Statistical Methods for Locating Performance Problems in Multi-Tier Applications

by

Michael R. Stunes

S.B., Massachusetts Institute of Technology (2011)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author .....
Department of Electrical Engineering and Computer Science
May 1, 2012

Certified by .....
Zhelong Pan
Staff Engineer, VMware
Thesis Supervisor

Certified by .....
Robert T. Morris
Professor
Thesis Supervisor

Accepted by .....
Prof. Dennis M. Freeman
Chairman, Master of Engineering Thesis Committee

Statistical Methods for Locating Performance Problems in Multi-Tier Applications

by

Michael R. Stunes

Submitted to the Department of Electrical Engineering and Computer Science
on May 1, 2012, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes an algorithm developed to aid in solving the problem of performance diagnosis, by automatically identifying the specific component in a multi-component application system responsible for a performance problem. The algorithm monitors the system, collecting load and latency information from each component, searches the data for patterns indicative of performance saturation using statistical methods, and uses a machine learning classifier to interpret those results. The algorithm was tested with two test applications in several configurations, with different performance problems synthetically introduced. The algorithm correctly located these problems as much as 90% of the time, indicating that this is a good approach to the problem of automatic performance problem location. Also, the experimentation demonstrated that the algorithm can locate performance problems in environments different from those for which it was designed and from that on which it was trained.

Thesis Supervisor: Zhelong Pan
Title: Staff Engineer, VMware

Thesis Supervisor: Robert T. Morris
Title: Professor

Acknowledgments

First and foremost, thank you to my parents, for providing me with a lifetime of opportunities, and for all of their love and support. Second, a special thanks goes to my mentor, Zhelong Pan, and advisor, Prof. Robert Morris, for their help and guidance during this project. Third, thank you to my colleagues Xiaoyun Zhu, Rean Griffith, and Pengcheng Xiong, for providing several of the ideas that aided in the development of this project.

I would like to acknowledge my colleagues Rajit Kambo and Linda Condon for their HR support during my time at VMware, and also thank you to my colleague Jeff Buell for sharing his expertise with Hadoop.

I humbly dedicate this work to my late uncle, Richard Carl Ritter, in recognition of his extraordinary achievements in education, and to my late grandfather, Richard Raymond Stunes, in honor of his dedication to his family.

Contents

1 Introduction 13

2 The Problem 15

2.1 Assumptions 17

2.2 Design Goals 18

2.3 Non-goals 19

3 Canonical Performance Characteristics 21

4 The Solution 27

4.1 Data Collection 27

4.2 Statistical Analysis 28

4.3 Aggregation 29

4.4 Machine Learning 31

4.5 CPU-Ready Enhancement 33

4.6 Implementation 33

5 Experimentation 35

5.1 Olio 35

5.2 Hadoop 38

5.3 Machine Learning 39

5.3.1 Evaluation 40

6 Results 43

6.1 Olio 43
6.2 Hadoop 47

7 Conclusion **49**

A Performance Metrics Collected **51**
A.1 Guest-Level Metrics 51
A.2 VM-Level Metrics 52
A.3 Host Metrics 52

B Datasets **53**

List of Figures

3-1 Canonical performance characteristics 23

3-2 Relationship between latency and CPU utilization 25

4-1 Decision tree for classifying performance metrics 30

5-1 Experimental setup for Olio Rails. 36

5-2 Experimental setup for Olio PHP. 36

6-1 Relationship between latency and user CPU utilization for the application bottleneck test 45

6-2 Relationship between latency and user CPU utilization for the database bottleneck test 46

List of Tables

5.1 Testbed variable parameter configurations. 38

6.1 Metric classification results from application bottleneck test 43
6.2 Metric classification results from database bottleneck test 44
6.3 Machine learning accuracy, separate training and testing datasets . . 46
6.4 Machine learning accuracy, with and without CPU-Ready attributes. 47
6.5 Machine learning results for Hadoop tests. 48

B.1 Olio Rails Dataset 55
B.2 Olio PHP Dataset 56
B.3 Hadoop Dataset 57

Chapter 1

Introduction

Diagnosing a performance problem is frequently difficult. Large software systems are continually growing in complexity, and frequently are built out of many interacting components. Interactions between components may disguise the actual source of a performance problem, or introduce more opportunities for problems to arise, and increase the difficulty of diagnosis. [7, 16] Furthermore, deploying such a system on a virtualized infrastructure adds a further source of complexity and interactions that may cause such problems. [9]

Quick diagnosis of performance problems is necessary because these problems can cause attrition of an application's user base and affect the application's revenue stream. Users who visit a Web site and experience poor performance may not return because of their experience, causing a permanent loss of potential revenue. [8]

This document describes an algorithm developed to aid in solving the problem of performance diagnosis by automatically identifying the specific component in a multi-component application system responsible for a performance problem. The algorithm monitors each component separately, collecting load vs. latency data and performance metrics, examining the latency of each component in the system as a function of its load. It then searches graphs of these data for inflection points, which are indicative of performance saturation, using statistical regressions. Finally, a machine learning classifier examines the results of the regressions, and decides for each component whether that component is exhibiting performance characteristics

typical of a machine operating beyond its capacity, and determines if it is the source of the system's performance problem.

The statistical methods used in this work were chosen for their close fit to universal models of application performance, described further in Chapter 3. Machine learning was used here as a convenient method for making a yes-or-no decision based on the outcome of those statistical analyses; the problem here is a natural fit for machine learning classification, and there are accepted methods of assessing the performance of a machine learning system. However, this is not to suggest that a relatively sophisticated machine learning system is the only valid approach here; there are certainly other equally valid and effective methods of solution for this problem.

This algorithm relies on universal performance characteristics common to a wide range of applications, discussed further in Chapter 3. These characteristics can be straightforwardly detected with common statistical methods; their universality makes the resulting algorithm useful for many different applications, even beyond those for which it was specifically designed.

One difficulty, however, is that individual components in such multi-component systems are frequently highly dependent on one another. As a result, individual components show quite similar performance characteristics, and isolating one component as the source of a bottleneck can be difficult. This algorithm solves that problem by breaking down the system's primary performance indication (the latency of a request made to the application) into parts, each of which is attributed to one particular component, and carrying out performance analysis on these partial latencies from each component.

Experiments were conducted to assess the performance of the algorithm on several test applications with performance bottlenecks deliberately and synthetically introduced. The algorithm performed well, with accuracies exceeding 90% in some cases. Also, the experimentation demonstrated that the algorithm can locate performance bottlenecks in environments different from that for which it was designed and on which it was originally trained.

Chapter 2

The Problem

The algorithm developed herein is targeted for specific types of performance problems, with particular assumptions made about the system under test. More specifically, the algorithm will locate the one server in a multi-server system that is first reaching the maximum rate at which it can respond to requests from users, and is therefore the rate-limiting step in the system. This algorithm is looking specifically for such performance bottlenecks caused by resource starvation (where one resource, e.g., CPU or I/O bandwidth) has reached the effective limit of what is available to be used, or background noise in a virtualized environment (where one virtual machine is unable to operate at its maximum capacity because of resource consumption by other virtual machines on the same host). In addition, this algorithm is targeted specifically for stable (as opposed to dynamic) performance problems, where a particular performance problem is an inherent property of the system as configured, and not caused by some transient state. However, it is possible that such a problem would not be triggered until the load offered to the system reaches a particular point; thus, such "stable" problems may appear and disappear with the ebb and flow of the system's load.

