Virtualized Application Performance Prediction Using System Metrics

by

Skye A. Wanderman-Milne

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

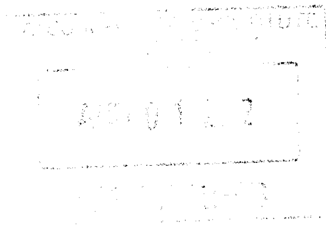
Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 2012

©2012 Massachusetts Institute of Technology
All rights reserved.

ARCHIVES



Author ....
Department of Electrical Engineering and Computer Science
May 21, 2012

Certified by ....
Una-May O'Reilly
Principal Research Scientist, CSAIL
Thesis Supervisor

Certified by ....
Prof. Saman Amarasinghe
Department of Electrical Engineering and Computer Science
Thesis Co-Supervisor

Certified by ....
Steve Muir
Director, VMware Academic Program
Thesis Co-Supervisor

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

Virtualized Application Performance Prediction Using System Metrics

by

Skye A. Wanderman-Milne

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 2012

In partial fulfillment of the requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Virtualized datacenter administrators would like to consolidate virtual machines (VMs) onto as few physical hosts as possible in order to decrease costs, but must leave enough physical resources for each VM to meet application service level objectives (SLOs). The threshold between good and bad performance in terms of resource settings, however, is hard to determine and rarely static due to changing workloads and resource usage. Thus, in order to avoid SLO violations, system administrators must err on the side of caution by running fewer VMs per host than necessary or setting reservations, which prevents resources from being shared. To ameliorate this situation, we are working to design and implement a system that automatically monitors VM-level metrics to predict impending application SLO violations, and takes appropriate action to prevent the SLO violation from occurring. So far we have implemented the performance prediction, which is detailed in this document, while the preventative actions are left as future work.

We created a three-stage pipeline in order to achieve scalable performance prediction. The three stages are prediction, which predicts future VM ESX performance counter values based on current time-series data; aggregation, which aggregates the predicted VM metrics into a single set of global metrics; and finally classification, which for each VM classifies its performance as good or bad based on the predicted VM counters and the predicted global state. Prediction of each counter is performed by a least-squares linear fit, aggregation is performed simply by summing each counter across all VMs, and classification is performed using a support vector machine (SVM) for each application. In addition, we created an experimental system running a MongoDB instance in order to test and evaluate our pipeline implementation. Our results on this experimental system are promising, but further research will be necessary before applying these techniques to a production system.

Thesis Supervisor: Una-May O'Reilly
Title: Principal Research Scientist, CSAIL

Thesis Co-Supervisor: Prof. Saman Amarasinghe
Title: Department of Electrical Engineering and Computer Science

Thesis Co-Supervisor: Steve Muir
Title: Director, VMware Academic Program

Acknowledgments

I would like to thank my adviser Una-May O'Reilly for her unflagging support, guidance, and encouragement. I could always count on her to put things in perspective and remind me of the big picture when I got too caught up in the details, and she helped me find solutions to many problems as well as contributed countless useful ideas. I always enjoyed talking to her, either about the project or any other topic. I would also like to thank Saman Amarasinghe, who helped guide the direction of the project. I appreciate him finding the time in his impossibly-busy schedule to sit down and discuss how to make a meaningful impact with this project.

I am immensely grateful to everyone at VMware who has helped me along the way. Rean Griffith offered me advice on the direction of the project, data conditioning, MongoDB, and everything in between. I wish I could have worked in the Palo Alto office with him more often because I was always amazed at the progress made when I did. Xiaoyun Zhu seemed to have a pointer to every paper relevant to our project, and greatly helped in establishing a context for our work. Ravi Soundararajan was always willing to help me navigate VMware and its products, and when he couldn't help me he would quickly connect me with the person who could. None of this would have been possible without the support of Julia Austin, Steve Muir, and Rita Tavilla.

Over the summer, I worked with Gartheeban Ganeshapillai, who created the foundation of the project and taught me an embarrassing amount about machine learning. I would also like to thank Grant Nowell and Hari Swarna for providing me with the vmware.com server logs.

Lastly, I would like to thank my friends and family for all their support over the last year, especially RJ Ryan, Amelia Arbisser, and Carmel Dudley for talking me through the various obstacles I faced.

Contents

List of Figures 9

List of Tables 11

1 Introduction 13

1.1 Goal 14

1.2 Approach 14

1.3 History 15

1.4 Roadmap 15

2 Machine Learning Pipeline Overview 17

2.1 Prediction 19

2.2 Aggregation 20

2.3 Classification 21

3 Experimental System 23

3.1 Overview 23

3.2 Testbed 24

3.3 Traffic Pattern: vmware.com 24

3.4 Application: MongoDB 26

3.5 Hogger VM 27

3.6 Load generator (Rain) 28

3.7 Data collection 29

3.7.1 Timing 29

3.8 SLO definition 29

3.8.1 Response time clustering 31

4 Pipeline Implementation and Results **33**
4.1 Pipeline performance metrics 33
4.1.1 Prediction metrics 33
4.1.2 Classification metrics 34
4.2 Training, validation and test sets 35
4.3 Preparing counters 36
4.4 Prediction 37
4.5 Classification 39
4.5.1 SVM parameter grid search 39
4.5.2 Feature selection 40
4.5.3 Weight violations 40
4.6 Pipeline results 40

5 Future Work **45**
5.1 Experiment configurations 45
5.2 Refine pipeline stages 46
5.3 Online learning 47
5.4 End-to-end system 47

6 Related Work **49**
6.1 Performance Modeling and Anomaly Detection 49
6.1.1 Anomaly prediction 50
6.2 Resource Scaling 50
6.2.1 Physical servers 51
6.2.2 VMs 51

7 Conclusions **55**

A ESX performance counters **57**

B Counter Prediction Results **59**

Bibliography **79**

List of Figures

2-1 Machine learning pipeline summary . . . 17

2-2 Overview of machine learning pipeline . . . 18

3-1 Experimental system overview . . . 23

3-2 vmware.com Apache request log sample . . . 24

3-3 vmware.com traffic pattern . . . 25

3-4 MongoDB response times . . . 26

3-5 Deriving SLO violations from response times . . . 30

(a) MongoDB 99th-percentile latency . . . 30

(b) MongoDB 99th-percentile smoothed latency . . . 30

(c) MongoDB SLO violations . . . 30

4-1 Training, validation, and test partitions . . . 36

4-2 Smoothed vs. unsmoothed counters . . . 37

4-3 Final performance prediction results . . . 42

B-1 Actual vs. predicted counters . . . 64

List of Tables

1.1 Historical experiment space 15

2.1 Machine learning pipeline stages 22

2.2 Prediction parameters 22

2.3 Classification parameters 22

3.1 SLO parameters 31

3.2 SLO parameter values 32

4.1 Prediction parameter values 37

4.2 Linear fit prediction results (all features) 38

4.3 Linear fit prediction results (selected features) 38

4.4 SVM parameter values 39

4.5 Counters used as classifier features 41

4.6 Final performance prediction results 43

(a) Training set results 43

(b) Validation set results 43

(c) Test set results 43

5.1 Potential experimental systems 45

B.1 Counter prediction results 59

Chapter 1

Introduction

Virtualization has significantly changed datacenter management and resource usage. Virtualization allows a single physical machine to run multiple independent virtual machines (VMs), each of which is nearly indistinguishable from a regular physical machine to the user of the VM. Before the advent of virtualization, system administrators had to purchase and manage a separate server whenever an application needed to be isolated, either for performance, business, or legal reasons (e.g., a company's website should not go down when the mail server requires a system restart, multiple tenants in a datacenter cannot share a machine). This meant resource utilization was very low, as most applications only use a tiny portion of the resources available on a physical system. With virtualization, system administrators can consolidate many virtual machines onto a smaller number of physical hosts, allowing for better resource utilization and subsequent energy savings, as well as ease in datacenter management.

System administrators would like to consolidate VMs onto as few physical hosts as possible in order to decrease costs associated with energy usage, cooling, etc. It is common practice to "overcommit" resources, meaning the total virtual resources of the VMs exceeds the total physical resources of the host (for example, creating two VMs with 12 GB of RAM each on a host with only 16 GB of RAM). This is possible because although the VMs are configured with more resources than are available, they will rarely use all available resources, allowing the same resources to be shared between VMs. Of course, this strategy can only be taken so far – at some point, there are not enough resources and performance suffers. Due to changing workloads and resource usage, the threshold between good and bad performance is rarely static, so system administrators must err on the side of caution by running fewer VMs per host than necessary or setting reservations,

which prevents resources from being shared. These precautions provide a buffer in case of a sudden resource usage increase but decrease consolidation, making the datacenter less efficient.

Further complicating matters, although VM users must specify resource settings (e.g., CPU and memory reservations), users ultimately care about application performance, not VM performance. Many users must meet service level objectives (SLOs), which stipulate that a certain level of application performance be maintained at all times (see Section 3.8 for an example of an SLO definition). However, the relationship between resource settings and performance is often non-intuitive, forcing users to make conservative resource settings that cost more and waste energy in order to account for unexpected resource usage. Solutions exist for VMs experiencing poor performance due to lack of resources: resource settings can be changed dynamically, or in the case where the host's resources are too overcommitted, running VMs can be transferred to a different host with more resources available using vMotion. vSphere's Distributed Resource Scheduler (DRS) [16] automatically uses vMotion to consolidate VMs while honoring resource settings; however, it cannot take application performance into account, and can only react to changes in resource usage, rather than take proactive measures.

1.1 Goal

We aim to design and implement a system that automatically monitors VM-level metrics to predict impending application SLO violations, and takes appropriate action to prevent the SLO violation from occurring. So far we have implemented the performance prediction, which is detailed in this document, while the preventative actions are left as future work. We have limited ourselves to VM metrics, rather than application metrics as well, because application metrics are not always available and VM metrics allow us to consider all VMs on the system, whereas integrating metrics from different applications is technically difficult and may be logistically infeasible or insecure. SLO violations are defined per application.

1.2 Approach

We created a three-stage pipeline in order to achieve scalable performance prediction. The three stages are prediction, which predicts future VM ESX performance counter values based on current time-series data; aggregation, which aggregates the predicted VM metrics into a single set of global metrics; and finally classification, which for each VM classifies its performance as good or bad

based on the predicted VM counters and the predicted global state. Prediction of each counter is performed by a least-squares linear fit, aggregation is performed simply by summing each counter across all VMs, and classification is performed using a support vector machine (SVM) for each application. See Chapter 2 for a detailed explanation of our approach.

1.3 History

Preliminary work was done by Lawrence Chan in Fall 2010 - Spring 2011 for his M.Eng. thesis [10]. Chan showed that application performance could be discovered through ESX counters. The pipeline was developed along with a proof-of-concept implementation during summer 2011 by Gartheeban Ganeshapillai and Skye Wanderman-Milne [13]. This document covers work done since then that refines the pipeline and runs it on more complex application workloads. Table 1.1 summarizes the major parameters of these efforts.

Table 1.1: Historical experiment space. A summary of the major parameters of the experimental test systems used in previous work and this work.

	Applications	Performance metric	Traffic
Fall 2010/ Spring 2011	Apache web server (homebrew app)	Percentage to breakdown	sawtooth
Summer 2011	Apache web server (Wordpress)	Clustered latency into "breakdown" and "non-breakdown" regions	sawtooth, Stack Overflow
Fall 2011/ Spring 2012	MongoDB server, hogger VM	Hard-coded SLO definition	vmware.com

1.4 Roadmap

Chapter 2 goes over the high-level machine learning pipeline in more detail. Chapter 3 presents the data we trained and tested the pipeline with, including how we collected it. Chapter 4 goes over the pipeline implementation, specific techniques we used for conditioning the data, the values chosen for various parameters, and subsequent results. Chapter 5 discusses future work to be done, and Chapter 6 covers similar work done by others. We conclude in Chapter 7.

