a framework that would allow us to replay historic monitoring data against an SML-based tool and compare its behavior to actions of the operators. An SML tool uses various performance metrics or application logs as inputs, builds a model based on the inputs, and then makes predictions about performance, workload, or failures. These metrics should be captured over long periods of time, especially during periods with unexpected workloads or updates of the application code. These historic metrics could later be replayed against the tool to verify its predictions. Such replay tools could also include support for sensitivity analysis, which would test the tools for different values of the tuning parameters.

6.2.3 Pervasive Monitoring to Improve System Visibility

Over the years, system designers learned how to design efficient datacenters and scalable Web application, but system monitoring is often an afterthought. We and others [Barroso and Hölzle, 2009] believe that monitoring and request tracing should become first-class citizens in datacenters. Standardized and pervasive monitoring would improve system visibility and also simplify the application of statistical techniques to datacenter management problems.

We propose the following three principles of pervasive monitoring in a datacenter:

1. Each software service should be monitored in detail and performance metrics should be reported through a standardized interface. Such monitoring should include incoming and outgoing requests (including their parameters), utilization of resources on hardware running the service, and performance metrics, such as latency and throughput.

2. The execution of each user request should be traced through all components of the system using frameworks such as X-Trace [Fonseca *et al.*, 2007]. Such request tracing

would allow the designers and operators to understand the logical dependencies among services, time spent in each service, and how workload on one service translates into workload and resource utilization on other services. Such instrumentation should be implemented not only at the level of individual services, but also at the level of tiers of each service (such as Web servers, application servers, and databases) as well as individual operations on each tier (for example, monitoring how SCADS [Armbrust *et al.*, 2009b] queries decompose into low-level get and put operations).

3. Because such detailed monitoring would have negative impact on performance of the system and require large amounts of storage, the operators need the ability to dynamically adjust the granularity of monitoring. For example, they need the option to reduce the frequency of data collection and to completely turn off monitoring of very frequent events.

Deploying standardized monitoring tools across all the applications, and software and hardware infrastructure of the datacenter, would simplify the development, testing and deployment of SML-based tools. Here we give examples of how such pervasive monitoring would impact the three problems described in this thesis.

In Chapter 2, we analyzed workload at the level of individual user requests or "clicks," as captured by the Web servers. Using request traces across the whole system stack would enable us to analyze how the original requests translated into requests to other services. For example, depending on the implementation and caching strategy, the same spike at the level of user requests might manifest differently on the different services in a datacenter. In some cases it might be simply absorbed by the cache, in others it might cause a significant data hotspot.

Monitoring requests across services would also simplify building performance models and resource allocation frameworks. Mainly, it would allow the decomposition of performance models of complex applications into simpler performance models of the individual services. Using such decomposition, we could estimate the impact of resource allocation in one of the services on end-to-end request latency.

Standardizing the monitoring across all services would simplify the deployment of crisis detection and identification tools like our fingerprinting method. Understanding the logical dependencies between services might help narrow down the possible causes of the problem and enable tracking of how crises spread among the different services.

Finally, such pervasive monitoring would simplify other datacenter-related problems, such as capacity planning. Using a dependency graph for the whole datacenter, it would be easy to estimate how a 20% increase in usage of a particular application translates into increase in network traffic, CPU utilization, or request latency.

6.3 Concluding Remarks

Statistical Machine Learning is a well understood field, but it has only recently started to be applied to problems in computer systems. Many systems experts are still skeptical about the benefits that SML could bring to debugging, analyzing, and controlling systems.

In this dissertation, we have presented three applications of SML to important problems in datacenter operations. We hope that we have demonstrated that SML techniques can be useful and that we have convinced some of the skeptics along the way. We believe that SML should become part of system designers vocabulary, just like basic analysis of algorithm complexity or queueing theory.

Bibliography

[Aldous, 1985] D. Aldous. Exchangeability and related topics. In *Ecole d'Ete de Probabilities de Saint-Flour XIII 1983*. Springer, 1985.