The algorithm focuses specifically on applications using a multi-tier architecture, in which the application is separated into different layers, or *tiers*, each handling one particular functional concern. A typical design for Web applications is a *three-tier* architecture, frequently comprising a user-interface or load-balancing tier, an

application tier responsible for the application's business logic, and a database tier for handling the system's persistent data. Each tier may be hosted on one or several servers.

Furthermore, this algorithm focuses on applications that use a request/response model for interaction with the user. In this model, a user issues a *request* for a particular item of content (e.g., the application's home page); the application then replies with a *response*, namely, the desired content.

An example will serve to both illustrate the problem and demonstrate some difficulties in diagnosis. The Thin web server [5], commonly used for serving Web applications written in the Ruby language, is single-threaded; its internal architecture renders it incapable of serving multiple requests simultaneously. While waiting for external I/O (e.g., making a request to the application's underlying database), Thin's server thread sleeps, which the OS kernel reports as idle CPU time. However, the application is unable to utilize that CPU time, because the Thin server is sleeping and not responding to incoming requests. The result is that the Thin server can saturate, reaching its maximum capacity for handling incoming requests, at a level of CPU utilization (as low as 50% in some tests, depending on configuration) that would suggest normal operation. This algorithm is able to detect this problem by searching for patterns caused by performance properties of this system: once the CPU utilization reaches its maximum (regardless of its exact value; 50% in this example), the latency imposed on a request by the Thin server will continue growing as the system's load increases, as the CPU utilization remains at its maximum, without growing further. However, the latency imposed on a request by other machines in the system (such as the database server in this example) will not continue growing as the load on the system increases; once the Thin server reaches its maximum capacity, it will stop offering additional load to the database server, and the latency imposed by the database server will stop growing. The algorithm detects that the latency imposed by the Thin server continues growing where the latency imposed by the database server does not; this is fundamentally how it detects the Thin server as the system's bottleneck.

One property of multi-tier applications that increases the difficulty of locating such problems is an interdependence between resource usages of different tiers, because each tier is receiving the same offered load. As an example, if the application server tier saturates and its utilization levels off, it stops offering more load to the database tier, so the database utilization levels off as well. As another example, if the database tier saturates, it limits the application server's throughput, making it appear to level off as well.

Because of this property, looking for relationships between performance metrics of each tier and the latency or throughput of the whole system is not particularly useful for finding performance bottlenecks: all of the tiers are behaving similarly performance-wise with respect to the system's offered load. However, it is possible to consider the latency imposed on a request by each tier separately. For example, in a typical three-tier application, we could measure the latency added by the application tier separately from that added by the database tier (i.e., measure the time a single request spends processing in the application tier, and then separately measure how much time is spent processing in the database tier). Using these per-tier latencies is one central part of this algorithm: if one particular server is not yet operating at its maximum capacity, the latency imposed by that server will not increase as the load on the system increases; likewise, if a particular server is operating at its maximum capacity, the latency imposed by that server will grow as the load on the system increases.

2.1 Assumptions

This algorithm makes several assumptions about the system under test. First, the algorithm focuses on applications that are built in an RPC-based architecture [7], where the individual components communicate using a call-and-response model. Users make requests to the application's outermost tier, which in turn may make requests to other tiers, which in turn issue responses. In this model, each action carried out by one tier, and each request and response between tiers, is caused by a request to the application

from a client. For each request, we can measure the total amount of time until a client receives a response (the *latency* of that request); in addition, in the RPC model, we can measure how much of the total latency is caused by each tier or each server in the application.

Second, the algorithm focuses on applications being operated as closed-queue systems. These applications can be modeled as queueing systems, where clients waiting to be served enter a queue (such as the server's operating system TCP queue), and the server takes requests from the queue one at a time and processes them, generating a response.

A closed-queue system is one in which "the total number of [users] is finite and fixed," or where users return to the same queue after having been served. [18] In such a system, at any point in time every user is either "thinking" (waiting to join the queue), or waiting in the queue to be serviced. Closed-queue systems are particularly relevant in performance testing and benchmarking; a typical performance benchmark will utilize a certain number of active workload generators, which are the "users" in a closed-queue system. Each generator is always either waiting to have a request serviced, or delaying for some "think time" (simulating a user waiting between requests). A website or other service with a certain number of active users at any given time can also be modeled in this way.

2.2 Design Goals

Based on these targeted performance problems, several design goals shaped the design of the algorithm. First, because of the behaviors demonstrated by the Thin server example above, where its CPU utilization reaches its effective maximum utilization well below the actual maximum of 100%, the algorithm should not rely on any hard-coded thresholds for particular performance metrics, as those can be misleading. A rule such as "alarm if CPU utilization greater than 95%" would fail to detect the Thin server above as the true source of a performance problem. Instead, the algorithm should rely on behavioral patters that are indicative of performance bottlenecks,

without using specific hard-coded cutoffs.

Second, the algorithm should make minimal assumptions about the application under test beyond what has been assumed above, in order to remain as broadly applicable as possible.

2.3 Non-goals

One step in the resulting algorithm creates statistical models that could be used to mathematically describe or predict the application's performance. However, describing or predicting the application's performance is not the end goal; rather, the end goal is specifically to detect which server in a multi-tier, multi-server system is causing a performance bottleneck.

Chapter 3

Canonical Performance Characteristics

Chapter 2 discussed particular performance characteristics, and indicated that patterns of these characteristics can be used to detect performance problems, such as the pathological behavior of the Thin application server. This chapter will discuss those performance characteristics and patterns in more detail, and describe how they can be used to locate such performance problems.

Section 2.1 described closed-queue systems as the focus of this work. In a closed-queue system, the throughput (rate at which requests are served by the system; the number of requests served in a given time interval) can be expressed as

$$X(N) = \frac{N - Q}{Z}$$

where N is the total number of users in the system, X is the throughput as a function of the number of users, Q is the number of users being serviced or waiting in the queue, and Z is the average think time. The throughput is simply the rate at which users are leaving the "thinking" state; $N - Q$ is the number of users who are not being serviced or in the queue, and therefore is the number of users in the "thinking" state. [18]

Section 2.1 described closed-queue systems as the focus of this work. Such sys-

tems exhibit particular relationships between offered load (the rate at which users are issuing requests) and throughput and latency (the rate at which requests are served by the system and the time required for a user to receive a response after submitting a request, respectively), known by [17] as the "canonical performance characteristics." Figure 3-1(a) illustrates the canonical relationship between offered load and throughput, with offered load (as the number of active users) on the X-axis and the throughput (in requests handled per unit of time) on the Y-axis. (Here, the exact units and quantities are not important; this is an illustration of a general phenomenon shared by many different systems.) This relationship is characterized by two linear regions, which describe the behavior of the system below and above its saturation point, which is the maximum rate of incoming requests that it is capable of processing. In a closed-queue system, the throughput will increase linearly with the number of active users, until the application reaches its saturation point. At that point, the throughput has reached a maximum, and stops growing.

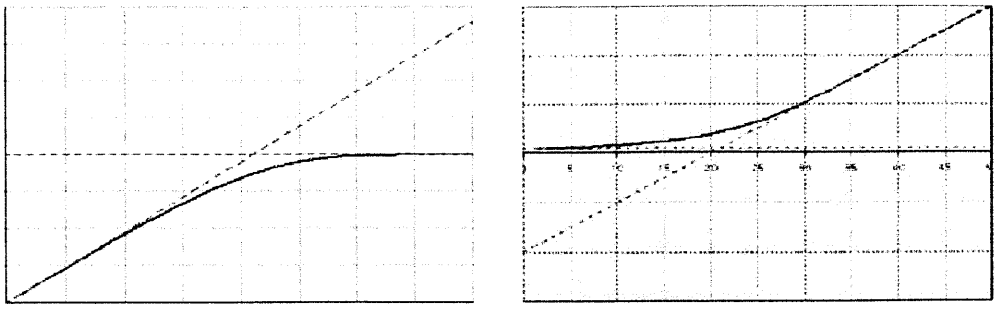
Figure 3-1(b) shows the canonical relationship between offered load and latency, with offered load on the X-axis and latency (the average time between a user making a request and receiving a response) on the Y-axis. Like the canonical throughput relationship, this is also characterized by two linear regions in a closed-queue system. Below the saturation point, the latency remains constant with respect to the number of active users, as there is a minimum for any one request dictated by the system. Above the saturation point, the average latency will grow linearly with the number of active users.