Chapter 2

Machine Learning Pipeline Overview

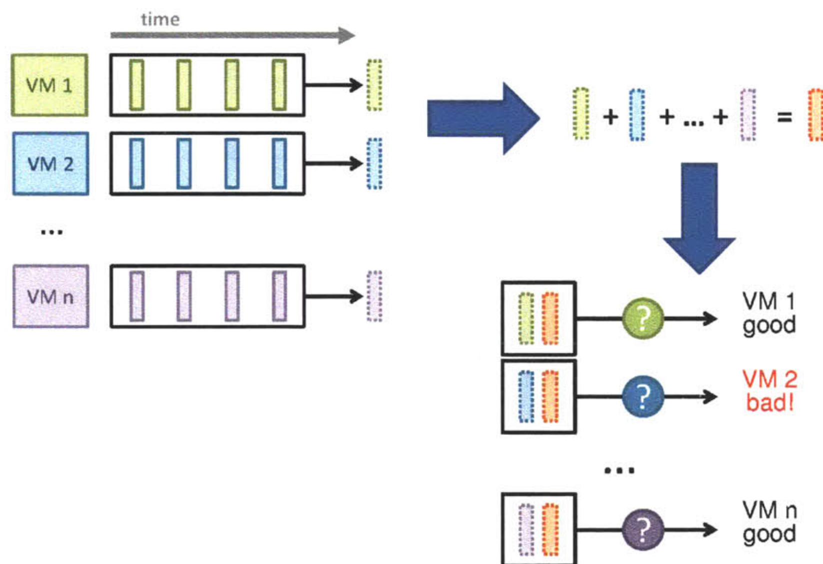
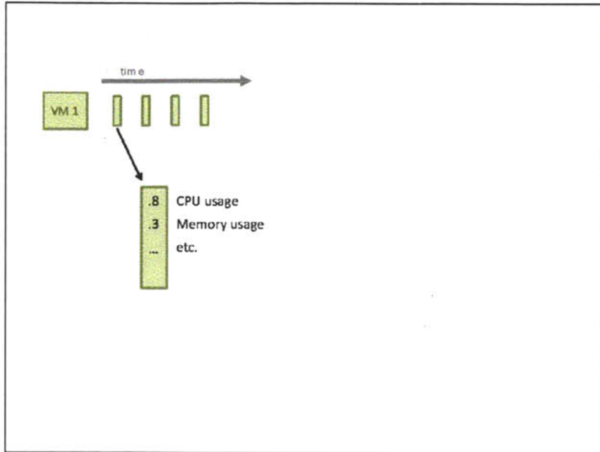
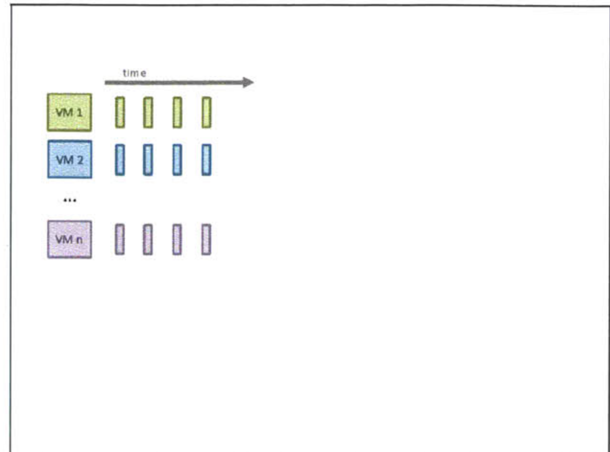


Figure 2-1: Machine learning pipeline summary showing the three stages: counter prediction (upper left), counter aggregation (upper right), and performance classification (lower right). See Figure 2-2 for details on each stage.

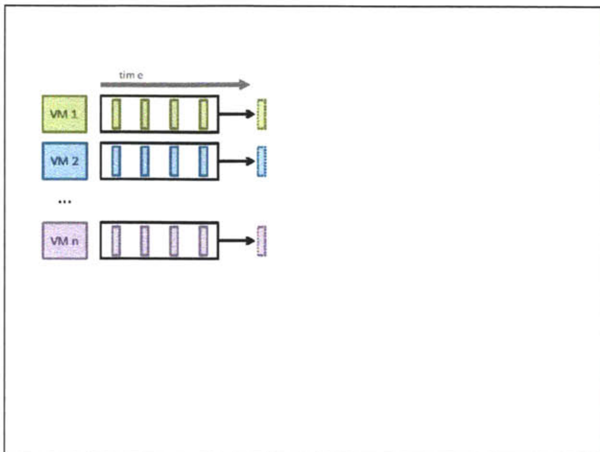
Our system is designed to perform application performance prediction in a scalable fashion. The system is organized in a three-stage pipeline, as shown in Figure 2-1. Each host in the system runs an instance of the pipeline. The three stages are prediction, which predicts future VM counter values based on current time-series data; aggregation, which aggregates the predicted VM



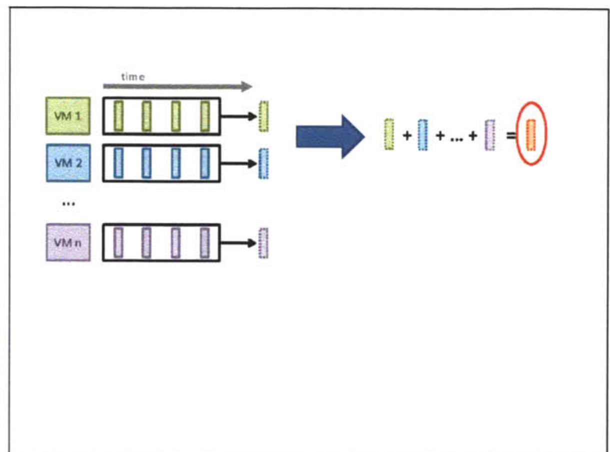
(a) The performance counters of a given VM are collected over time.



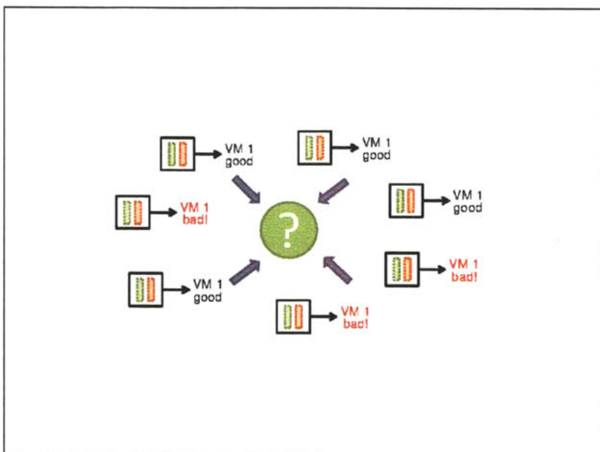
(b) Many VMs may be running on a single host, each with its own counters.



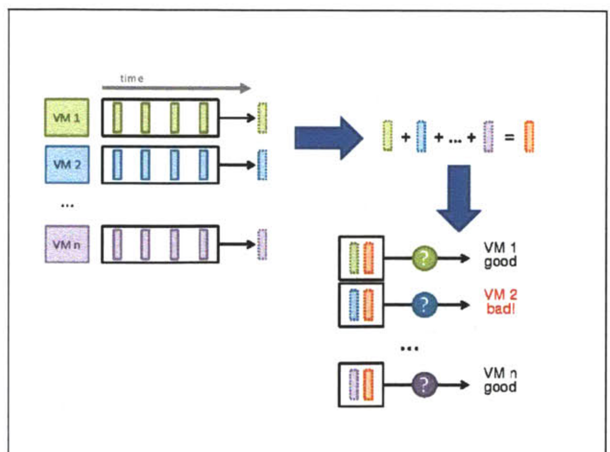
(c) We predict future counter values independently for each VM and counter based on past values.



(d) We aggregate the predicted counters into a global metric by summing each counter.



(e) We train a performance classifier for each VM that takes that VM's counters and the global counters as input. The classifier is trained using example datapoints labeled with the correct answer.



(f) Using the classifiers and incoming predicted counters, classify whether each VM's performance will be good or bad.

Figure 2-2: Overview of machine learning pipeline

metrics into a single set of global metrics; and finally classification, which for each VM classifies its performance as an SLO violation or non-violation based on the predicted VM counters and the predicted global state. These stages are described in more detail below as well as in Figure 2-2, and the inputs and outputs of each stage are summarized in Table 2.1. Different techniques and parameters can be used for each of these stages without affecting the other stages. This modular approach allows improvements to be made to each stage with minimal effort and few unintended consequences for the rest of the system. Furthermore, each stage can be treated as an independent problem using labeled training data, although we only take this approach with the classification stage (we found simpler methods to be sufficient for the first two stages given our data).

2.1 Prediction

For each VM running on the host, we collect ESX performance counters periodically at a fixed interval (Figures 2-2a and 2-2b). These performance counters represent different system metrics such as CPU, memory, and network usage. Given historical lag k and predictive lag h, each time new counter readings are collected, the latest k readings are used to predict the values of each counter h timesteps in the future (Figure 2-2c). Each counter is predicted independently, and although for a given counter we only use that counter's values for prediction, a different technique could use past values of many counters to predict a single counter. Since prediction is also done independently for each VM, we avoid having to consider which other VMs are running on the host, if VMs are being added or removed from the host, etc. (of course, the other VMs running on the host may have an effect on the given VM's performance counters, but we assume that this interaction does not greatly inhibit prediction accuracy). The predicted counter values represent the predicted future system state of the VM.

We perform prediction via least-squares linear regression, fitting a line through the input k datapoints and returning the future point h steps ahead on the line. See Table 2.2 for a summary of all prediction parameters.

2.2 Aggregation

Although we predict each VM's system state independently, we do not assume that other VMs running on the host have no effect on application performance. Thus, in addition to each VM's own state, we must use some information about the rest of the host machine as input to the classifiers in order to make an accurate performance prediction.

One solution would be to use counters from every VM running on the host as input to the classification stage. However, this would mean that new performance classifiers would have to be trained every time a VM was added to or removed from the host, since the input to the classifier would change with the VMs currently running. Therefore this is not an easily scalable solution.

Another solution is to use the host-level performance counters in addition to the specific VM's counters as input to each VM's classifier. This solves the problem of dynamic VM configurations – the state of all VMs running on the host is "summarized" by the host counters, allowing us to use a fixed-size input to the classifiers. In order to provide host counters as input to the classifiers, though, we must predict the host counters in addition to each VM's counters during the prediction stage. Although we do not train models for counter prediction, the system is designed to allow for this possibility, in which case a new host counter model would be needed when the VMs on the host changed, introducing a scalability issue similar to the one mentioned above. Additionally, if many VMs with unique usage patterns are running on the same host, the host counters may be very hard to predict.

The solution we use is to aggregate the predicted VM counters into a single set of counters representing the global state of the host, and use these aggregated counters as the additional input to the classifiers (Figure 2-2d). The aggregate is essentially used like predicted host counters would be, allowing us to maintain consistent input to the classifiers while avoiding having to predict the host counters directly. We perform the aggregation by summing each counter across all VMs (e.g., the aggregate CPU usage is the sum of each VM's CPU usage), although more complicated methods could be used such as learning which types of aggregate functions (sum, mean, max, etc.) are best for each counter.

2.3 Classification

Finally, given each VM's predicted counters and the aggregated counters representing the global host state, we classify whether or not the application running on each VM is performing adequately (Figure 2-2f). This requires defining what constitutes "good" and "bad" performance for a given application (see 3.8 for details on the definition we used). Each application may have a different performance metric, since we create an independent classifier for every application.

We trained support vector machines (SVMs) for each VM to perform classification (Figure 2-2e) using the LIBSVM library [11] via mlpy [6], a Python machine learning module. See Table 2.3 for a summary of the relevant parameters.

Table 2.1: Machine learning pipeline stages

Stage	Input	Output
Prediction	VM counters time-series	predicted VM counters
Aggregation	predicted VM counters	aggregated global counters
Classification	predicted VM counters, aggregated counters	SLO violation or not

Table 2.2: Prediction parameters (see Section 2.1)

Parameter	Description
Counter collection interval	How often performance counter values are fetched from the host. This defines the timestep length.
Historical lag	The number of consecutive counter datapoints used as input to the predictor
Predictive lag	The number of timesteps ahead the predictor predicts a counter value
Counter smoothing window†	The window size, in number of datapoints, over which to perform a moving average of each counter's values

† Shared with classification parameters

Table 2.3: Classification parameters (see Section 2.3)

Parameter	Description
Input counters	Which counters to include as input to the classifier (i.e., the features)
Counter smoothing window†	The window size, in number of datapoints, over which to perform a moving average of each counter's values
C††	Determines cost of mistakes in training data
γ††	RBF kernel parameter
Weight††	Penalty of violation (vs. penalty of non-violation, which is always set to 1)

† Shared with prediction parameters
†† SVM parameter

Chapter 3

Experimental System

3.1 Overview

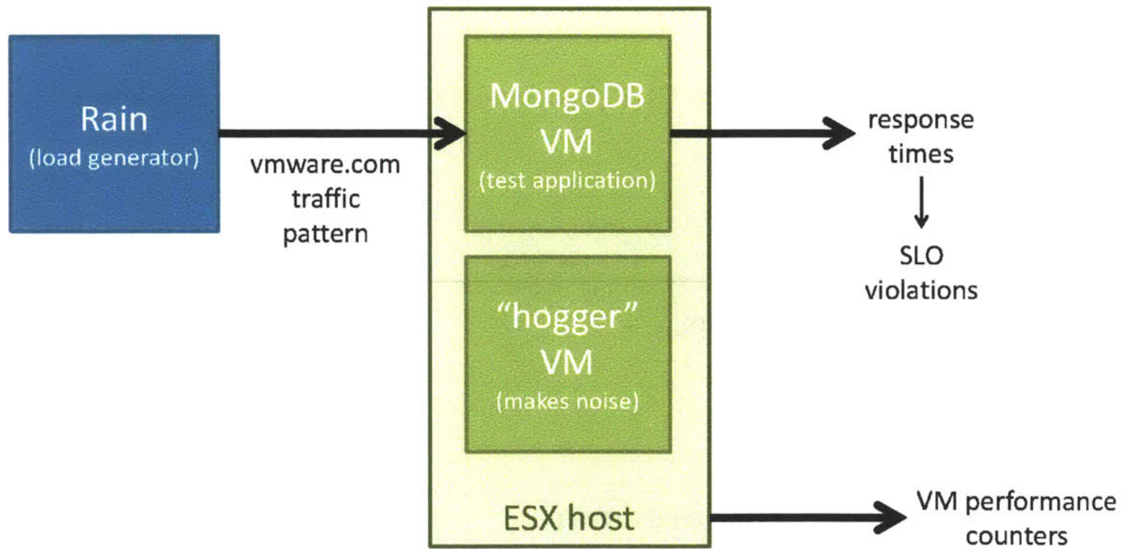


Figure 3-1: Experimental system overview

The machine learning pipeline described in Chapter 2 predicts performance based on ESX performance counters, so at a bare minimum it needs counter time-series data to operate. Additionally, application performance data is needed in order to train the performance classifiers (see Section 2.3). While this data is intended to be easily collected from live applications running in a public or private virtualized cloud, we unfortunately did not have access to any such applications and instead

generated realistic artificial data as shown in Figure 3-1 in order to train and test our system.