[Allspaw, 2008] John Allspaw. *The Art of Capacity Planning: Scaling Web Resources*. O'Reilly Media, Inc., 2008.

[Amazon.com, 2010] Amazon.com. Amazon Web Services. `http://aws.amazon.com`, 2010.

[Amur *et al.*, 2010] Hrishikesh Amur, James Cipar, Varun Gupta, Gregory R. Ganger, Michael A. Kozuch, and Karsten Schwan. Robust and flexible power-proportional storage. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, New York, NY, USA, 2010. ACM.

[Apache, 2010a] Apache. Cassandra. `http://incubator.apache.org/cassandra/`, 2010.

[Apache, 2010b] Apache. HBase. `http://hadoop.apache.org/hbase/`, 2010.

[Apache, 2010c] Apache. JMeter project web site. `http://jakarta.apache.org/jmeter/`, 2010.

[Arlitt and Jin, 1999] Martin Arlitt and Tai Jin. Workload characterization of the 1998 World Cup Web site. Technical Report HPL-1999-35R1, HP Labs, 1999.

[Armbrust *et al.*, 2009a] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A Berkeley view of Cloud Computing. Technical Report UCB/EECS-2009-28, UC Berkeley, 2009.

[Armbrust *et al.*, 2009b] Michael Armbrust, Armando Fox, David Patterson, Nick Lanham, Haruki Oh, Beth Trushkowsky, and Jesse Trutna. SCADS: Scale-independent storage for social computing applications. In *Conference on Innovative Data Systems Research (CIDR)*, 2009.

[Balakrishnan *et al.*, 2003] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems. *Commun. ACM*, 46(2):43–48, 2003.

[Barford and Crovella, 1998] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *SIGMETRICS*, 1998.

[Barham *et al.*, 2004] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, Berkeley, CA, USA, 2004. USENIX Association.

[Barroso and Hölzle, 2009] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[Bennani and Menasce, 2005] Mohamed N. Bennani and Daniel A. Menasce. Resource allocation for autonomic data centers using analytic performance models. In *ICAC*, 2005.

[Bodík *et al.*, 2005] Peter Bodík, Greg Friedman, Lukas Biewald, Helen Levine, George Candea, Kayur Patel, Gilman Tolle, Jon Hui, Armando Fox, Michael I. Jordan, and David Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *International Conference on Autonomic Computing (ICAC)*, 2005.

[Bodík *et al.*, 2006] Peter Bodík, Armando Fox, Michael I. Jordan, David A. Patterson, Ajit Banerjee, Ramesh Jagannathan, Tina Su, Shivaraj Tenginakai, Ben Turner, and Jon Ingalls. Advanced tools for operators at amazon.com. In *Hot Topics in Autonomic Computing (HotAC)*, 2006.

[Bodík *et al.*, 2009a] Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael I. Jordan, and David A. Patterson. Automatic exploration of datacenter performance regimes. In *ACDC '09: Proceedings of the 1st workshop on Automated control for datacenters and clouds*, New York, NY, USA, 2009. ACM.

[Bodík *et al.*, 2009b] Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael I. Jordan, and David A. Patterson. Statistical Machine Learning makes automatic control practical for Internet datacenters. In *Workshop on Hot Topics in Cloud Computing (HotCloud '09)*, 2009.

[Bodík *et al.*, 2010] Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Symposium on Cloud Computing (SOCC)*, 2010.

[Bodík *et al.*, 2008] Peter Bodík, Moisés Goldszmidt, and Armando Fox. Hilighter: Automatically building robust signatures of performance behavior for small- and large-scale systems. In Armando Fox and Sumit Basu, editors, *Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML)*. USENIX Association, 2008.

[Borthakur, 2007] Dhruba Borthakur. The Hadoop Distributed File System: Architecture and design, 2007.

[Chase *et al.*, 2001] Jeff Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *Symposium on Operating Systems Principles (SOSP)*, 2001.