[18] explains why latency grows linearly with increasing load in a closed-queue system. We know from before that:

$$X(N) = \frac{N - Q}{Z}$$

The latency is related to the throughput by Little's law:

$$Q = XR$$



(a) Throughput as a function of offered load. (b) Latency as a function of offered load.

Figure 3-1: Canonical relationship between offered load (X-axis) and performance characteristics (Y-axis). Reproduced from [17]. Solid lines indicate canonical relationships; dashed lines indicate linear regions that characterize those relationships.

where R is the latency.

Substituting and rearranging to find R gives:

$$R = \frac{N}{X(N)} - Z$$

Z (think time) is constant; also, above the saturation point, $X(N)$ (throughput) is also constant. Thus, above the saturation point, latency grows linearly with the number of active users.

The key feature of these canonical performance characteristics is that regions of the curves can be described by either a linear relationship, or by a segmented relationship comprising two linear regions. These types of relationships can be straightforwardly detected by simple statistical methods. Using these methods to detect these relationships is an important component of the work presented in Chapter 4. This work relies primarily on determining whether a given system component is exhibiting linear relationships in its performance characteristic (which could indicate that it is operating only below its saturation point, is not operating at its maximum capacity, and therefore is not the source of a performance problem), or is exhibiting segmented relationships like the canonical characteristics, indicating that at some times, it is operating above its maximum capacity and is therefore causing a performance bottleneck.

Finally, the relationship between latency and a performance metric, such as CPU utilization, follows much the same pattern as the canonical characteristics above. See 3-2 for an example latency vs. CPU characteristic for a server known to be operating both below and above its saturation point; note that its shape is similar to the shape of the canonical characteristics, comprising two roughly-linear segments. (This graph shows data from the Thin server example discussed in Chapter 2, exhibiting the problem where it reaches its maximum capacity at low levels of CPU utilization.) Below saturation, a server's CPU utilization (or other performance metric) will grow with increasing load, as more resources are consumed. At the same time, the latency will remain roughly constant, as the server is not yet saturated. Once the server reaches saturation, the performance metric will stop increasing, as the server has reached its maximum utilization of that resource; at this point, the latency will also begin growing with increasing load. The result is that the relationship between latency and a performance metric will also show the same segmented shape, and it is possible to use the same analyses mentioned above.

This fact is also particularly useful because in typical applications, monitoring these performance metrics is easy, whereas directly measuring the offered load in terms of the number of currently active users can be more difficult. However, if it is possible to directly measure the number of active users at any point in time, that information could be used to construct a canonical performance characteristic, which could be used in this algorithm either alongside or instead of the latency vs. metric characteristics. Also, when using performance metric characteristics, it is of note that many different performance metrics should be used (a full list of those used during experimentation appears in Appendix A). Any one performance metric may or may not be related to the performance of a particular server (e.g., a disk bandwidth metric will be of little use on a server hosting a CPU-bound workload). Furthermore, aggregating the analysis over many metrics makes the algorithm more robust to statistical noise; due to the properties of the statistical analyses used, a performance characteristic for one metric may be spuriously detected as showing a segmented relationship, due to noise in the data, when in fact the corresponding

server has not reached its saturation point.

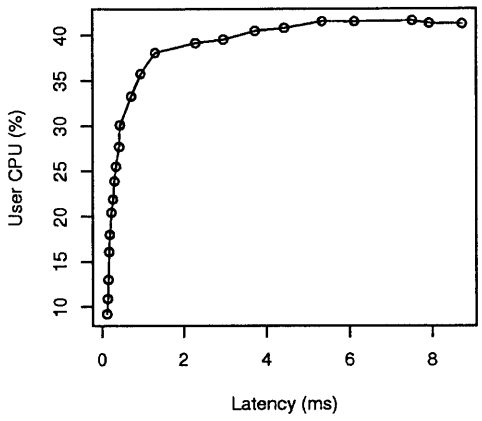


Figure 3-2: Relationship between latency and CPU utilization

Chapter 4

The Solution

This chapter describes the algorithm used for locating the server that has first hit its maximum capacity. The general procedure is to first collect performance data from each machine, and use the data to create graphs of each server's performance characteristic, like those described at the end of Chapter 3. Then, those characteristics are classified based on whether they show a segmented relationship characteristic of a server having reached its saturation point, or a linear relationship indicating that the server may not be at saturation. Finally, a machine learning classifier examines the results of the classification, and indicates, for each server, whether it is reaching its maximum capacity and is thus the system's performance bottleneck.

4.1 Data Collection

The algorithm begins by collecting performance data from the system under test, including performance metrics (e.g., CPU or I/O utilization) and performance responses (latency and throughput) from each server in the application individually, at periodic intervals (on the order of 1 to 10 seconds). The latency measurement for each individual tier is the part of the latency for a request for which that tier is responsible, including any queueing delay imposed on requests arriving at that tier. On a production system, collecting data over a relatively long period of time would give a range of load values (both above and below the system's point of saturation)

because of natural variations in load. In a test environment, such a variation in load can be deliberately introduced.

These performance data are noisy, so the data are smoothed out by aggregating over longer time windows (on the order of 1 minute), using arithmetic mean for performance metrics and 90th percentile for latency.

4.2 Statistical Analysis

Next, the algorithm searches the collected performance data for patterns like the canonical performance characteristics. Recall from Chapter 3 that a server exhibiting a linear performance characteristic may not have reached its saturation point, but a server exhibiting a segmented characteristic is operating past its saturation point at least some of the time. For each server in the system, the algorithm must determine which of those characteristics that server is exhibiting (or neither at all). The algorithm uses statistical analyses on the data for each individual metric on each server in order to do this. Statistical regressions are performed between the aggregated values of each metric on each server and the aggregated latency for that server.

Linear characteristics can be detected with a simple linear regression, which attempts to express a linear relationship between two variables. Formally, suppose that two variables X and Y are believed to be related by

$$Y = \beta_0 + \beta_1 X + \epsilon$$

where ϵ is a normally-distributed error term. A simple linear regression estimates values for β_0 and β_1, and also computes a value known as the *coefficient of determination* (denoted R^2), which provides a measure of the goodness of fit of the estimated relationship. R^2 takes values between 0 and 1, inclusive; values near 1 indicate a very good fit, whereas values near 0 indicate a poor fit. [14]

Segmented characteristics can be detected with a segmented regression. Segmented regression is similar to simple linear regression, except its goal is to detect

broken-line relationships, consisting of two or more linear regions connected at unknown breakpoints along the X-axis. A statistical procedure known as Davies' test is used to check for the presence of such a breakpoint, and thus determine whether a segmented model is a good fit for the data. Davies' test chooses a number of fixed breakpoints along the X-axis, performs a segmented regression for each, and checks for a statistically-significant difference in regression slopes (the β_1 parameter in the model above) on each side of the breakpoint. The result is a p-value, which can be interpreted as the probability that any difference in slopes is due to random chance. A p-value close to 1 indicates that any such difference is likely due to chance; a p-value close to 0 indicates that the difference is likely caused by an actual breakpoint in the data, and that a segmented model is probably a good fit. [11, 20, 21]

Based on the results of each of the two regression techniques, each metric on each server is classified into one of three categories. Figure 4-1 illustrates the classification procedure. The categories are as follows:

Linear A simple linear regression gave $R^2 > 0.7$, indicating that a linear model is a good fit for this metric. This most likely indicates that this metric falls into one linear region of a canonical performance characteristic, and that the corresponding server may not be operating past its saturation point.