Our test application is MongoDB [1], a NoSQL database. We run a MongoDB instance on a VM and use its log to collect latency data, which we use to label good and bad performance intervals. Besides the MongoDB VM, there is a "hogger" VM that uses system resources in order to simulate other applications creating load on the system. We collect ESX performance counters from both of these VMs. We induce load to the MongoDB instance using Rain [3, 7], a load generator that replays a traffic pattern based on real data from vmware.com's server logs.

3.2 Testbed

We ran two vSphere hosts, one for our test application and hogger VMs and one for the load generator. We separated the load generator so its resource usage would not have an effect on our experiments. The two hosts both had the following specifications:

- Dell Poweredge R610 server
- 12 x Intel Xeon CPU X5670 2.925 GHz (2 sockets, 6 cores/socket, hyperthreaded)
- 40 GB RAM
- 930 GB disk
- 1000 Mbps network interface
- VMware vSphere 4.1 Enterprise Plus
- VMware vCenter 4.1 Standard

3.3 Traffic Pattern: vmware.com

```
10.113.78.14 "122.208.160.20, 10.209.38.223, 64.209.38.232" - -
[23/Oct/2011:04:22:15 -0700] "GET /products/ HTTP/1.1" 200 109468
"https://www.vmware.com/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_2)
AppleWebKit/535.7 (KHTML, like Gecko) Chrome/16.0.912.4 Safari/535.7"
```

Figure 3-2: vmware.com Apache request log sample. Although this is shown on multiple lines, this corresponds to a single line (i.e., a single request) in the log.

Since our system makes predictions based on time-series data, it is very important for us to test our system using realistic time-series data, rather than simple benchmarks that are not based on

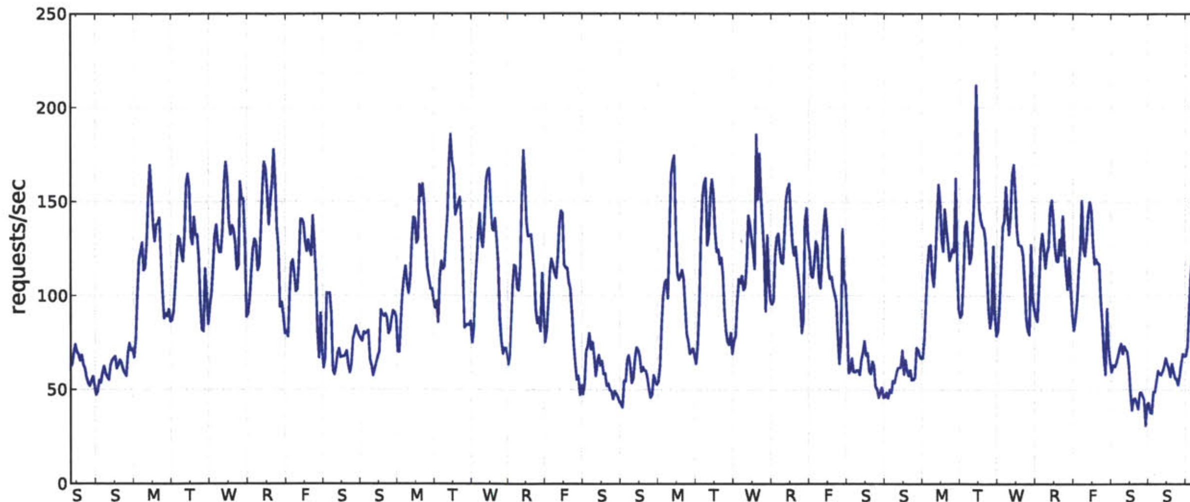


Figure 3-3: vmware.com traffic pattern. This traffic pattern represents 30 days' worth of requests to the vmware.com servers. We replayed this month-long pattern in 24 hours, condensing each hour down to two minutes. The request rates were also scaled down.

actual usage patterns. We used log data from the servers hosting vmware.com to construct realistic time-series traffic data.

The server logs we accessed recorded every request served from the vmware.com domain over a 30-day period, totaling 258,047,454 requests. A sample of these logged requests is shown in Figure 3-2. For each request, we parsed the timestamp and discarded all other data. We calculated a histogram over the requests based on their timestamps, with each "bin" representing the number of requests that occurred within a 1-hour interval for a total of 720 bins. This histogram represents the traffic pattern, as shown in Figure 3-3. The traffic pattern exhibits clear diurnal behavior: traffic spikes during the day, and then goes down at night. In addition to this daily cycle, there is a weekly cycle, with weekdays seeing much more traffic than weekends.

We scaled this traffic pattern both horizontally and vertically. In order to replay the full month's worth of data in a reasonable amount of time, we compressed the timescale, replaying each datapoint for two minutes instead of the full hour it represents. Since vmware.com experiences more traffic than our single application server can handle, especially when an hour's worth of requests are compressed into a two-minute interval, we linearly scaled down the number of requests per time interval, lowering the request rate while preserving the relative differences between each datapoint. In Rain, load is adjusted via the initial number of "users," which is the number of threads sending synchronous requests to the database. The number of users is then adjusted at each time interval

according to the traffic pattern. We set the initial number of users to 30, which corresponds to roughly 50-100 requests per second.

3.4 Application: MongoDB

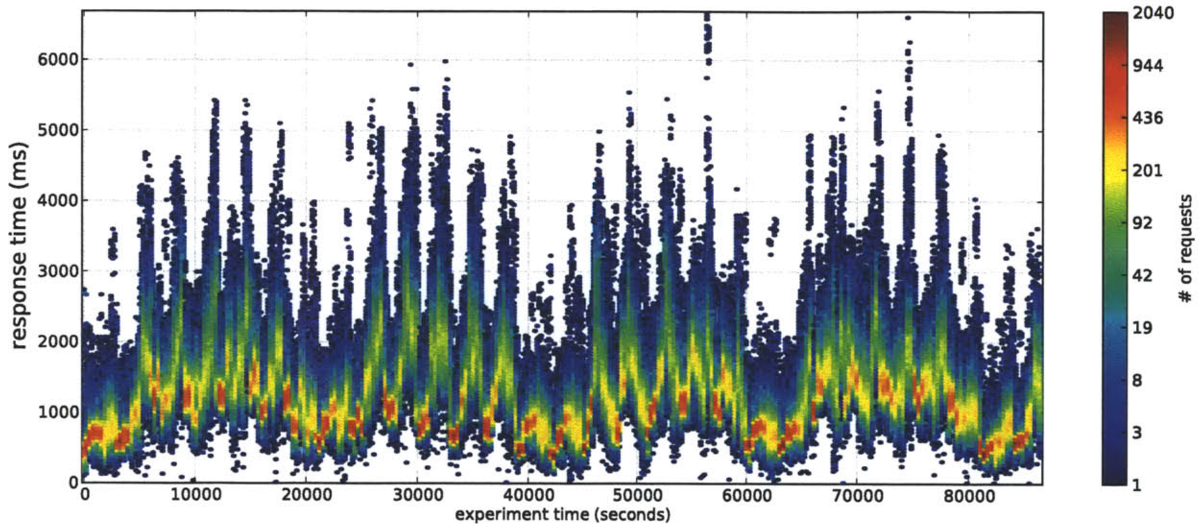


Figure 3-4: MongoDB response times

Although we have access to the vmware.com traffic, we do not have access to the web application that comprises the vmware.com web site, meaning we must run the traffic pattern on a different application. We used MongoDB as our test application. MongoDB [1] is an open-source NoSQL database often used in web applications, making it an appropriate substitution for the vmware.com web application since a database in a web application will likely experience a similar load as the rest of the application. We measured the response time of each read request via MongoDB's log file, the results of which are shown in Figure 3-4. (Although we also issued write requests, we did not collect their response times because difference types of queries have different performance characteristics. In a production system, write response times would be collected and independently analyzed as well, but for test purposes we concerned ourselves with only one type of request). Note that the response times in Figure 3-4 have peaks corresponding to those in the traffic pattern we used, shown in Figure 3-3.

We chose MongoDB in order to create a complex, non-trivial prediction task. MongoDB consumes CPU, memory, network, and disk resources, all of which contribute to its performance but

none of which are strongly linearly correlated to latency. In our earlier work [13], the simple web server we used as our test application primarily consumed CPU resources, which were very strongly correlated with latency. While this setup yielded encouraging results, we wished to test the robustness of our techniques on applications with more complex behavior. In hindsight, we may have overly complicated our experimental setup, which is further discussed in the Future Work chapter.

Additionally, MongoDB exhibits clear latency changes in response to load, provided it must fetch some documents from disk. This is important so we can reliably induce enough variance in response time to create meaningful violation and non-violation labels. Initially we had created a MongoDB database small enough to fit entirely in memory, but it was too difficult to create enough load to produce noticeable changes in latency.

Using Rain's MongoUtil utility, the MongoDB database was loaded with a single collection containing one million documents. Each document had an indexed "key" field containing a unique identifier between 1 and 1000000 and a "value" field containing a 32KB sequence of random bytes. The total size of the database, including the index, was approximately 31GB. We created large documents in order to induce disk I/O, which greatly increases latency, allowing us to produce significant changes in latency based on load.

The VM running the MongoDB instance had the following specifications:

- Name: skye-mongo2
- 4 vCPUs
- 8 GB RAM
- 60 GB virtual disk
- Ubuntu Server 10.10 64-bit

3.5 Hogger VM

For simplicity, rather than run multiple applications on the host, we simulated other application activity with a "hogger" VM that uses a variable amount of system resources over the course of the experiment. Since the machine learning pipeline only considers aggregate counter data in addition to the given application's counter data, it doesn't make a difference to the pipeline whether other activity comes from many separate application VMs or a single VM (e.g., the hogger). By using

the hogger VM instead of other application VMs, we could focus on the MongoDB performance prediction while still testing the effects of other system activity.

The Hogger VM runs stress [4], a program that can put a variable amount of CPU, memory, disk I/O, and disk write load on a machine. We ran stress using the command `stress --timeout 600 --vm <n>`, which spawns n processes (each of which will run on its own core if possible) that repeatedly allocate and free 256 MB of memory for 600 seconds (10 minutes). We varied n linearly from 0 to 4, cycling back to 0 after passing 4 (when $n = 0$, the hogger VM sleeps for 600 seconds).

The hogger VM had the following specifications:

- Name: skye-hogger2
- 4 vCPUs
- 8 GB RAM
- 15 GB virtual disk
- Ubuntu Server 10.10 64-bit

3.6 Load generator (Rain)

We used Rain [3, 7], an open-source load generator, to replay the vmware.com traffic pattern to the MongoDB application. Rain decouples the traffic pattern from the output, providing a number of application adapters so the same pattern can be replayed to different applications and different patterns can be replayed to the same application in a controlled fashion. We used the provided MongoDB adapter, and customized the DiurnalScheduleCreator to use the vmware.com traffic pattern rather than the provided diurnal pattern. The MongoDB adapter creates a specified number of threads, each of which synchronously queries the database. We set the initial number of threads to 30. The number of threads is increased or decreased at each timestep (in our case 60 seconds) according to the traffic pattern. The threads issue 50% read queries and 50% update queries (i.e., read a single item or update the "value" field of a single item).