[Chen and Zhang, 2003] Xin Chen and Xiaodong Zhang. A popularity-based prediction model for web prefetching. *IEEE Computer*, 2003.

[Chen *et al.*, 2004] Mike Y. Chen, Emre Kıcıman, Anthony Accardi, Eric A. Brewer, David Patterson, and Armando Fox. Path-based failure and evolution management. In *Proc. 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI'04)*, San Francisco, CA, 2004.

[Chen *et al.*, 2006] Jin Chen, Gokul Soundararajan, and Cristiana Amza. Autonomic provisioning of backend databases in dynamic content web servers. In *IEEE International Conference on Autonomic Computing (ICAC)*, 2006.

[CNN, 2009] CNN. World watches odyssey of 'balloon boy' in real time. http://www.cnn.com/2009/US/10/15/colorado.boy.world.watching, 2009.

[Cohen *et al.*, 2004] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2004)*, San Francisco, CA, 2004.

[Cohen *et al.*, 2005] Ira Cohen, Steve Zhang, Moisés Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving system history. In Andrew Herbert and Kenneth P. Birman, editors, *Symposium on Operating Systems Principles (SOSP)*. ACM, 2005.

[Cook *et al.*, 2007] Brian Cook, Shivnath Babu, George Candea, and Songyun Duan. Toward Self-Healing Multitier Services. In *Proceedings of the 2nd International Workshop on Self-Managing Database Systems (joint with ICDE), 2007*, 2007.

[Cooper *et al.*, 2008] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[Cooper *et al.*, 2010] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. *Symposium on Cloud computing (SOCC)*, 2010.

[Crovella and Bestavros, 1996] Mark E. Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic evidence and possible causes. *IEEE/ACM Transactions on Networking*, 1996.

[Dean and Ghemawat, 2004] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementations (OSDI)*, 2004.

[Dean, 2009] Jeffrey Dean. Large-scale distributed systems at Google: Current systems and future directions, 2009. Keynote presentation at LADIS 2009.

[Dean, 2010] Jeffrey Dean. Evolution and future directions of large-scale storage and computation systems at Google, 2010. Keynote presentation at SOCC 2010.

[DeCandia *et al.*, 2007] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Symposium on Operating Systems Principles (SOSP)*, 2007.

[Duan and Babu, 2008] Songyun Duan and Shivnath Babu. Guided problem diagnosis through active learning. In *International Conference on Autonomic Computing (ICAC) 2008*, Washington, DC, USA, 2008. IEEE Computer Society.

[Faban, 2009] Faban. Faban project web site. http://faban.sunsource.net/, 2009.

[Fonseca *et al.*, 2007] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: A pervasive network tracing framework. In *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 2007.

[Ganapathi, 2009] Archana Sulochana Ganapathi. *Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning*. PhD thesis, EECS Department, University of California, Berkeley, 2009.

[Glerum *et al.*, 2009] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Symposium on Operating Systems Principles (SOSP)*, Big Sky, Montana, 2009.

[Goldszmidt *et al.*, 2005] Moises Goldszmidt, Ira Cohen, Steve Zhang, and Armando Fox. Three research challenges at the intersection of machine learning, statistical inference, and systems. In *Tenth Workshop on Hot Topics in Operating Systems (HotOS-X)*, Santa Fe, NM, 2005.

[Google, 2009] Google. Outpouring of searches for the late Michael Jackson. http://googleblog.blogspot.com/2009/06/outpouring-of-searches-for-late-michael.html, 2009.

[Guha and McGregor, 2009] Sudipto Guha and Andrew McGregor. Stream order and order statistics: Quantile estimation in random-order streams. *SIAM Journal on Computing*, 38(5):2044–2059, 2009.

[Gulati *et al.*, 2009] Ajay Gulati, Chethan Kumar, and Irfan Ahmad. Storage workload characterization and consolidation in virtualized environments. In *Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*, 2009.