Segmented Davies' test gave $p < 0.05$, indicating that a segmented model is a good fit for this metric. This likely indicates that this metric follows the canonical performance characteristic, and the corresponding server is reaching its saturation point.

None Neither a linear model nor a segmented model adequately explains this metric.

4.3 Aggregation

After all of the performance metrics have been classified, metrics that fall into the same category for every server are discarded. The reason is that, knowing that one

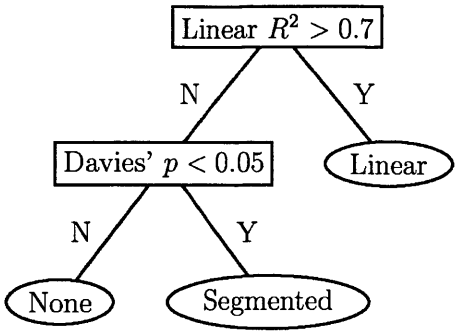


Figure 4-1: Decision tree for classifying performance metrics based on regression parameters.

tier/server is responsible for a performance bottleneck, the algorithm should look for properties unique to each server, and not properties that they share.

Then, for each server, the algorithm counts the number of metrics falling into each of the three categories. This is done to add statistical robustness to the algorithm, and to give a more broad overview of the server's performance characteristic. If a server is reaching its saturation point, we expect that many of its metrics will exhibit a segmented relationship with its latency. It may seem counterintuitive at first glance that many metrics should show such a relationship (e.g., if a server is bottlenecked on CPU, why would its disk metrics show a segmented relationship indicative of a bottleneck?); however, each metric is actually dependent on the others. If a server is bottlenecked on CPU, for example, then it has no more CPU bandwidth available to utilize any extra disk or network capacity. Thus, the extra capacity of its non-bottlenecked resources is unusable, and metrics for those resources will cease growing with increasing load once the server hits its saturation point, thus causing them to exhibit a segmented relationship.

In addition, only one or two metrics showing a segmented relationship could be attributed to statistical noise instead of a meaningful relationship. Because this algorithm uses Davies' test at the 5% significance level, we expect that approximately 5% of the metrics tested will show a false-positive segmented relationship.

Next, the counts for each metric are normalized to the total number of metrics remaining for that server. As an example, suppose that the counts for one server are 2

linear, 3 segmented, and 3 none. These are normalized to sum to 1, giving 0.25 linear, 0.375 segmented, and 0.375 none. This is done because after removing metrics shared between tiers, the total number of metrics remaining may vary greatly between result sets, so the results are normalized to the same range to facilitate consistent decision in the next step.

4.4 Machine Learning

After getting normalized metric counts for each server, the final step in the algorithm is to use a machine learning classifier to decide if each set of counts indicates that the corresponding server is causing a performance bottleneck. Machine learning classifiers aim to classify *instances* into *classes*, based on *attributes*, after having been trained on previous instances. As a concrete example, consider Fisher's well-known iris flower data set [15], in which 150 iris flowers of three different species were measured and classified. In this example, each flower is an instance, each of the four measurements taken on it (sepal length, sepal width, petal length, petal width) is an attribute, and the three species are the output classes. The goal of a machine learning classifier in this example is to classify a new iris flower into the correct species, given those four measurements as input. In this work, each set of metric counts from one server forms an instance, with the three category counts as attributes. The output classes are "yes" and "no," signifying whether each set of metric counts indicates whether the corresponding server is exhibiting a performance bottleneck.

This work uses the well-known Naïve Bayes classifier, as implemented by Orange [12], a Python library for data mining and machine learning. Naïve Bayes determines the probability that a particular example belongs to a particular class, making an assumption of independence between the input attributes (the "Naïve" part of "Naïve Bayes"). Although this assumption is generally untrue in practice, the classifier gives quite good results. [13] However, Naïve Bayes can be problematic when two or more attributes are highly correlated; in that case, the classifier's performance can be improved by using only a subset of the available attributes, known as *feature selection.*

[24] The specific method of feature selection used here is discussed in Section 5.3.

The Naïve Bayes classifier provides several advantages over other classification algorithms. First, it is *observable* [19], meaning that it is possible to construct a visual representation of the trained classifier that shows the relative influence of each attribute on the final output. This gives a human operator some insight into what criteria are used for classification. Also, in addition to providing a classification as output, Naïve Bayes also provides, for each possible output class, a probability that the instance belongs in that class. Thus, a human operator can see how strong or weak the classifier's conclusion is.

Given the universality of the canonical characteristics, as discussed in Chapter 3, we expect that quite different applications will produce similar performance characteristics, and therefore will produce similar metric counting results for components that are and are not exhibiting performance bottlenecks, as long as those applications follow the properties laid out in Chapters 2 and 3. Thus, training data generated from different applications should show broad similarities, and classifiers generated from one application should perform reasonably well on another. This assertion will be examined in depth in Chapter 6.

Given that we know that a server showing many segmented relationships has likely reached its saturation point, the resulting machine learning classifiers are effectively reporting those sets of counts with many metrics in the "segmented" category as sources of a performance problem. Thus, invoking an elaborate machine learning system may not be strictly necessary; a simpler system of analysis that merely uses a threshold on the count of segmented metrics may be sufficient. However, a machine learning classifier contains its own training algorithm, whereas a threshold-based system would need to decide on a suitable threshold. In addition, there is a set of established evaluation methodologies for machine learning classifiers that can be used to assess the algorithm's effectiveness.

4.5 CPU-Ready Enhancement

The "CPU-Ready-Time" performance metric is of particular interest for virtualized environments. CPU-Ready indicates the amount of host processor time during which a virtual machine is runnable (i.e., there is work available for the virtual machine's virtual CPU), but cannot be scheduled because another virtual machine is currently scheduled. In a more abstract sense, CPU-Ready greater than 0 indicates an unmet demand for computational resources, in the same way that performance metrics such as CPU utilization indicate a fulfilled demand for resources.

This property of the CPU-Ready metric is domain knowledge that can enhance the algorithm's predictive ability in a virtualized environment. Instead of considering only the application's resource consumption, the algorithm can consider the application's total resource demand. Each CPU metric is scaled and normalized to a mean of 0 and standard deviation of 1, and then summed with the CPU-Ready metric, also scaled and normalized, to produce a representation of the application's total resource demand. Then, the regression and classification procedure is repeated, producing an additional set of category counts, which becomes an additional three attributes as input for the machine learning classifier.

4.6 Implementation

Performance metrics are monitored using `dstat` [2] and `esxtop` [6], through a remote command execution and data collection framework written in Java and backed by a PostgreSQL database. Latency information is collected from application logs and the Faban [23] load-generation tool. Statistical analysis and classification is done in Python, using R [22] (through RPy [1]). The Orange [12] toolkit is used for machine learning classification.

Chapter 5

Experimentation

In order to test the algorithm's accuracy, experiments were carried out with two test applications in which performance problems were deliberately introduced on one server of the application, and the algorithm was used to attempt to locate the performance bottleneck. Two applications in several configurations were used in order to test the effectiveness of the algorithm in environments other than that in which it was trained.