The VM running Rain had the following specifications:

- Name: skye-loadgen2
- 2 vCPUs
- 6 GB RAM

- 16 GB virtual disk
- Ubuntu Server 10.04 64-bit

3.7 Data collection

The MongoDB VM, hogger VM, and load generator VM (and the physical hosts they run on) comprise the experimental system that generates the data required by the machine learning pipeline. However, we also needed to collect the data from this system. To collect the ESX performance counter data, we set up a separate VM, running on the load generator's host so as not to interfere with the MongoDB and hogger VMs, that collected the counter values from the ESX host running the MongoDB and hogger VMs. It collected all counters available; for a complete list, see Appendix A. To collect the MongoDB response time data (with which we calculate our performance metric), we configured MongoDB to log every request using the `verbose=true` configuration parameter, and parsed the log to find all request timestamps, types, and response times.

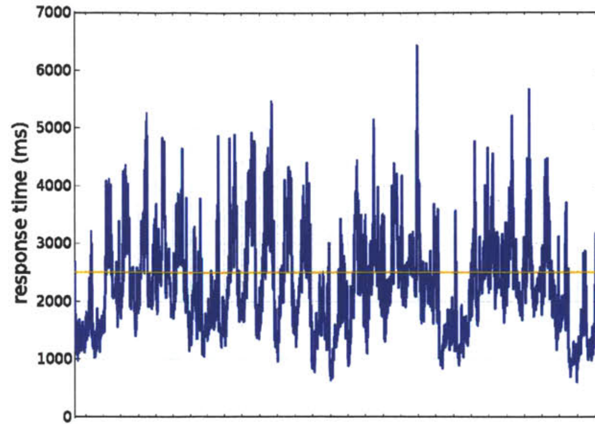
We also ran the mongostat program on the MongoDB VM, which collects various statistics (e.g., requests/second, queue lengths) every second. We stored its output as well as Rain's output, which contains information about how many threads are active at a given time, any errors, etc. This data was used in the machine learning pipeline but could be used to understand the relevant data and for debugging.

3.7.1 Timing

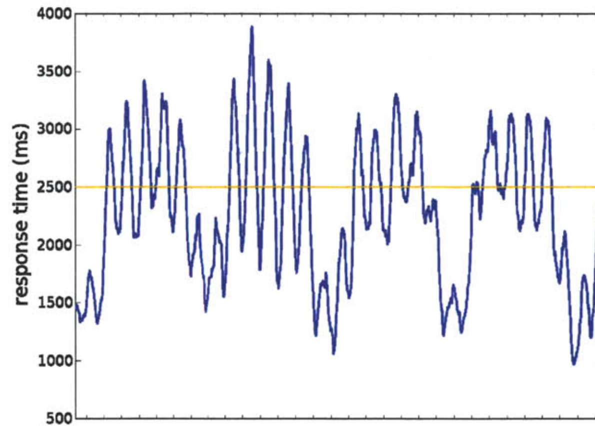
Because we were collecting data from several sources, it was very important that the clocks on each source be synchronized so the data could be correctly lined up via timestamp. We installed the ntpd dameon on every VM, which synchronizes the system clock using NTP (Network Time Protocol) [2].

3.8 SLO definition

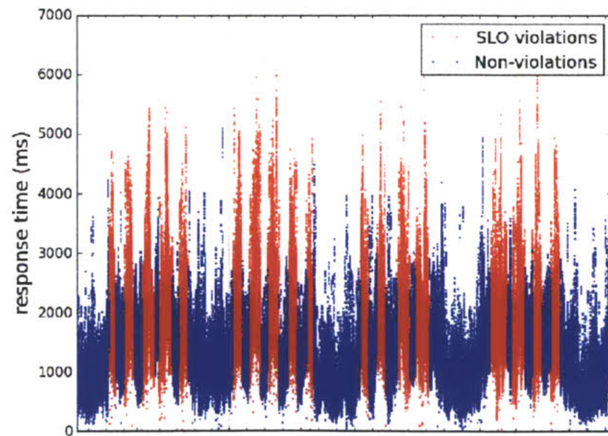
Once we had the raw response time data from MongoDB, we had to convert it to a binary performance metric to use as labels for the performance classifiers. We used a common service level



(a) MongoDB 99th-percentile latency. SLO violation threshold marked in orange.



(b) MongoDB 99th-percentile smoothed latency. SLO violation threshold marked in orange.



(c) MongoDB response times with SLO violations marked in red

Figure 3-5: Deriving SLO violations from response times. The 99th percentile latency is calculated every 60 seconds, the results of which are shown in (a). A moving average across 20 datapoints is taken to smooth the 99th-percentile latency curve, showin in (b). Any 60-second windows with a smoothed latency value above 3000ms is an SLO violation. (c) shows the original latency data with SLO violations marked.

objective (SLO) definition: at a given time interval (e.g., every minute, every ten minutes), some latency percentile is calculated (e.g., the 95th percentile, the 99.9^{th} percentile). If the latency value is above a certain threshold (e.g., 200 ms, 1 second), that time interval in considered to be an SLO violation. Otherwise it is a non-violation. We add the additional step of smoothing the latency percentile values using a moving average in order to avoid quick fluctuations between violations and non-violations. Table 3.1 and Table 3.2 explain these parameters in detail and the values we chose, respectively.

Illustrating this process, Figure 3-5a shows the 99th-percentile values of the response times shown in Figure 3-4. The violation threshold is shown in orange; points above this line are SLO violations while points below are non-violations. Without smoothing, the percentile values are very noisy, with many small fluctuations above and below the threshold.

Compare this to Figure 3-5b, which shows the same values and threshold but with a moving average applied over the 99th-percentile latency in order to smooth it. This gives us our final SLO violation definition, which is shown against the original raw latency data in Figure 3-5c. Here the points from Figure 3-4 are colored red if the smoothed percentile values are above the threshold, corresponding to SLO violations, and blue otherwise.

Table 3.1: SLO parameters

Parameter	Description
Percentile	Percentile of latency under consideration (e.g., 99th percentile)
Percentile window	The window size, in number of seconds, over which to calculate each SLO datapoint
Smoothing window	The window size, in number of datapoints, over which to perform a moving average of latency percentile
Threshold	Latency at percentile over which we consider that datapoint a violation

3.8.1 Response time clustering

We also tried a different technique for converting the raw response time data to a binary performance metric. Rather than manually choosing parameters to use with the algorithm above, we used

Table 3.2: SLO parameter values

Percentile	99
Threshold	3000 ms
Percentile window	60 seconds
Smoothing window	20 datapoints

clustering to automatically establish good and poor performance regions. First we clustered the response times into "fast" and "slow" responses. We then divided the response times into equally-sized windows (as we do in the SLO algorithm) and for every window computed the fraction of slow responses. We then clustered all the fractions into "violation" and "non-violations," which determined the final labeling of each window.

We ultimately chose to use the SLO algorithm because it allowed tuning of the labels (including tuning the labels to be very similar to those produced via the clustering method), was much faster to run, and is a more realistic metric – system administrators already apply SLOs in the fashion described above, and would likely be wary of a performance metric that offered no human interaction.

Chapter 4

Pipeline Implementation and Results

In this section we go into further detail on our implementation of the machine learning pipeline described in Chapter 2, as well as the results produced by our implementation.

4.1 Pipeline performance metrics

4.1.1 Prediction metrics

During the prediction stage, for a given ESX counter we denote its recorded time-series as y, with individual values y_i where $1 \leq i \leq n$ and n is the total number of datapoints. Likewise, the predicted time-series for that counter is denoted $\hat{y}$ with individual values $\hat{y}_i$, $1 \leq i \leq n$.

Root mean square error (RMSE)

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Mean absolute error (MAE) Unlike RMSE, MAE does not give extra weight to larger errors.

$$\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Pearson correlation coefficient (R) The linear correlation between the measured counter time-series and the predicted time-series. A correlation coefficient of 1 means complete positive

correlation, -1 complete negative correlation, and 0 no (linear) correlation.

$$\frac{\sum_{i=1}^{n}(y_i - \bar{y}_i)(\hat{y}_i - \bar{\hat{y}}_i)}{\sqrt{\sum_{i=1}^{n}(y_i - \bar{y}_i)^2 \sum_{i=1}^{n}(\hat{y}_i - \bar{\hat{y}}_i)^2}}$$

4.1.2 Classification metrics

For the classification task, SLO violations are considered positive examples and non-violations negative examples. We denote the set of SLO violation datapoints as y^+, also referred to as "actual positives", and non-violation datapoints as y^-, or "actual negatives". Similarly, predicted violations are denoted as $\hat{y}^+$, or "predicted positives", and predicted non-violations as $\hat{y}^-$, or "predicted negatives. $|y|$ is the number of datapoints in the set y, and $x \cap y$ is the intersection of x and y (i.e., points that are included in both x and y). Thus, $\hat{y}^+ \cap y^+$ is the set of correct positive predictions, or "true positives." Likewise, $\hat{y}^- \cap y^-$ is the set of correct negative predictions, or "true negatives."

Accuracy This may appear like a very natural metric to use, but is not very informative because it does not take into account the number of positive (violation) datapoints vs. the number of negative (non-violation) datapoints. For instance, if 90% of the datapoints are negative and a naive classifier predicts all points to be negative, it will yield an accuracy of 90%, which seems quite high but does not indicate that we are not correctly predicting a single positive. The following metrics better address this problem.

$$\frac{|\hat{y}^+ \cap y^+| + |\hat{y}^- \cap y^-|}{|y|} = \frac{\text{\# correct predictions}}{\text{\# datapoints total}}$$

Precision Also called positive predictive value (PPN). This represents what percentage of the predicted SLO violations are "true positives,", as opposed to "false alarms."

$$\frac{|\hat{y}^+ \cap y^+|}{|\hat{y}^+|} = \frac{\text{\# true positives}}{\text{\# predicted positives}}$$

Recall Also called sensitivity. This is an important metric for our application because it represents the percentage of SLO violations we successfully predict. We consider predicting all violations

more important than avoiding "false alarms," which is represented by precision.

$$\frac{|\hat{y}^+ \cap y^+|}{|y^+|} = \frac{\#\text{ true positives}}{\#\text{ actual positives}}$$

F-score Recall and precision reflect two different views of classification performance (e.g., if we classify everything as positive, we will have 100% recall but low precision). F-score is a combination of these two metrics.

$$2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{4.1}$$

Non-violation precision The negative analogue to precision.

$$\frac{|\hat{y}^- \cap y^-|}{|\hat{y}^-|} = \frac{\#\text{ true negatives}}{\#\text{ predicted negatives}}$$

Non-violation recall The negative analogue to recall.

$$\frac{|\hat{y}^- \cap y^-|}{|y^-|} = \frac{\#\text{ true negatives}}{\#\text{ actual negatives}}$$

In evaluating different methods and parameters, we attempted to optimize F-score; although recall is an important metric for us, it is still necessary to maintain some level of precision as well.

4.2 Training, validation and test sets

We divide counter and SLO violation time-series data into a training set, a validation set, and a test set based on timestamp: the first 50% of data is the training set, the following 25% the validation set, and the remaining 25% the test set. (See Chapter 3 for more information about the data, and in particular Section 3.8 for details on the SLO time-series data.) All models are trained using the training set, model parameters are evaluated and tuned based on results from the validation set, and the test set is used only as a final evaluation metric. It is important that the datapoints are partitioned by time, rather than randomly, since adjacent or nearby datapoints will be very similar, allowing the three sets to contain nearly identical datapoints. Figure 4-1 illustrates how we partitioned the data.

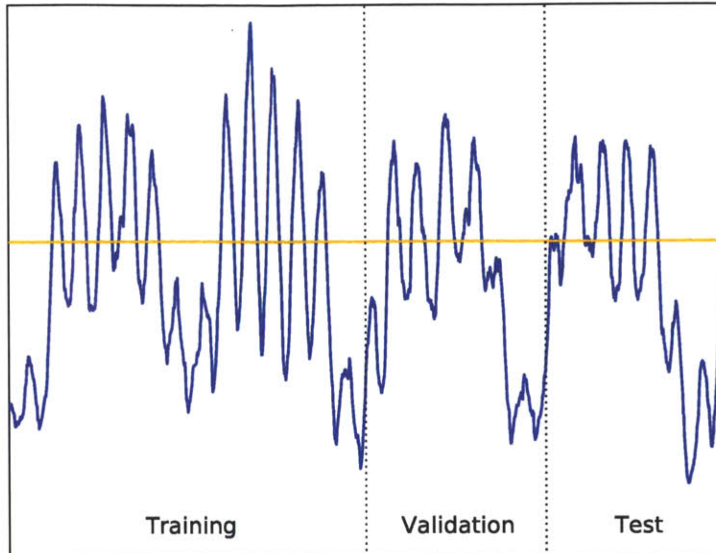


Figure 4-1: Training, validation, and test partitions. The partitions are shown over the smoothed 99th-percentile latency and violation threshold from Figure 3-5b.