[Hellerstein *et al.*, 2008] Joseph L. Hellerstein, Vance Morrison, and Eric Eilebrecht. Optimizing concurrency levels in the .net threadpool: A case study of controller design and implementation. In *Feedback Control Implementation and Design in Computing Systems and Networks*, 2008.

[HP, 2010] HP. HP OpenView. `http://welcome.hp.com/country/us/en/prodserv/software.html`, 2010.

[Httperf, 2009] Httperf. Httperf project web site. `http://code.google.com/p/httperf/`, 2009.

[Isard *et al.*, 2009] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Symposium on Operating Systems Principles (SOSP)*, 2009.

[Jung *et al.*, 2002] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for CDNs and Web sites. In *International World Wide Web Conference (WWW)*, 2002.

[Kavalanekar *et al.*, 2008] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of storage workload traces from production windows servers. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[Koh *et al.*, 2007] Kwangmoo Koh, Seung-Jean Kim, and Stephen Boyd. An interior-point method for large-scale L1-regularized logistic regression. *Journal of Machine Learning Research*, 8:1519–1555, 2007.

[Kusic *et al.*, 2008] Dara Kusic, Jeffrey O. Kephart, James E. Hanson, Nagarajan Kandasamy, and Guofei Jiang. Power and performance management of virtualized computing environments via lookahead control. In *ICAC '08: Proceedings of the 2008 International Conference on Autonomic Computing*, Washington, DC, USA, 2008. IEEE Computer Society.

[Lachiche and Flach, 2003] N. Lachiche and P. Flach. Improving accuracy and cost of two-class and multi-class probabilistic classifiers using ROC curves. In *International Conference on Machine Learning (ICML)*, 2003.

[Lakhina *et al.*, 2004] Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing network-wide traffic anomalies. In *SIGCOMM*, 2004.

[LeFebvre, 2001] William LeFebvre. CNN.com: Facing a world crisis. `http://www.tcsa.org/lisa2001/cnn.txt`, 2001.

[Liu *et al.*, 2006] Xue Liu, Jin Heo, Lui Sha, and Xiaoyun Zhu. Adaptive control of multi-tiered web applications using queueing predictor. *Network Operations and Management Symposium (NOMS)*, 2006.

[Massie, 2004] M. Massie. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7), 2004.

[Mi *et al.*, 2009] Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. Injecting realistic burstiness to a traditional client-server benchmark. In *International Conference on Autonomic Computing (ICAC)*, 2009.

[Narayanan *et al.*, 2008] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony I. T. Rowstron. Everest: Scaling down peak loads through I/O off-loading. In Richard Draves and Robbert van Renesse, editors, *Symposium on Operating Systems Design and Implementations (OSDI)*. USENIX Association, 2008.

[Padmanabhan and Qiu, 2000] Venkata N. Padmanabhan and Lili Qiu. The content and access dynamics of a busy web server: Findings and implications. In *SIGCOMM*, 2000.

[Patterson *et al.*, 2002] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emere Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupamn, and Noah Treuhaft. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical report, UC Berkeley, 2002.

[Pertet *et al.*, 2007] Soila Pertet, Rajeev Gandhi, and Priya Narasimhan. Fingerpointing correlated failures in replicated systems. In *SYSML'07: Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques*, Berkeley, CA, USA, 2007. USENIX Association.

[Pitman and Yor, 1997] Jim Pitman and Marc Yor. The two-parameter Poisson-Dirichlet distribution derived from a stable subordinator. *Annals of Probability*, 25(2):855–900, 1997.

[Rabkin and Katz, 2010] Ariel Rabkin and Randy H. Katz. Chukwa: A system for reliable large-scale log collection. Master's thesis, EECS Department, University of California, Berkeley, 2010.

[Redstone *et al.*, 2003] Joshua A. Redstone, Michael M. Swift, and Brian N. Bershad. Using computers to diagnose computer problems. In *9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, Elmau, Germany, 2003.