5.1 Olio

Olio [23] is a social networking and event planning application, designed specifically for benchmarking and performance comparison across application frameworks. Olio exists in three implementations (PHP, Ruby on Rails, and Java EE) and is built atop common Web application software (e.g., Linux, MySQL, Apache). Olio is a multi-tier application, divided into an application tier, holding the application's business logic, and a database tier, responsible for managing the application's persistent data.

The Rails and PHP implementations of Olio were used for these experiments. Olio was hosted on two virtual machines (VMs) running on a VMware ESX Server host. One VM hosts the persistent database tier, running either MySQL or PostgreSQL, depending on configuration. The second VM hosts the application tier, using the Thin [5] web server and Nginx [4] load-balancing proxy for the Rails implementation,

and the Apache HTTP server for the PHP implementation.

The Faban [23] load-generating tool was used to generate synthetic load for Olio. Faban is designed for benchmarking Web 2.0 applications, and supports a style of operation that simulates actions taken by real users when using the application under test. The Olio benchmark for Faban comprises seven operations, and chooses between them using a Markov chain. In addition, the benchmark simulates user think time by adding random delays between operations. Faban simulates multiple users by spawning a number of threads, each of which independently issues a single request to the application, waits to simulate think time, and then repeats. Each thread, therefore, acts as a single simulated user.

Figures 5-1 and 5-2 illustrate the Olio experimental setups.

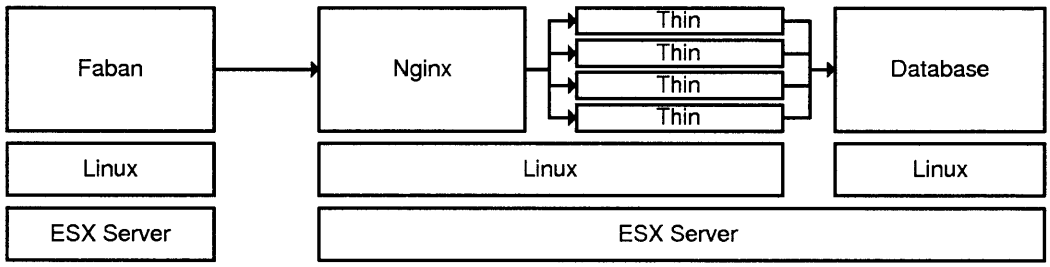


Figure 5-1: Experimental setup for Olio Rails.

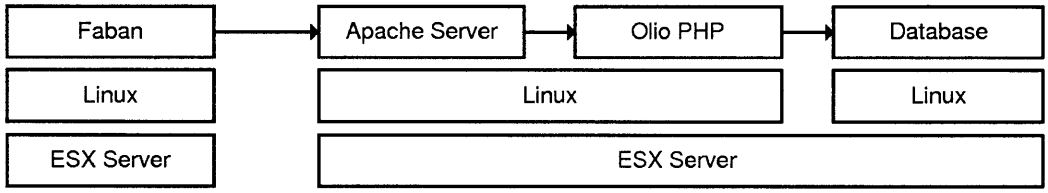


Figure 5-2: Experimental setup for Olio PHP.

Several tests were carried out, each with a particular performance problem deliberately induced in one tier of the application, while driving synthetic load to the application. Each test consisted of a series of executions of the Faban benchmark. Each execution was carried out with a 10-minute steady state, with all configuration parameters held constant during steady state. Each test then varied one parameter

over the set of benchmark executions. The specific parameter variations are shown in Table 5.1 and explained below. The tests carried out were as follows:

Application saturation bottleneck The testbed was configured to introduce a resource saturation bottleneck in the application tier by limiting the number of simultaneous connections that the application server can accept. This limits the maximum amount of CPU bandwidth that the application server can consume to much less than what is available.

For the Rails implementation, the testbed was configured with four instances of the Thin application server. For the PHP implementation, the Apache server's `MaxClients` parameter was set to 8192 simultaneous connections.

For this test, the number of simulated concurrent users was varied, from a point well below saturation to a point well above saturation. The endpoints for the variation in concurrent users were dependent on the Olio implementation and database server in use.

Database saturation bottleneck The testbed was configured to introduce a resource saturation bottleneck in the database tier by directly restricting the available CPU bandwidth using ESX Server resource limits. Constraints were chosen to be somewhat less than the CPU bandwidth consumed by the database VM when the application VM was being driven to its effective limit.

For the Rails implementation, the database VM was limited to 2659 MHz (equivalent to one physical CPU on the host machine); for the PHP implementation, the database VM was limited to 1595 MHz (approximately 75% of the CPU bandwidth consumed when the application tier was being driven to its limit).

For this test, the number of simulated concurrent users was varied, from a point well below saturation to a point well above saturation. As with the application bottleneck test, the endpoints for the variation in concurrent users were dependent on the Olio implementation and database server in use.

Background noise The testbed was configured with additional, unrelated VMs to consume CPU cycles, thereby limiting the CPU bandwidth available to Olio. Additional VMs were pinned to the same physical CPUs as one of the Olio VMs; separate tests were run for the application and database tiers. These extra VMs ran `lookbusy` [3], a synthetic CPU load generator that simply alternates between an empty loop and sleep to generate as much CPU activity as desired.

For this test, the CPU bandwidth consumed by each `lookbusy` VM, from the point of view of its guest OS, was varied from 0% to nearly 100%. (The algorithm that `lookbusy` uses for computing the balance between executing and sleeping breaks down when trying to set it at 100%; however, 95% or 99% is sufficient.)

Test	Configuration	Parameter
Application bottleneck	Rails, MySQL	Users: 50 to 300, by 10
	Rails, PostgreSQL	Users: 50 to 350, by 10
	PHP, MySQL	Users: 100 to 2000 by 100
Database bottleneck	Rails, MySQL	Users: 50 to 300, by 10
	Rails, PostgreSQL	Users: 50 to 350, by 10
	PHP, MySQL	Users: 100 to 2000, by 100
Application background noise	Rails, MySQL	Noise: 0% to 99%, by 3%
	Rails, PostgreSQL	Noise: 0% to 95%, by 5%
	PHP, MySQL	Noise: 0% to 95%, by 5%
Database background noise	Rails, MySQL	Noise: 0% to 99%, by 3%
	Rails, PostgreSQL	Noise: 0% to 95%, by 5%
	PHP, MySQL	Noise: 0% to 95%, by 5%

Table 5.1: Testbed variable parameter configurations.

5.2 Hadoop

Hadoop is a free, open-source distributed computing framework, comprising a computational engine similar to Google MapReduce, and a distributed filesystem similar to Google File System. Hadoop was chosen for experimentation to supplement Olio

because it uses a different architecture and different model of computation, and is therefore suitable for testing the generality of the algorithm across a wide spectrum of applications. Also, Hadoop was chosen as a true "cloud" application, with features typical of cloud applications, including built-in scaling and fault tolerance.

The algorithm was designed for multi-tier applications supporting a model of "requests" and "latency"; however, Hadoop is not a multi-tier application, and does not operate in the request/response model. Instead, Hadoop uses multiple independent nodes, each providing a thread pool for worker tasks, coordinated by a central work dispatcher (the JobTracker node). Due to this difference, it is necessary to somehow map the concepts of "request" and "latency" to something provided by Hadoop. For these experiments, each individual map or reduce job was considered a "request," and its time to completion was treated as its "latency."