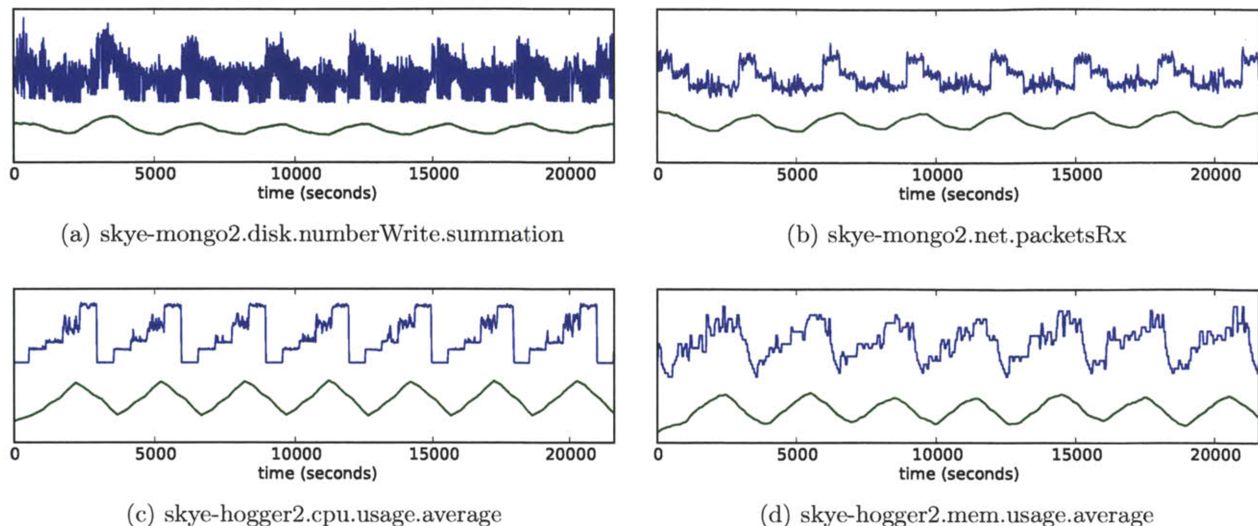
These partitions are only relevant during the classification stage, since this is the only stage that uses a model that requires training data. However, if modeling techniques were also applied to the prediction or aggregation stages, the same partitions should be used.

The three datasets are of the following sizes:

- Training set: 2119 counter datapoints
- Validation set: 1059 counter datapoints
- Test set: 1061 counter datapoints
- Total: 4296 counter datapoints

4.3 Preparing counters

Before using any of the performance counter data, the raw counter data is normalized so that each counter's time-series has zero mean and unit standard deviation, since we are using the counter data as input to SVMs. Additionally, a moving average is taken over each counter's time-series. Smoothing the counters in this way increases prediction accuracy since the counters change in a less erratic manner. Smoothing the counters also increases classification accuracy. Although information is lost in the smoothing, much of the variance within counter values is due to system noise, not variation in load or performance. Smoothing the counters reduces the amount of noise,



(a) skye-mongo2.disk.numberWrite.summation

(b) skye-mongo2.net.packetsRx

(c) skye-hogger2.cpu.usage.average

(d) skye-hogger2.mem.usage.average

Figure 4-2: Examples of smoothed vs. unsmoothed counters. For each counter, four hours worth of the original unsmoothed data is shown in blue, and the corresponding smoothed data shown below in green. The smoothing is performed by taking a moving average with a window size of 70 datapoints.

leaving a signal that varies on a slower timescale closer to that of the load and performance changes. An example of the unsmoothed vs. smoothed counter data is shown in Figure 4-2. In order to determine the window size of the moving average, we tried a range of sizes and picked the size that produced the highest F-score on the validation set. This technique yielded a window size of 70 datapoints, corresponding to 23.3 minutes of counter data.

4.4 Prediction

Initially, we used support vector regressions (SVRs) to perform counter prediction, training an SVR model for each counter. We used the SVR model implemented by mlpy [6], a Python machine learning library. mlpy provides a wrapper over LIBSVM's [11] SVR implementation.

Table 4.1: Prediction parameter values (see Table 2.2 for definitions)

Parameter	Value	Replay length	Traffic pattern length
Counter collection interval	20 seconds (minimum possible with vSphere)		
Historical lag	100 counter datapoints	33.33 minutes	16.67 hours
Predictive lag	1 counter datapoint	20 seconds	10 minutes

We found, however, that simple least-squares linear fits out-perform SVR models, as well as being much simpler to use since they do not need to be trained. For each counter, we consider only that counter's time-series, and given an input window of the counter's datapoints find the linear equation that minimizes the mean squared error. We then use this linear equation to compute the predicted future value. The size of the input window and how far ahead we predict are determined by the historical lag and predictive lag, respectively. The values we used for these parameters are shown in Table 4.1. Summary statistics aggregated across all counter results are shown in Table 4.2, and the same statistics aggregated across the counters used as input to classification (see Section 4.5.2) are shown in Table 4.3. Besides the minimum and maximum values, these tables show the three quartiles, which are the points that divide the dataset into four sets each containing an equal number of points based on value. For example, the 1st quartile is the value at which 25% of the points are smaller and 75% of the points are larger. When considering the root mean square error (RMSE) and mean absolute error (MAE), note that each counter's time-series is normalized to have zero mean and unit variance. When considering the Pearson correlation coefficient (R), note that unlike for RMSE and MAE, 1 is the best possible score and 0 the worst. The individual results of all counters are shown in Appendix B.

Table 4.2: Linear fit prediction results (all features).

	Min	1st quartile	2nd quartile (median)	3rd quartile	Max
RMSE	0.00	0.11	0.17	0.29	0.68
MAE	0.00	0.05	0.13	0.20	0.61
R	0.00	0.00	0.66	0.83	1.00

Table 4.3: Linear fit prediction results (selected features)

	Min	1st quartile	2nd quartile (median)	3rd quartile	Max
RMSE	0.06	0.15	0.18	0.48	0.68
MAE	0.02	0.10	0.15	0.42	0.61
R	0.57	0.64	0.66	0.68	0.89

4.5 Classification

For the performance classification stage of the machine learning pipeline, we use support vector machines (SVMs). SVMs, in their most common formulation, are a statistical method for binary classification, i.e., classifying datapoints as belonging to one of two groups (SVMs can also be used for similar tasks such as single-class classification and regression). Given a training set of labeled datapoints, we find the hyperplane that separates one group from the other and maximizes the distance to the nearest point (note that there will be at least one point from each group at this distance). Once we have found this hyperplane, we classify new datapoints based on what side of the hyperplane they lie. For further details, [12] is a good introduction to SVMs.

Each VM's performance is modeled using an SVM. We use the SVM model provided by mlpy [6], which is a wrapper over LIBSVM's [11] SVM implementation. The rest of this section details the techniques we used to improve classification results. Since our experimental system includes only a single application VM, we only needed to train a single SVM, but these techniques could be independently applied to further application VMs.

4.5.1 SVM parameter grid search

We use an `svm_type` of `c_svc` (i.e., C-Support Vector Classification, see [11]), which has a tunable parameter C that determines the penalty of making mistakes in the training data, and a `kernel_type` of `rbf` (i.e., Gaussian Radial Basis Function), which has a parameter γ. We chose to use a Gaussian RBF kernel because it is a flexible general-purpose kernel that can fit any dataset (although correctly labeling every training point can lead to overfitting, which is why we also use the slack parameter C). The kernel's γ parameter determines how tight the boundary is drawn around positive datapoints. We tune these parameters by performing a grid search over a wide range of values, and evaluate the results of every parameter combination using the F-score of the validation set. Table 4.4 shows the final parameter values.

Table 4.4: SVM parameter values
(see Table 2.3 for definitions)

C	100
γ	0.02

4.5.2 Feature selection

Although every counter is available as input to the performance classifiers, feature selection (i.e., choosing a subset of available inputs, or features) can improve results. To perform feature selection, we iterated through every counter $c \in C$, where C is the set of all counters, trained an SVM using counters $C \setminus c$, and compared the resulting F-score of the validation set to the F-score of using all counters C. Our final set of counters was every counter that lowered the F-score when omitted. Table 4.5 lists and describes these counters.

4.5.3 Weight violations

The `c_svc` SVM has an additional weight parameter that can be applied to either class. The weight value for a given class is multiplied with the C value for that class, allowing for one class to be more heavily weighted than another and to be favored by the classifier. Since we are more concerned with correctly predicting SLO violations and avoiding false negatives (i.e., missing SLO violations) than correctly predicting non-violations and avoiding false positives (e.g., false alarms), it intuitively makes sense to weight the positive class (corresponding to classifying performance as an SLO violation) more than the negative class. In practice, though, this did not improve our results.

4.6 Pipeline results

Table 4.6 shows the numerical performance metrics of running our pipeline implementation on the training, validation, and test sets. Figure 4-3 shows the same results graphically.

As expected, the pipeline performs best on the training data, predicting only a few datapoints incorrectly. More surprisingly, the validation set results are considerably worse than the training set results. We would expect the test results to be worse or at most similar to the validation results, since the various pipeline parameters were chosen to optimize the validation results, and the assumption is that the validation and test data are somewhat similar. However, based on the results it appears that the time-series data varies considerably over time, causing the test and validation sets to be quite different. More work is needed to identify what these differences are, and how to make the pipeline more robust to different types of behavior.

Table 4.5: Counters used as classifier features. Descriptions are from [5].

Counter name	Description
cpu.usage.average	Amount of actively used virtual CPU, as a percentage of total available CPU.
cpu.usagemhz.average	Amount of actively used virtual CPU, as measured in megahertz.
datastore.numberReadAveraged.average	Average number of read commands issued per second to the datastore.
datastore.read.average	Rate of reading data from the datastore.
disk.numberReadAveraged.average	Average number of read commands issued per second to the datastore.
disk.numberWrite.summation	Number of disk writes.
disk.read.average	Average number of kilobytes read from the disk each second.
mem.active.average	Amount of memory that is actively used, as estimated by VMkernel based on recently touched memory pages.
mem.activewrite.average	Amount of memory actively being written to by the virtual machine.
mem.usage.average	Memory usage as percentage of total configured memory, expressed as a hundredth of a percent ($1 = 0.01\%$).
net.packetsRx.summation	Number of packets received.
net.transmitted.average	Average rate at which data was transmitted. This represents the bandwidth of the network.
net.usage.average	Network utilization (combined transmit- and receive-rates).
rescpu.actav1.latest	The average percentage of time the VM is running or ready to run over a 1-minute interval.
rescpu.actav5.latest	The average percentage of time the VM is running or ready to run over a 5-minute interval.
rescpu.actav15.latest	The average percentage of time the VM is running or ready to run over a 15-minute interval.
rescpu.actpk15.latest	The maximum percentage of time the VM is running or ready to run over a 15-minute interval.
rescpu.runav15.latest	The average amount of CPU utilization over a 15-minute interval.
rescpu.runpk1.latest	The highest amount of CPU utilization over a 1-minute interval.
rescpu.runpk5.latest	The highest amount of CPU utilization over a 5-minute interval.
rescpu.runpk15.latest	The highest amount of CPU utilization over a 15-minute interval.