[Reynolds *et al.*, 2005] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, Charles Killian, and Amin Vahdat. Experiences with Pip: finding unexpected behavior in distributed systems. In *Symposium on Operating Systems Principles (SOSP)*, New York, NY, USA, 2005. ACM.

[Rossiter, 2003] J. A. Rossiter. *Model based predictive control: a practical approach*. CRC Press, 2003.

[Rubis, 2008] Rubis. Rubis project web site. http://rubis.ow2.org/, 2008.

[Schroeder and Harchol-Balter, 2006] Bianca Schroeder and Mor Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Transactions Internet Technology*, 6(1):20–52, 2006.

[Schroeder *et al.*, 2006] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: a cautionary tale. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, Berkeley, CA, USA, 2006. USENIX Association.

[Shoup, 2007] Randy Shoup. eBay's architecture principles, 2007. QCON talk.

[Soundararajan *et al.*, 2006] Gokul Soundararajan, Cristiana Amza, and Ashvin Goel. Database replication policies for dynamic content applications. In *EuroSys*, number 4, 2006.

[Stewart and Shen, 2005] Christopher Stewart and Kai Shen. Performance modeling and system management for multi-component online services. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[Tesauro *et al.*, 2006] Gerald Tesauro, Nicholas Jong, Rajarshi Das, and Mohamed Bennani. A hybrid reinforcement learning aproach to autonomic resource allocation. In *International Conference on Autonomic Computing (ICAC)*, 2006.

[Thereska *et al.*, 2009] Eno Thereska, Austin Donnelly, and Dushyanth Narayanan. Sierra: a power-proportional, distributed storage system. Technical Report MSR-TR-2009-153, Microsoft, 2009.

[trendistic.com, 2010] trendistic.com. Top 20 Twitter trends in 2009. http://trendistic.com/_top-twenty-trending-topics-2009/, 2010.

[Twitter, 2009] Twitter. Inauguration Day on Twitter. http://blog.twitter.com/2009/01/inauguration-day-on-twitter.html, 2009.

[Urgaonkar *et al.*, 2005a] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. *SIGMETRICS*, 2005.

[Urgaonkar *et al.*, 2005b] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, and Pawan Goyal. Dynamic provisioning of multi-tier internet applications. In *International Conference on Autonomic Computing (ICAC)*, 2005.

[Wikipedia, 2007] Wikipedia. Wikipedia page counters. http://mituzas.lt/2007/12/10/wikipedia-page-counters/, 2007.

[Wolman *et al.*, 1999] Alec Wolman, M. Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry M. Levy. On the scale and performance of cooperative web proxy caching. In *Symposium on Operating Systems Principles (SOSP)*, 1999.

[Woodard and Goldszmidt, 2009] Dawn Woodard and Moises Goldszmidt. Model-based clustering for online crisis identification in distributed computing. Technical report, Microsoft Research, 2009.

[Xu *et al.*, 2009] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, New York, NY, USA, 2009. ACM.

[Yemini *et al.*, 1996] S. A. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High speed and robust event correlation. *Communications Magazine, IEEE*, 34(5):82–90, 1996.

[Young and Hastie, 2006] Mee Young and Park Trevor Hastie. L1 regularization-path algorithm for generalized linear models. *Journal of the Royal Statistical Society: Series B*, 2006.

[Yuan *et al.*, 2006] Chun Yuan, Ni Lao Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. Automated known problem diagnosis with event traces. In *EuroSys 2006*, Leuven, Belgium, 2006.

[Zaharia *et al.*, 2010] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In Christine Morin and Gilles Muller, editors, *EuroSys*. ACM, 2010.

[Zhang *et al.*, 2005] Steve Zhang, Ira Cohen, Moises Goldszmidt, Julie Symons, and Armando Fox. Ensembles of models for automated diagnosis of system performance problems. In *International Conference on Dependable Systems and Networks (DSN 2005)*, Yokohama, Japan, 2005.