A physical machine was configured as four virtual machines, each running a Hadoop DataNode and TaskTracker. One VM also ran Hadoop's NameNode and JobTracker. Another VM was configured with `lookbusy` VMs attached, to generate background noise, as in the Olio background noise tests (see Section 5.1). The background noise test was run, using Hadoop's Monte Carlo pi estimation benchmark. Each VM was configured with 5 map slots and 5 reduce slots. The benchmark was configured to run 240 tasks with one billion Monte Carlo samples per map.

5.3 Machine Learning

Data sets for machine learning classification were created by assembling the metric classification counts from each of the two application tiers for each test. The set of counts from one server in one test became one instance in the dataset. The data was divided into three sets that could then be combined as needed: Olio Rails, Olio PHP, and Hadoop. Each instance had two class variables attached to it: one for resource saturation and one for background noise, indicating whether that instance is an example of that particular problem. All of the instances from resource saturation tests were classified as "no" for background noise; likewise, all of the instances from

background noise tests were classified as "no" for resource saturation. The datasets used for this experimentation are reproduced in Appendix B.

An initial check on the accuracy of each resulting classifier was performed using leave-one-out cross-validation, where, for each instance in the dataset, that instance was omitted, the classifier was trained on the remaining instances, and then the classifier was tested on the omitted instance.

The goals stated at the beginning of this chapter were then tested by training classifiers on one data set, and then measuring their accuracy on another, thus testing the classifiers with an application configuration and environment different from that in which they were trained.

As discussed in Section 4.4, using a feature-selection procedure to eliminate extraneous variables in the input data can increase the classification accuracy. A brute-force search over the space of attribute subsets here was used for feature selection. Preference was given to the attribute subset with the best balanced accuracy using leave-one-out on the training data, with ties going to the subset with fewer attributes. This approach is reasonable here given the small space of attribute subsets (8 subsets for 3 attributes, or 6 with the CPU-Ready enhancement) and the speed of training the Naïve Bayes classifier.

5.3.1 Evaluation

After creating machine-learning classifiers and using them to classify instances as discussed above, a suitable method of evaluating their performance (i.e., how many instances were correctly classified) is needed. For measuring the classifier's performance, a common measure is classification accuracy (CA), which is simply the fraction of the test instances that were classified correctly. However, for this work, CA is not a suitable measure. In practice, one class (i.e., "no," this tier is not responsible for a performance problem) will occur much more frequently than the other. In this case, a "dumb" classifier could always pick the class that occurs more frequently, and produce a very high CA. To compensate for this effect, this work uses a measure known as *balanced accuracy*, which is the average of the sensitivity (proportion of

"yes" instances that were classified correctly) and the specificity (proportion of "no" instances that were classified correctly). [10]

Chapter 6

Results

This chapter will present an analysis of the data collected during experimentation and will demonstrate the ability of the algorithm to determine the server responsible for a performance bottleneck in an application system, particularly on a system different from that on which it was initially trained.

6.1 Olio

Tables 6.1 and 6.2 show the results of the metric classification procedure, after metrics common to both tiers have been removed but before the metric counts have been normalized, for the application bottleneck and database bottleneck tests, respectively. These results are from the Rails MySQL configuration. These results were chosen to illustrate the patterns and statistical characteristics that are indicative of particular performance problems.

	Metrics		Normalized	
Class	App	DB	App	DB
None	1	13	0.029	0.382
Linear	2	18	0.059	0.529
Segmented	31	3	0.912	0.088

Table 6.1: Metric classification results from application bottleneck test. "Metrics" columns are number of metrics in each category after removing metrics common to both tiers.

	Metrics		Normalized	
Class	App	DB	App	DB
None	21	0	1.00	0.00
Linear	0	0	0.00	0.00
Segmented	0	21	0.00	1.00

Table 6.2: Metric classification results from database bottleneck test. "Metrics" columns are number of metrics in each category after removing metrics common to both tiers.

Table 6.1 shows the classification results from the application bottleneck test, in which the application tier was deliberately configured to exhibit a performance bottleneck. In addition, Figure 6-1 shows the relationship between user CPU utilization and latency for the application (Figure 6-1A) and database (Figure 6-1B) tiers. Table 6.1 indicates that on the application server, 31 metrics showed a segmented relationship according to the classification procedure (described in Section 4.2); two metrics showed a linear relationship, and 1 showed no statistically-significant relationship. The right-hand columns of Table 6.1 show these same numbers, except normalized to the total number of metrics that were classified.

Note that many more application metrics show a segmented relationship than database metrics, indicating that the application tier is operating across both regions of the canonical performance characteristic, and has reached saturation at larger numbers of users. Also, note that metrics on the database server mostly fall into either the linear or none categories, and very few database metrics show segmented relationships. This indicates that the database tier is operating only in one linear region of the canonical relationship, and is therefore not reaching its saturation point. This same pattern is clearly visible in the CPU utilization plots. Hence, these results indicate that the saturation bottleneck is in the application tier, as expected.

Contrast these results with those from the database bottleneck test, shown in Table 6.2. Also, Figure 6-2 shows the relationship between user CPU utilization and latency for the application (Figure 6-2A) and database (Figure 6-2B) tiers. This test shows roughly opposite results: more database-tier metrics show a segmented relationship than application metrics. Figure 6-2 shows this relationship as well.

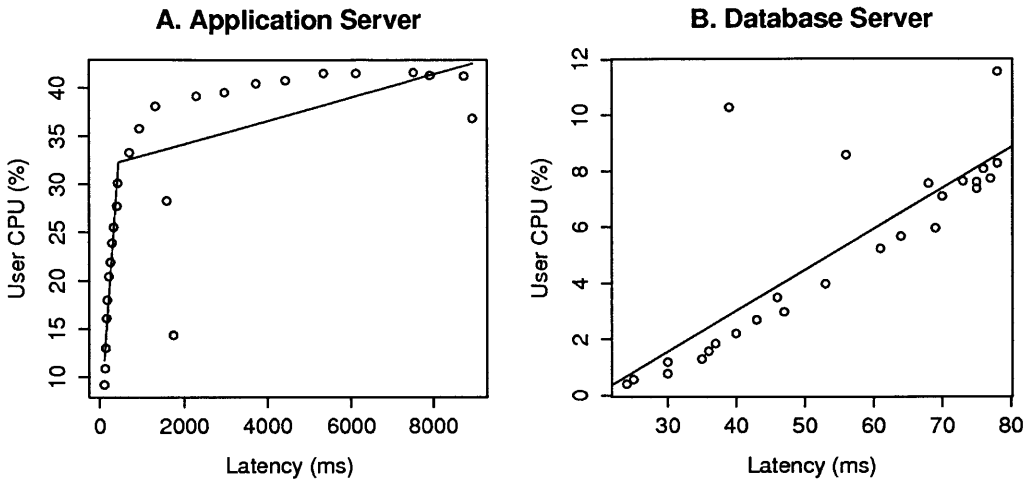


Figure 6-1: Relationship between latency and CPU utilization for the application bottleneck test. Best-fit lines are shown: segmented regression for the application tier, and simple linear regression for the database tier. Exact equations for the best-fit lines are not important; rather, this diagram illustrates the presence of segmented and linear relationships, respectively.

These results indicate that, as expected, the database tier has reached saturation and is the source of the bottleneck.

Table 6.3 shows machine learning results for Naïve Bayes classifiers trained on the Olio Rails dataset, and tested on the Olio PHP dataset, testing the accuracy of the classifiers with a different application configuration from that on which they were trained. Accuracies for the "Training" column are computed with leave-one-out cross-validation on the training data; accuracies in the "Test" column are the balanced accuracy of the test data set using classifiers trained on the training set. The high accuracies in both columns indicate that the algorithm can effectively locate the server responsible for a performance problem; furthermore, the high accuracies in the "Test" column demonstrate that the algorithm can locate performance problems in environments different from that on which it was trained.