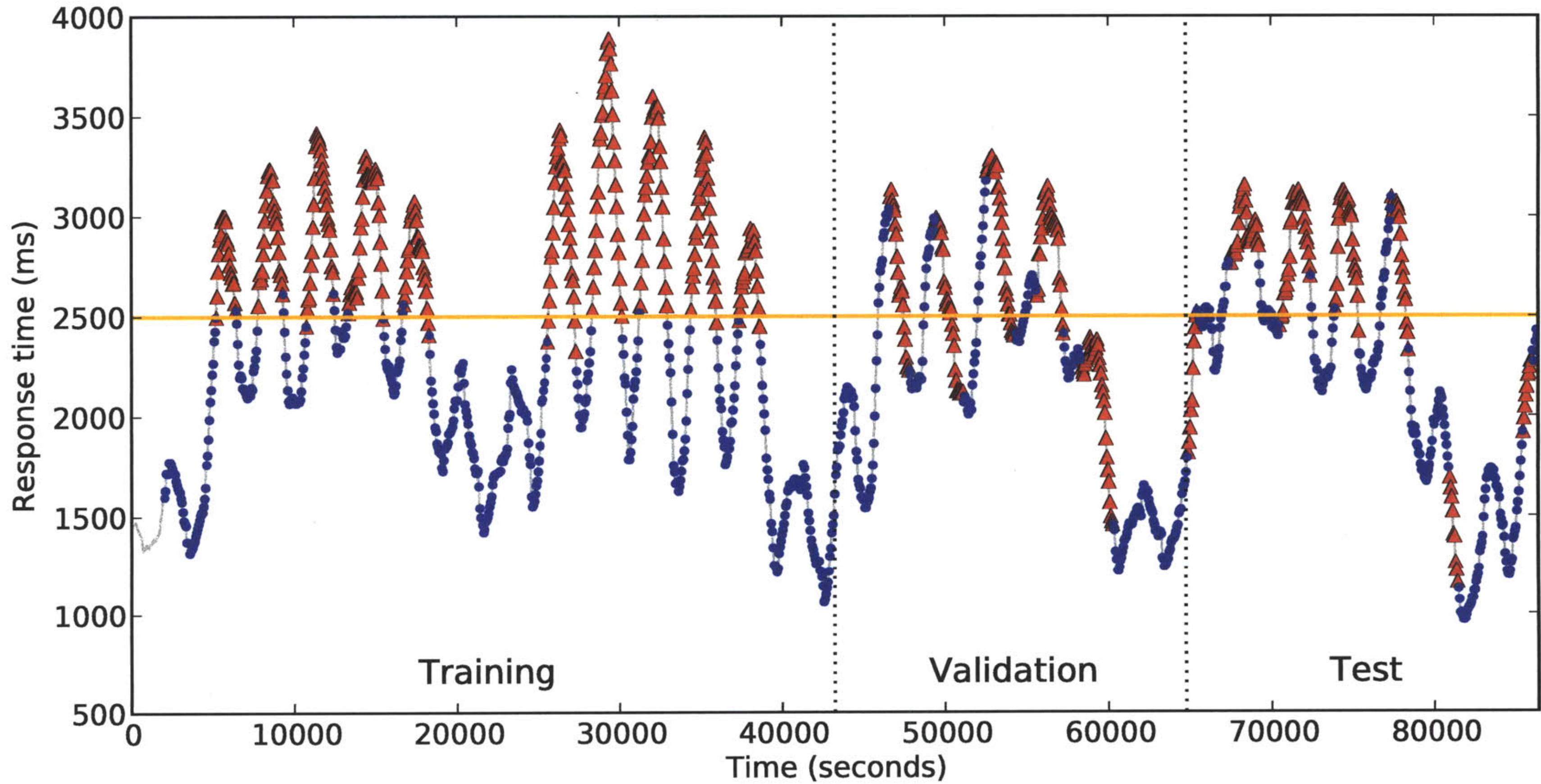


Figure 4-3: Final performance prediction results. SLO violation predictions are shown as red triangles and non-violations as blue circles. The points are drawn over the smoothed 99th-percentile latency, with the violation threshold shown in orange, meaning correct violation predictions appear above the threshold and correct non-violation predictions appear below the threshold (and vice versa for incorrect predictions).

Table 4.6: Final performance prediction results. All metrics except for F-score are shown as percentages. The "Predicted % positive/negative" entries refer to the percentage of datapoints predicted to be positive (violations) and negative (non-violations), for comparison with the "Actual % positive/negative" entries.

(a) Training set results

Metric	Value
Precision	95.5
Recall	96.0
F-score	.957
Non-violation precision	97.4
Non-violation recall	97.1
Accuracy	96.6
Actual % positive	39.2
Actual % negative	60.8
Predicted % positive	39.4
Predicted % negative	60.6

(b) Validation set results

Metric	Value
Precision	60.6
Recall	68.8
F-score	.645
Non-violation precision	82.0
Non-violation recall	76.1
Accuracy	73.6
Actual % positive	34.8
Actual % negative	65.2
Predicted % positive	39.6
Predicted % negative	60.4

(c) Test set results

Metric	Value
Precision	77.8
Recall	89.1
F-score	.831
Non-violation precision	93.2
Non-violation recall	85.5
Accuracy	86.8
Actual % positive	36.3
Actual % negative	63.7
Predicted % positive	41.6
Predicted % negative	58.4

Chapter 5

Future Work

5.1 Experiment configurations

The experimental system described in Chapter 3 is just one of many possible configurations, and in hindsight was a very complicated choice that made it difficult to interpret the results and identify what needs to be improved. There are two main parameters of the experimental system we can vary: the application(s) running on the test host, and the traffic pattern(s) determining the load on the system. Some choices for these parameters, including the ones we chose, are summarized Table 5.1, with simpler parameters towards the top left and more complex parameters towards the bottom right.

Table 5.1: Potential experimental systems

	linear	vmware.com, realtime	vmware.com, sped up
Mongo			
Mongo + hogger			You are here!
Mongo + Mongo			

Along the application axis, we chose to run a MongoDB instance as our test application, with a "hogger" VM simulating resource usage from other applications on the host. In a real system, there would be multiple applications running and our pipeline would monitor each of them, so running two or more MongoDB instances (or other test applications) is a natural extension to our experiments. However, in order to better understand the relationships between load, ESX counters, and latency, we would like to simplify the system rather than increase its complexity. Thus, we

would first like to try running the MongoDB test application alone on the host in order to isolate its behavior from that of other VMs.

Along the traffic pattern axis, we ran the vmware.com pattern with each hour's datapoint replayed for two minutes. Speeding up the traffic pattern allowed us to collect the month's worth of data in a reasonable amount of time, but also meant that the load to the system fluctuated much more rapidly than it would in a real system. Since we obviously can't increase the VM's processing speed to match the increase in load change speed, the rapid changes in load lead to noisier measurements than gradual changes would. Additionally, such dramatic time scaling doesn't lend itself well to all time intervals. For instance, we set our predictive lag to twenty seconds, which scaled back to the traffic pattern's original timing is ten minutes; however, if we run the traffic pattern in realtime and actually predict ten minutes head, it seems unlikely that the results will be similar since system metrics can change much more in ten minutes than in twenty seconds.

To better learn how noisy the latency and ESX counters are in relation to load, we would like to simplify the traffic pattern even further to a gradual linear increase before returning to more realistic traffic data.

5.2 Refine pipeline stages

Both the prediction and classification stages of the pipeline can be further studied, either concurrently or independently. Any improvements to either stage should yield better overall pipeline results.

Our current prediction scheme is very simple. We would like to explore more sophisticated prediction methods, such as auto-regressive correlation. As for classification, we have only considered SVMs so far and have not tried different types of classifiers (e.g., models such as Bayesian classifiers that provide better transparency into why a datapoint is classified the way it is). Different feature selection approaches should also be explored, both in selecting our current counter features and using the counter data to create new features. For example, we could compute the variance of each counter for a given window and include those values as input to the classifier.

5.3 Online learning

In order for our pipeline to be deployed on a real system, it must be able to adapt to changes in workload and performance characteristics. For example, software updates may change the relationship between CPU usage (or any other resource) and latency. If the classifier for this VM cannot adapt to this change, it will continue to classify performance based on the old relationship and its accuracy will suffer. Right now we only use offline learning, where the classifiers are trained once on a specified training set. Once the models are trained, they do not change. In order to adapt to changes in performance characteristics, the models must change over time, which is achieved via online learning. A model that employs online learning is trained iteratively, so after learning the true label of a datapoint that it previously classified, the model is updated based on that result. This would allow the classifiers to "learn" from the incoming datapoints that were previously predicted.

Another option is to save incoming data for a certain amount of time, and use this batched data to periodically train a new classifier completely. Retraining the model can happen at a set interval or when the error of the classifier reaches a certain threshold. This allows us to continue using offline learning techniques while allowing the model to change over time. However, it means the current model at a given time has no knowledge of past data that has been discarded.

This discussion has been centered around the classification stage because it is currently the only trained model we use. However, if a similar model is applied to the prediction stage, the same reasoning applies: the prediction model would need to adapt based on changes in the incoming traffic pattern.

5.4 End-to-end system

The ultimate goal of this research is to provide not just performance prediction, but automatic performance management. We would like to develop an end-to-end system that uses the prediction pipeline described here as input to a controller that provisions VM resources, initiates VM migration to different hosts, or both in order to maintain good application performance. Section 6.2 of the Related Work chapter overviews some systems that perform this kind of management.

Chapter 6

Related Work

6.1 Performance Modeling and Anomaly Detection

Other work focuses on monitoring, rather than predicting, performance. These approaches could be applied to the classification stage of our pipeline.

Zhang et al. [21] use ensembles of Tree-Augmented Bayesian Networks (TANs) to model application performance based on system metrics such as CPU, memory, network, and disk usage. They found that TANs are able to accurately determine if an application is meeting its SLO or not using a small number of metrics. TANs also indicate which metrics most strongly influenced the outcome, revealing which system metrics are correlated with SLO violations. A single TAN model is sufficient for a steady-state workload, but evolving workloads influenced by, e.g., unexpected traffic surges or software changes, require a different solution. Zhang et al. found ensembles of TANs to be more effective than either a single monolithic TAN or a single TAN retrained over time. Under the ensemble model, new models are periodically trained from recent data, and if the new model's accuracy is better than the existing ensemble's, it is added to the ensemble. Rather than combine the results of all models in the ensemble, a winner-takes-all approach is implemented where the results of the model with the best Brier score, which measures accuracy and confidence, computed over a short interval of past data are used.

Nguyen et al. [17] develop the Propagation-aware Anomaly Localization (PAL) system to detect performance anomalies due to software bugs in distributed systems and to pinpoint the component in which the problem originated. PAL is application- and topology-independent, meaning it uses

only system-level metrics and assumes no prior knowledge of the applications running. After detecting an SLO violation, PAL considers the time-series preceding the violation of each system metric of each component. After smoothing the time series to remove small spurious fluctuations, PAL uses cumulative sum charts and bootstrapping to identify change points in each time series. PAL then locates the change point corresponding to the onset of the anomalous behavior in each time series by comparing the mean of the time series before and after each change point and finding the earliest dramatic change. For each component, the earliest change point among all the metrics' time series corresponds to the onset of that component's anomalous behavior. Finally, the components are sorted by their anomaly onset times to determine where the problem started and how it propagated through the system.

6.1.1 Anomaly prediction

Tan et al. [20] present ALERT, a system for predicting anomalies caused by faulty applications, e.g., a program with a memory leak that will cause its performance to eventually degrade. ALERT uses decision trees to assign system metric measurements to a normal, alert, or anomaly state, where the alert state corresponds to the time period preceding the onset of an anomaly. ALERT specifically avoids triggering on "execution context" changes, i.e., when the underlying hosting infrastructure running an application is altered, such as when less CPU is allocated (this contrasts with most other research including our project, which focuses solely on hosting-related issues). ALERT uses a clustering algorithm to partition and group the input data stream into multiple execution contexts, then trains a model for each context. A Bayesian model is used to determine the current context given current system measurements. Tan et al. chose this approach over a single evolving model in order to decrease training overhead and give robust predictions even given rapidly changing contexts. Incorrect predictions can trigger new context definitions and inactive contexts are removed, allowing ALERT's context-switching model to adapt to evolving conditions.

6.2 Resource Scaling

Other work describes systems for automatically adjusting VM resource limits or placement. These works could be useful both for refining our techniques and for creating a similar end-to-end system.

6.2.1 Physical servers

Bodík et al. [8] combine machine learning and control theory to automatically add or remove physical servers from a web application with the goal of providing a practical, robust controller that can react to changes in the application and its environment. They train a performance model which, given input of workload and number of active servers, estimates the fraction of requests slower than a given SLA threshold. They then perform a simple workload prediction to input to the model. By monitoring the residuals of model, they determine when the model no longer accurately predicts performance, at which point a new model is trained using the latest production data. Based on the performance model, servers are added or removed by the controllers. The controller's parameters are determined by simulating the application's performance under different control policies.

In [9], Bodík et al. present a method to effectively gather the production data needed to train an online performance model such as that discussed in [8]. They note that in order to train an accurate model, data from a variety of performance regimes must be collected, including near the application's maximum capacity, and ideally the data is collected quickly. However, this is at odds with preserving the application's SLA. The authors propose an exploration policy in which machines are added or removed in order to reach a frequently-changing latency target, which is randomly chosen but always beneath a certain safety threshold. If the safety threshold is exceeded, another server is immediately added. This is achieved with "hot standby" machines that run the application without receiving any requests, allowing them to be quickly added. They also quickly estimate a linear capacity model to prevent pushing the application past capacity. The system switches from the exploration policy to model-based control when it determines that the model is stable at the current workload.

6.2.2 VMs

When dealing with VMs, rather than physical machines, resource scaling systems can adjust applications at a much finer-grained level than turning entire machines on and off by adjusting the amount of physical resources (e.g., CPU cycles, memory) available to each VM. Padala et al. [18] take such an approach with the AutoControl system, which allocates resources over short time

intervals (e.g., seconds) such that all the virtualized applications on a physical host meet their SLOs, or gives preference to high-priority applications if satisfying all SLOs is not possible with the available resources. For each application, AutoControl periodically measures performance and uses this to update a linear model approximating the relationship between the application's resource allocations and performance. Using this model, AutoControl determines the resources needed to meet the SLO. Finally, the resources requested by each application are distributed based how many total resources are available, how much each application requested, and the applications' relative priorities. The results indicate that AutoControl better maintains application SLOs than either a static resource allocation or allowing unlimited access to resources.

Gong et al. [15] extend the idea of dynamic resource allocation among VMs to include resource usage prediction, so that necessary resources can be allocated ahead of time. The PRedictive Elastic ReSource Scaling (PRESS) system identifies repeated patterns in resource usage by finding "signatures" in historical usage data using a fast Fourier transform (FFT), and then uses dynamic time warping (DTW) to line up the signature with incoming data and extend the signature forward in time in order to predict future resource use. If no repeating pattern is found, PRESS uses a Markov chain for short-term prediction. This process is repeated in order to capture changes in resource usage patterns over time.

Migration

Besides adjusting physical resource allocations, VM resource scaling can also be achieved via live migrations, in which a running VM is moved to a different host. This can be used to decrease load on a overloaded host by reducing the number of VMs on it, or to consolidate VMs onto fewer hosts in order to save resources and money. Gong et al. apply their signature approach to migration in [14], which introduces the Pattern-Driven Application Consolidation (PAC) system. PAC finds signatures for each running VM using the same FFT-based technique as [15]. Additionally, PAC finds the signature of each host's unused, or residual, resources. By comparing the VMs' signatures to the hosts' residual signatures, PAC can match a VM to an appropriate host. This matching is done at scale using a variation of DTW that includes pre-filtering to avoid running the expensive DTW algorithm on all signature pairs. PAC can migrate a VM in response to an overloaded host or a change in the VM's signature pattern. PAC also performs periodic global VM consolidation

in which it attempts to find an optimal placement for all running VMs based on their signatures.

Shen et al. continue this work in [19] with the CloudScale system, combining the resource allocation and migration techniques of [15] and [14], as well as adding CPU voltage and frequency scaling to save energy from unused resources. CloudScale uses the same signature-finding algorithm described above to make short-term predictions of VMs' resource usage and adjust VMs' resource caps similarly to [15]. In addition to predicting short-term resource usage, CloudScale also predicts scaling conflicts, which is when there aren't enough resources on the host to accommodate all the predicted resource needs. The scaling prediction algorithm is the same as the usage prediction algorithm except it operates over longer time periods. By comparing the sum of the VMs' predicted usage over time and the total capacity of the host, the scaling prediction algorithm can estimate when a conflict will occur, how long it will last, and the severity of the conflict.

When a scaling conflict is predicted, CloudScale can either under-provision VMs until the conflict ends (local conflict handling) or migrate a VM to a different host (migration-based conflict handling). In general, it is best to use local conflict handling when the predicted conflict is short and small and migration-based conflict handling when the conflict is sustained and severe, since migrating a VM takes a relatively long time and uses even further resources during this time. When performing local conflict handling, CloudScale must determine how to distribute the available resources among VMs. It either treats all VMs uniformly and allocates resources proportional to demand, or can allocate resources to high-priority application VMs first before distributing the rest among lower-priority VMs. The SLO penalty incurred by each VM m_i is estimated by $\sum_{t=t_1}^{t_2} RP_i \cdot e_{i,t}$, where RP_i is the Resources under-provisioning Penalty for m_i and e_{i_t} is the under-estimation error at time t, and t_1 and t_2 are the start and end time of the conflict. Q_{RP} is the sum of all VMs' SLO penalties.

When performing migration-based conflict handling, CloudScale must determine which VM to evict from the soon-to-be-overloaded host, with the goal of migrating as few VMs as possible and minimizing SLO violations. For each VM m_i, a normalized SLO penalty metric is defined as $Z_i = MP_i \cdot T_i/(w_1 \cdot cpu_i + w_2 \cdot mem_i)$, where MP_i is the Migration Penalty, T_i is the time it takes to migrate m_i (computed based on the average memory usage), cpu_i and mem_i are the normalized CPU and memory use of m_i, and w_1 and w_2 are weights that determine the relative importance of CPU and memory, the two resources managed by the CloudScale implementation. CloudScale

migrates the VM with the smallest Z until there are enough resources to resolve the conflict. Q_M is the sum of all Z_i. CloudScale chooses whether to use local or migration-based conflict resolution by comparing Q_{RP} to Q_M.

The CloudScale system is similar to our work in that it uses resource usage prediction to achieve application SLOs. However, our approach for leveraging prediction to prevent SLO violations is fundamentally different: rather than compare predicted resource usage to available capacity and then use estimates of SLO penalty to decide if and what action is necessary, we predict many fine-grained system metrics and train models to convert the resource predictions directly into predictions of SLO violation. Thus, we are predicting SLO violations directly so appropriate action can be taken, rather than using penalties based on SLOs as a decision-making heuristic.

Chapter 7

Conclusions

We have designed and implemented a machine learning pipeline for predicting future performance of applications running on virtual machines based on system-level performance counters. The pipeline is divided into three stages: prediction, which predicts future VM counter values based on current time-series data; aggregation, which aggregates the predicted VM metrics into a single set of global metrics; and finally classification, which for each VM classifies its performance as an SLO violation or non-violation based on the predicted VM counters and the predicted global state. We have implemented prediction using a simple linear fit algorithm, aggregation as a sum, and classification via SVMs, meaning classification requires an initial training dataset. However, the implementation of any stage can be changed without dramatic changes to any other stage, allowing easier refinement of the stages. The pipeline is designed to allow for dynamic VM placement changes (or VMs to be powered on or off) without requiring updates to the algorithms.

We have also created an experimental system for testing the pipeline and evaluating its performance. Our experimental system consists of a MongoDB instance running on its own VM, which is our test application; a "hogger" VM running on the same host that consumes CPU and memory resources to simulate the effect of other applications running on the host; and a load generator running on a separate VM that sends traffic to the MongoDB instance. Since we are making predictions it is important to create realistic traffic, so we used traffic data from vmware.com as the basis for our traffic pattern. We defined an SLO violation based on latency as our performance metric for the MongoDB test application.

The results from running the pipeline on data collected from the experimental system are

promising, but indicate that there is more work to be done before the pipeline is ready for a production system. The recall of the test set is 89.1%, meaning we correctly predict just under nine out of ten SLO violations. We consider this a good start for an automatic system, but will need to be much higher for a production system. Additionally, the scores of the validation set are considerably lower than those of the test set, suggesting that small changes in the input data yield very different results. Given this observation, we believe that future work should first be directed at collecting simpler datasets so as to better isolate variations in pipeline performance.

Appendix A

ESX performance counters

The following are the VM-level ESX performance counters available to us from our vSphere 4.1 setup (79 in total):

```
cpu.ready.summation
cpu.swapwait.summation
cpu.system.summation
cpu.usage.average
cpu.usagemhz.average
cpu.used.summation
cpu.wait.summation
datastore.numberReadAveraged.average
datastore.numberWriteAveraged.average
datastore.read.average
datastore.totalReadLatency.average
datastore.totalWriteLatency.average
datastore.write.average
disk.busResets.summation
disk.commands.summation
disk.commandsAborted.summation
disk.commandsAveraged.average
disk.maxTotalLatency.latest
disk.numberRead.summation
disk.numberReadAveraged.average
disk.numberWrite.summation
disk.numberWriteAveraged.average
disk.read.average
disk.usage.average
disk.write.average
mem.active.average
mem.activewrite.average
mem.compressed.average
mem.compressionRate.average
mem.consumed.average
mem.decompressionRate.average
mem.granted.average
mem.overhead.average
mem.overheadMax.average
```

mem.shared.average

mem.swapin.average

mem.swapinRate.average

mem.swapout.average

mem.swapoutRate.average

mem.swapped.average

mem.swaptarget.average

mem.usage.average

mem.vmmemctl.average

mem.vmmemctltarget.average

mem.zero.average

mem.zipSaved.latest

mem.zipped.latest

net.packetsRx.summation

net.packetsTx.summation

net.received.average

net.transmitted.average

net.usage.average

power.energy.summation

power.power.average

rescpu.actav1.latest

rescpu.actav15.latest

rescpu.actav5.latest

rescpu.actpk1.latest

rescpu.actpk15.latest

rescpu.actpk5.latest

rescpu.maxLimited1.latest

rescpu.maxLimited15.latest

rescpu.maxLimited5.latest

rescpu.runav1.latest

rescpu.runav15.latest

rescpu.runav5.latest

rescpu.runpk1.latest

rescpu.runpk15.latest

rescpu.runpk5.latest

rescpu.sampleCount.latest

rescpu.samplePeriod.latest

sys.heartbeat.summation

sys.uptime.latest

virtualDisk.numberReadAveraged.average

virtualDisk.numberWriteAveraged.average

virtualDisk.read.average

virtualDisk.totalReadLatency.average

virtualDisk.totalWriteLatency.average

virtualDisk.write.average

Appendix B

Counter Prediction Results

Table B.1 shows the root mean squared error (RMSE), mean absolute error (MAE), and correlation coefficient (R) of the predictions for each VM counter. Figure B-1 plots the recorded counter values vs. the predicted values and the error residuals. These predictions were performed using a least-squares linear fit as described in Section 4.4. Note that counters with constant values are omitted, and that each counter's time-series is normalized to have zero mean and unit variance. When considering the Pearson correlation coefficient (R), note that unlike for RMSE and MAE, 1 is the best possible score and 0 the worst (none of our predictions were so off as to produce negative correlations).

Table B.1: Counter prediction results

Counter	RMSE	MAE	R
skye-hogger2.cpu.ready.summation	0.620	0.564	0.641
skye-hogger2.cpu.system.summation	0.199	0.079	0.894
skye-hogger2.cpu.usage.average	0.424	0.394	0.635
skye-hogger2.cpu.usagemhz.average	0.424	0.394	0.635
skye-hogger2.cpu.used.summation	0.306	0.266	0.711
skye-hogger2.cpu.wait.summation	0.489	0.417	0.675
skye-hogger2.datastore.numberWriteAveraged.average	0.107	0.021	0.852
skye-hogger2.datastore.totalWriteLatency.average	0.132	0.080	0.818
skye-hogger2.datastore.write.average	0.094	0.032	0.892

Table B.1: Counter prediction results (continued)

Counter	RMSE	MAE	R
skye-hogger2.disk.commands.summation	0.164	0.072	0.877
skye-hogger2.disk.commandsAveraged.average	0.107	0.021	0.852
skye-hogger2.disk.maxTotalLatency.latest	0.132	0.080	0.819
skye-hogger2.disk.numberWrite.summation	0.164	0.072	0.877
skye-hogger2.disk.numberWriteAveraged.average	0.107	0.021	0.852
skye-hogger2.disk.usage.average	0.094	0.032	0.892
skye-hogger2.disk.write.average	0.094	0.032	0.892
skye-hogger2.mem.active.average	0.574	0.514	0.657
skye-hogger2.mem.activewrite.average	0.575	0.515	0.657
skye-hogger2.mem.overhead.average	0.411	0.328	0.879
skye-hogger2.mem.usage.average	0.574	0.514	0.657
skye-hogger2.net.packetsRx.summation	0.177	0.099	0.856
skye-hogger2.net.packetsTx.summation	0.485	0.116	0.885
skye-hogger2.net.received.average	0.126	0.063	0.756
skye-hogger2.net.transmitted.average	0.509	0.096	0.871
skye-hogger2.net.usage.average	0.425	0.113	0.888
skye-hogger2.rescpu.actav1.latest	0.582	0.536	0.639
skye-hogger2.rescpu.actav15.latest	0.677	0.610	0.642
skye-hogger2.rescpu.actav5.latest	0.615	0.564	0.639
skye-hogger2.rescpu.actpk1.latest	0.575	0.530	0.639
skye-hogger2.rescpu.actpk15.latest	0.596	0.526	0.639
skye-hogger2.rescpu.actpk5.latest	0.579	0.531	0.638
skye-hogger2.rescpu.runav1.latest	0.429	0.399	0.635
skye-hogger2.rescpu.runav15.latest	0.468	0.423	0.639
skye-hogger2.rescpu.runav5.latest	0.449	0.415	0.635
skye-hogger2.rescpu.runpk1.latest	0.461	0.425	0.636
skye-hogger2.rescpu.runpk15.latest	0.537	0.480	0.636

Table B.1: Counter prediction results (continued)