Table 6.4 shows the effect of adding CPU-Ready attributes, making them available to the classifiers and feature selection procedure. Balanced accuracies were computed using leave-one-out with Naïve Bayes classifiers trained on the entire Olio dataset,

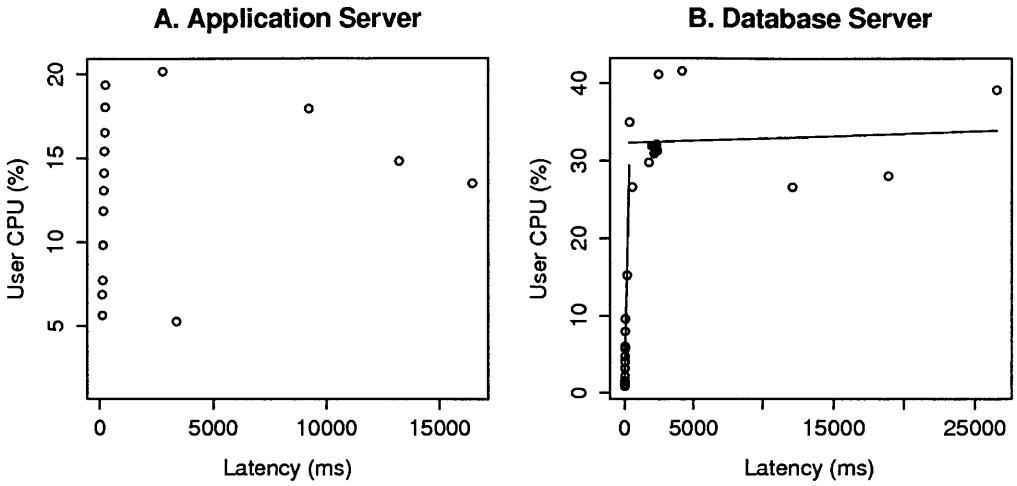


Figure 6-2: Relationship between latency and CPU utilization for the database bottleneck test. The best-fit line is shown for the database tier. The exact equation for the best-fit line is not important; rather, this diagram illustrates the presence of a segmented relationship there. The application tier data was not well-modeled by either a simple linear regression or a segmented regression.

Class Variable	Training	Test
Saturation	1.000	0.833
Noise	0.917	1.000
Bottleneck	1.000	1.000

Table 6.3: Machine learning accuracy, trained on Rails and tested on PHP.

using the attribute subsets that give the best balanced accuracy. Given that that CPU-Ready attributes are specific to and only meaningful in a virtualized environment, it would be expected that adding them would not result in any increase in accuracy for locating performance problems not related to virtualization. Table 6.4 shows results that are close to this expectation. For the saturation test, where the performance bottleneck is not virtualization-related, the classification accuracy improves slightly, by 2.8 percentage points. However, for the background noise test, where the cause of the bottleneck is virtualization-related, the classification accuracy improves more dramatically, by 11.1 percentage points.

With the high accuracies shown in Tables 6.3 and 6.4 from training the algorithm

Class Variable	Without	With	Change
Saturation	0.944	0.972	0.028
Noise	0.750	0.861	0.111
Bottleneck	0.917	1.000	0.083

Table 6.4: Machine learning accuracy, with and without CPU-Ready attributes.

on one configuration of the application and testing its accuracy on another, we can conclude that the algorithm can effectively locate the server responsible for a performance bottleneck. Furthermore, the algorithm is effective at locating bottlenecks even in an environment different from that in which it was trained. Given the generality of the characteristics that this algorithm is searching for, we could expect that this would be the case; furthermore, we also expect that this conclusion would carry over to much different applications; this assertion will be examined in more depth in the following section.

6.2 Hadoop

This section will present results from the experimentation carried out in Section 5.2, and demonstrate that the algorithm is capable of locating performance problems in an application much different from those discussed above.

Table 6.5 shows balanced accuracies from Naïve Bayes classifiers tested on the Hadoop dataset, after training on the combined Olio dataset, with and without the normalization procedure included in the algorithm. Tests for Hadoop were performed only with the Noise class variable, because only the background noise tests were run with Hadoop. Balanced accuracies were computed using the attribute subsets that give the best balanced accuracy on the training data. With normalization, the classifier achieved perfect accuracy on the Hadoop data. However, without normalizing the attributes to a 0-to-1 scale, the classifier performed considerably worse, indicating that normalizing to correct for varying ranges of counts of metrics is a valuable addition to the algorithm. Given that the classifier used in this experiment was trained on data from one application and tested on another application entirely, this experiment

shows that the resulting classifiers (and underlying performance theory, namely the canonical performance characteristic) are fairly general.

Normalized	Accuracy
Yes	1.000
No	0.625

Table 6.5: Machine learning results for Hadoop tests.

In addition, recall from Section 4.4 that a Naïve Bayes classifier outputs not only a chosen class, but also probabilities for each class. Of the four Hadoop test instances that were exhibiting a background noise bottleneck, when CPU-Ready attributes were included, the classifier predicted correctly with a mean probability of 0.84; without CPU-Ready attributes available, the classifier predicted all four instances correctly with a mean probability of 0.71. Even though the balanced accuracy was the same in each case, using the virtualization-specific CPU-Ready enhancement did strengthen the classifier's conclusions.

Given the perfect accuracy attainable by the algorithm in this experiment, we can conclude that it can effectively locate the source of a performance bottleneck in an application quite different from those for which it was designed. Again, this is expected due to the generality of the canonical characteristics that this algorithm is searching for in the application's performance data; hence, this algorithm should prove effective with a wide variety of applications, regardless of what application it was trained on originally.

Chapter 7

Conclusion

This work presented an algorithm designed to aid in performance diagnosis for multi-tier applications, by automatically identifying the specific component in a multi-component system responsible for a performance problem. This algorithm monitors each component, and using load vs. latency characteristics, uses statistical analyses to locate canonical patterns indicative of performance saturation, and uses a machine learning classifier to decide, based on the results of those statistical analyses, whether each machine in the system is reaching its maximum capacity and is thus the source of a performance problem.

This work has shown that this algorithm can reliably determine which component in a multi-component system is first reaching its maximum capacity, and thus can locate the source of a performance problem in an application system. In addition, although the algorithm was intended only for applications of one architectural type (interactive multi-tier applications), the algorithm was shown to be sufficiently general to apply to a broader spectrum of applications. Finally, a specific enhancement for virtualized environments was developed, and shown to produce improved results for virtualization-specific performance problems.

Appendix A

Performance Metrics Collected

This is a list of metrics collected from the testbed on each tier during the Olio and Hadoop experimentation.

A.1 Guest-Level Metrics

These metrics are collected from the VM guest OS using `dstat`.

User CPU	Int 18	Net Send
System CPU	Int 19	Processes Running
Idle CPU	1-min Load Avg	Processes Blocked
I/O Wait CPU	5-min Load Avg	Processes New
H/W Int CPU	15-min Load Avg	I/O Read
S/W Int CPU	Used Memory	I/O Write
Disk Read	Buffer Memory	Swap Used
Disk Write	Cache Memory	Swap Free
Page In	Free Memory	Interrupts
Page Out	Net Receive	Context Switches
Int 17		

A.2 VM-Level Metrics

These metrics are collected from each VM using `esxtop`.

CPU Used	CPU Ready	Net Recvd Dropped
CPU Run	CPU Costop	Net Received
CPU System	Net Out Dropped	Net Sent
CPU Wait		

A.3 Host Metrics

These metrics are collected from the VM host using `esxtop`.

CPU Proc Time	Disk Cmds/sec	Disk Avg Driver msec
CPU Util Time	Disk Reads/sec	Disk Avg Kernel msec
CPU Core Util Time	Disk Writes/sec	Disk Avg Guest msec
Swap Read	Disk MBytes read/sec	Disk Avg Queue msec
Swap Write	Disk MBytes write/sec	

Appendix B

Datasets

Datasets used for machine learning are reproduced in this chapter. Each row in a table represents one instance. Table column headings are as follows:

PT Problem type: saturation or noise, corresponding to which test this instance came from.

Config Which testbed configuration was used for this instance.

Tier Which tier this instance was taken from.

PL Problem location: which tier was the cause of the performance problem for this test.

None Value of the "none" attribute, before scaling by total number of metrics.

Lin Value of the "linear" attribute, before scaling by total number of metrics.

Seg Value of the "segmented" attribute, before scaling by total number of metrics.

R_None Value of the "none" attribute with the CPU-Ready enhancement, before scaling by total number of metrics.

R_Lin Value of the "linear" attribute with the CPU-Ready enhancement, before scaling by total number of metrics.

R_Seg Value of the "segmented" attribute, before scaling by total number of metrics.

BN Value of the "bottleneck" class variable, indicating whether this instance is an example of a performance bottleneck.

Sat Value of the "saturation" class variable, indicating whether this instance is an

example of a saturation bottleneck.

Noise Value of the "noise" class variable, indicating whether this instance is an example of a background noise bottleneck.

PT	Config	Tier	PL	None	Lin	Seg	R_None	R_Lin	R_Seg	BN	Sat	Noise
Saturation	MySQL	App	App	1	2	31	0	1	28	Yes	Yes	No
		DB	App	13	18	3	5	24	0	No	No	No
		App	DB	21	0	0	52	0	0	No	No	No
		DB	DB	0	0	21	0	0	52	Yes	Yes	No
Saturation	Postgres	App	App	5	7	25	1	25	14	Yes	Yes	No
		DB	App	15	20	2	31	6	3	No	No	No
		App	DB	35	0	0	47	0	0	No	No	No
		DB	DB	0	19	16	0	40	7	Yes	Yes	No
Noise	MySQL	App	App	3	2	3	0	3	2	Yes	No	Yes
		DB	App	5	0	3	5	0	0	No	No	No
		App	DB	12	0	2	28	1	2	No	No	No
		DB	DB	2	11	1	1	27	3	Yes	No	Yes
Noise	Postgres	App	App	8	8	8	3	6	21	Yes	No	Yes
		DB	App	13	8	3	24	4	2	No	No	No
		App	DB	13	2	4	32	0	1	No	No	No
		DB	DB	5	12	2	0	32	1	Yes	No	Yes

Table B.1: Olio Rails Dataset

PT	Config	Tier	PL	None	Lin	Seg	R_None	R_Lin	R_Seg	BN	Sat	Noise
Saturation	MySQL	App	App	2	4	26	0	4	25	Yes	Yes	No
		DB	App	28	2	2	27	1	1	No	No	No
		App	DB	28	0	0	41	0	0	No	No	No
		DB	DB	0	5	23	0	7	34	Yes	Yes	No
Noise	MySQL	App	App	2	7	24	0	5	32	Yes	No	Yes
		DB	App	24	8	1	36	0	1	No	No	No
		App	DB	21	0	8	43	0	2	No	No	No
		DB	DB	7	5	17	2	0	43	Yes	No	Yes

Table B.2: Olio PHP Dataset

PT	Tier	PL	None	Lin	Seg	R_None	R_Lin	R_Seg	BN	Noise
Noise	Hadoop1	Hadoop4	35	1	4	15	1	29	No	No
	Hadoop2		1	0	39	40	0	5	No	No
	Hadoop3		39	0	1	39	0	6	No	No
	Hadoop4		11	17	12	8	9	28	Yes	Yes
Noise	Hadoop1	Hadoop4	22	0	1	34	0	0	No	No
	Hadoop2		23	0	0	33	0	1	No	No
	Hadoop3		17	0	6	34	0	0	No	No
	Hadoop4		4	13	6	0	9	25	Yes	Yes
Noise	Hadoop1	Hadoop4	0	0	38	33	0	5	No	No
	Hadoop2		1	0	37	33	0	5	No	No
	Hadoop3		38	0	0	38	0	0	No	No
	Hadoop4		14	14	10	0	8	30	Yes	Yes
Noise	Hadoop1	Hadoop4	38	0	2	33	0	6	No	No
	Hadoop2		2	0	38	34	0	5	No	No
	Hadoop3		36	0	4	33	0	6	No	No
	Hadoop4		15	14	11	8	7	24	Yes	Yes

Table B.3: Hadoop Dataset

Bibliography

[1] A simple and efficient access to R from Python. `http://rpy.sourceforge.net/`.

[2] Dstat: versatile resource statistics tool. `http://dag.wieers.com/home-made/dstat/`.

[3] lookbusy: a synthetic load generator. `http://www.devin.com/lookbusy/`.

[4] Nginx. `http://www.nginx.org/`.

[5] Thin: a fast and very simple Ruby web server. `http://code.macournoyer.com/thin/`.

[6] Using esxtop to troubleshoot performance problems. `http://www.vmware.com/pdf/esx2_using_esxtop.pdf`.

[7] M.K. Aguilera, J.C. Mogul, J.L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 74–89. ACM, 2003.

[8] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, et al. Above the clouds: A berkeley view of cloud computing. Technical report, Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[9] S.K. Barker and P. Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pages 35–46. ACM, 2010.

[10] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J.S. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation-Volume 6*, pages 16–16. USENIX Association, 2004.

[11] R.B. Davies. Hypothesis testing when a nuisance parameter is present only under the alternative. *Biometrika*, pages 247–254, 1977.

[12] J. Demšar, B. Zupan, G. Leban, and T. Curk. Orange: From experimental machine learning to interactive data mining. *Knowledge discovery in databases: PKDD 2004*, pages 537–539, 2004.

[13] P. Domingos and M. Pazzani. Beyond independence: Conditions for the optimality of the simple bayesian classi er. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 105–112, 1996.

[14] D.D. Dunlop and A.C. Tamhane. *Statistics and Data Analysis: From Elementary to Intermediate.* Prentice Hall, 2000.

[15] R.A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Human Genetics*, 7(2):179–188, 1936.

[16] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. Netlogger: A toolkit for distributed system performance analysis. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings. 8th International Symposium on*, pages 267–273. IEEE, 2000.

[17] N.J. Gunther. Benchmarking blunders and things that go bump in the night. *Arxiv preprint cs/0404043*, 2004.

[18] N.J. Gunther. *Analyzing computer system performance with Perl:: PDQ.* Springer, 2005.

[19] M. Možina, J. Demšar, M. Kattan, and B. Zupan. Nomograms for visualization of naive bayesian classifier. *Knowledge Discovery in Databases: PKDD 2004*, pages 337–348, 2004.

[20] V.M.R. Muggeo. Estimating regression models with unknown break-points. *Statistics in Medicine*, 22(19):3055–3071, 2003.

[21] V.M.R. Muggeo. Segmented: an r package to fit regression models with broken-line relationships. *R news*, 8(1):20–25, 2008.

[22] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2011. ISBN 3-900051-07-0.

[23] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, S. Patil, A. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of CCA*, 2008.

[24] I.H. Witten, E. Frank, and M.A. Hall. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, 2011.