Counter	RMSE	MAE	R
skye-hogger2.rescpu.runpk5.latest	0.489	0.452	0.635
skye-hogger2.sys.heartbeat.summation	0.025	0.021	0.662
skye-hogger2.sys.uptime.latest	0.000	0.000	1.000
skye-hogger2.virtualDisk.numberWriteAveraged.average	0.107	0.021	0.852
skye-hogger2.virtualDisk.totalWriteLatency.average	0.132	0.080	0.819
skye-hogger2.virtualDisk.write.average	0.094	0.032	0.892
skye-mongo2.cpu.ready.summation	0.536	0.495	0.638
skye-mongo2.cpu.system.summation	0.107	0.093	0.671
skye-mongo2.cpu.usage.average	0.183	0.155	0.656
skye-mongo2.cpu.usagemhz.average	0.183	0.155	0.657
skye-mongo2.cpu.used.summation	0.177	0.149	0.653
skye-mongo2.cpu.wait.summation	0.249	0.227	0.640
skye-mongo2.datastore.numberReadAveraged.average	0.071	0.021	0.804
skye-mongo2.datastore.numberWriteAveraged.average	0.173	0.149	0.664
skye-mongo2.datastore.read.average	0.061	0.025	0.862
skye-mongo2.datastore.totalReadLatency.average	0.123	0.096	0.753
skye-mongo2.datastore.totalWriteLatency.average	0.144	0.124	0.663
skye-mongo2.datastore.write.average	0.218	0.188	0.648
skye-mongo2.disk.commands.summation	0.172	0.149	0.663
skye-mongo2.disk.commandsAveraged.average	0.172	0.148	0.663
skye-mongo2.disk.maxTotalLatency.latest	0.146	0.126	0.656
skye-mongo2.disk.numberRead.summation	0.072	0.028	0.805
skye-mongo2.disk.numberReadAveraged.average	0.071	0.021	0.804
skye-mongo2.disk.numberWrite.summation	0.173	0.150	0.664
skye-mongo2.disk.numberWriteAveraged.average	0.173	0.149	0.664
skye-mongo2.disk.read.average	0.061	0.025	0.862
skye-mongo2.disk.usage.average	0.217	0.188	0.648

Table B.1: Counter prediction results (continued)

Counter	RMSE	MAE	R
skye-mongo2.disk.write.average	0.218	0.188	0.648
skye-mongo2.mem.active.average	0.174	0.097	0.618
skye-mongo2.mem.activewrite.average	0.182	0.103	0.567
skye-mongo2.mem.consumed.average	0.002	0.001	1.000
skye-mongo2.mem.granted.average	0.002	0.001	1.000
skye-mongo2.mem.overhead.average	0.085	0.068	0.996
skye-mongo2.mem.shared.average	0.064	0.006	0.594
skye-mongo2.mem.usage.average	0.174	0.097	0.618
skye-mongo2.mem.zero.average	0.064	0.005	0.000
skye-mongo2.net.packetsRx.summation	0.231	0.200	0.648
skye-mongo2.net.packetsTx.summation	0.237	0.206	0.647
skye-mongo2.net.received.average	0.248	0.215	0.647
skye-mongo2.net.transmitted.average	0.177	0.154	0.646
skye-mongo2.net.usage.average	0.224	0.194	0.645
skye-mongo2.rescpu.actav1.latest	0.145	0.129	0.669
skye-mongo2.rescpu.actav15.latest	0.126	0.112	0.689
skye-mongo2.rescpu.actav5.latest	0.142	0.127	0.675
skye-mongo2.rescpu.actpk1.latest	0.196	0.175	0.666
skye-mongo2.rescpu.actpk15.latest	0.149	0.131	0.686
skye-mongo2.rescpu.actpk5.latest	0.180	0.161	0.666
skye-mongo2.rescpu.runav1.latest	0.184	0.156	0.656
skye-mongo2.rescpu.runav15.latest	0.163	0.143	0.640
skye-mongo2.rescpu.runav5.latest	0.182	0.155	0.653
skye-mongo2.rescpu.runpk1.latest	0.118	0.096	0.667
skye-mongo2.rescpu.runpk15.latest	0.136	0.121	0.641
skye-mongo2.rescpu.runpk5.latest	0.122	0.099	0.670
skye-mongo2.rescpu.sampleCount.latest	0.191	0.019	0.000

Table B.1: Counter prediction results (continued)

Counter	RMSE	MAE	R
skye-mongo2.sys.heartbeat.summation	0.022	0.018	0.654
skye-mongo2.sys.uptime.latest	0.000	0.000	1.000
skye-mongo2.virtualDisk.numberReadAveraged.average	0.111	0.019	0.831
skye-mongo2.virtualDisk.numberWriteAveraged.average	0.167	0.145	0.646
skye-mongo2.virtualDisk.read.average	0.061	0.012	0.862
skye-mongo2.virtualDisk.totalReadLatency.average	0.087	0.041	0.777
skye-mongo2.virtualDisk.totalWriteLatency.average	0.146	0.126	0.655
skye-mongo2.virtualDisk.write.average	0.216	0.187	0.647

Figure B-1: Actual vs. predicted counters. The red lines are the actual recorded counter values, and the blue lines are the corresponding predicted counter values. The green line is the residuals (predicted value - actual value). Note that some values are missing from the beginning of the predicted counter plots due to the initial input window.

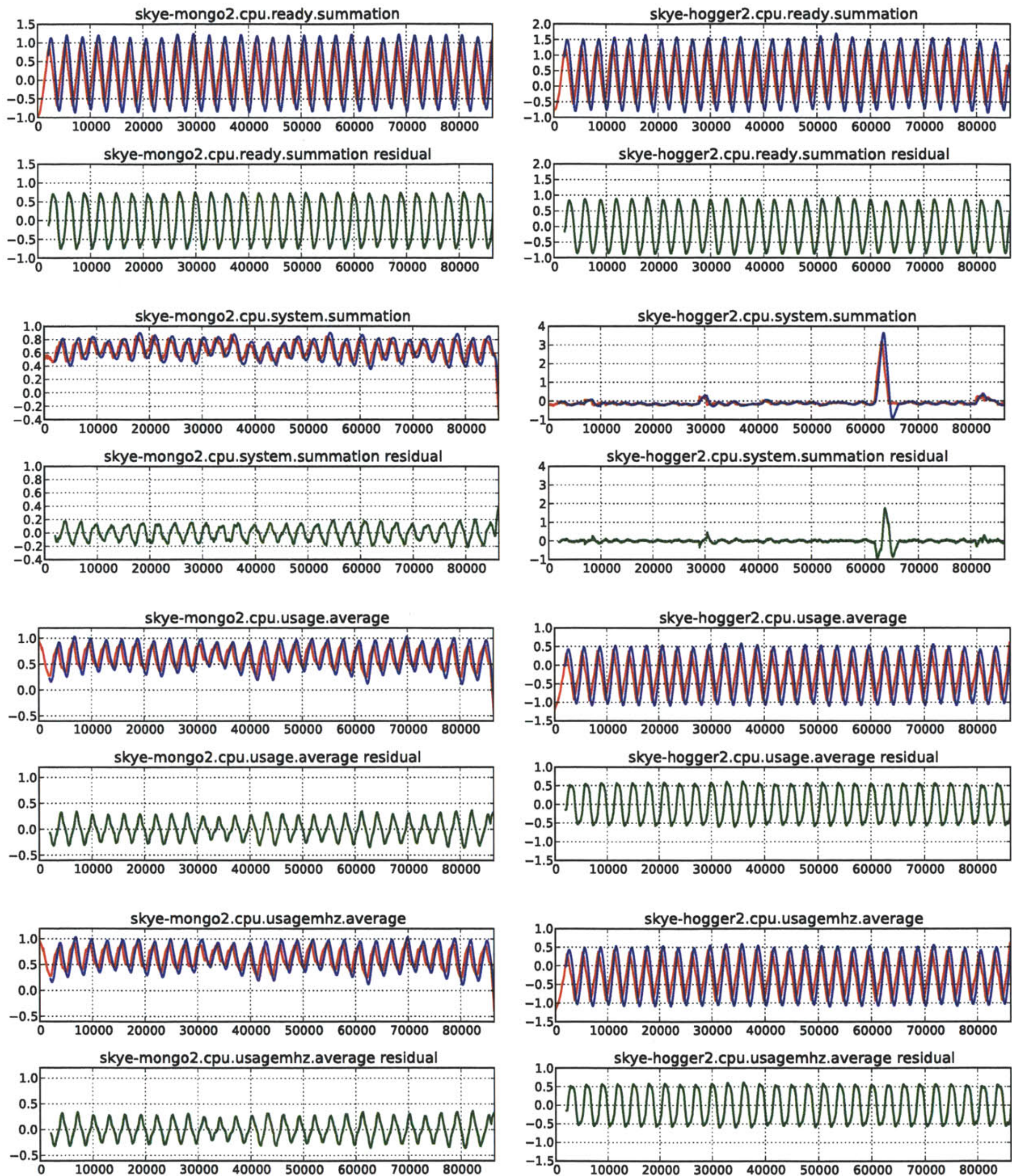


Figure B-1: Actual vs. predicted counters (continued)

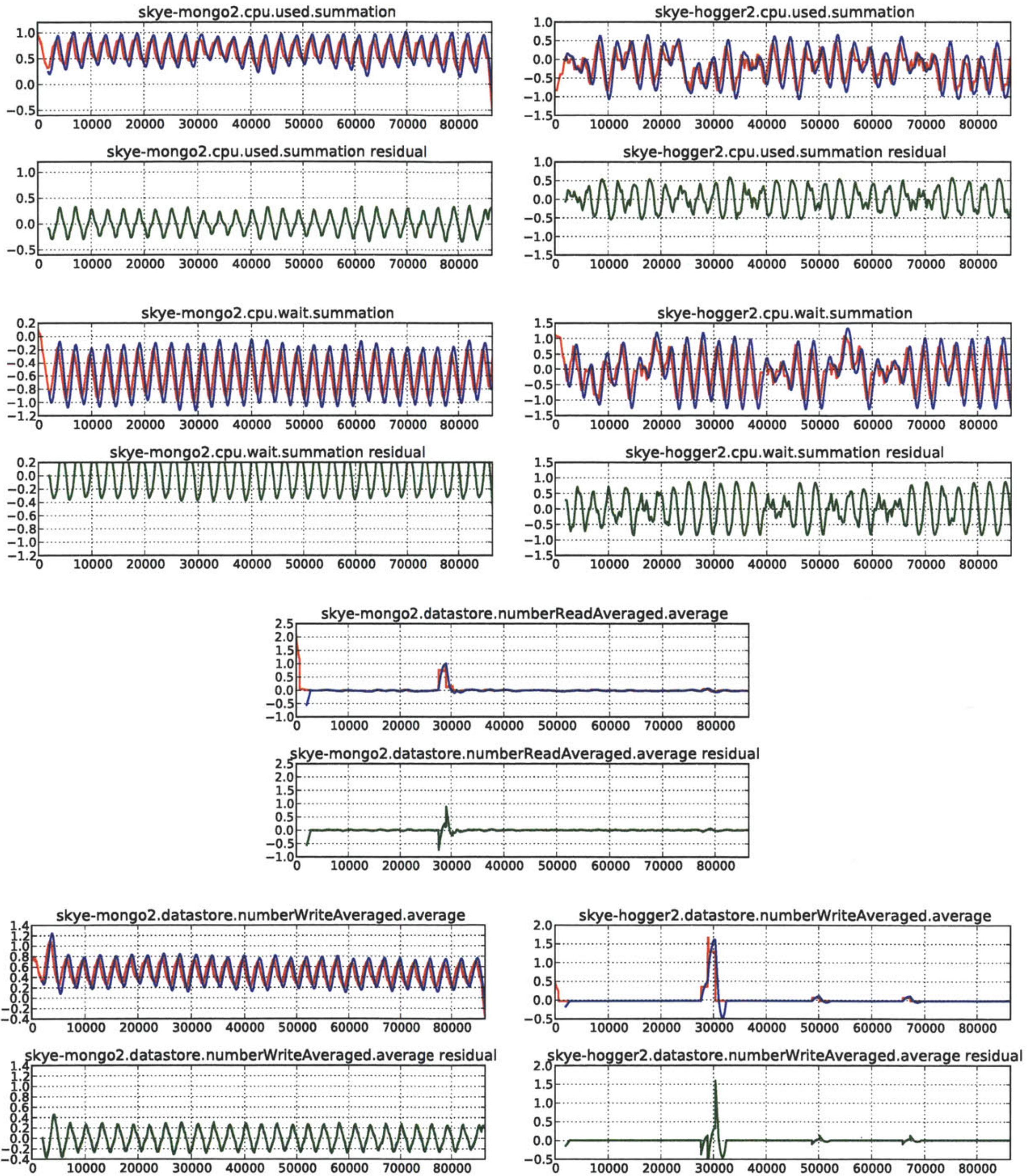


Figure B-1: Actual vs. predicted counters (continued)

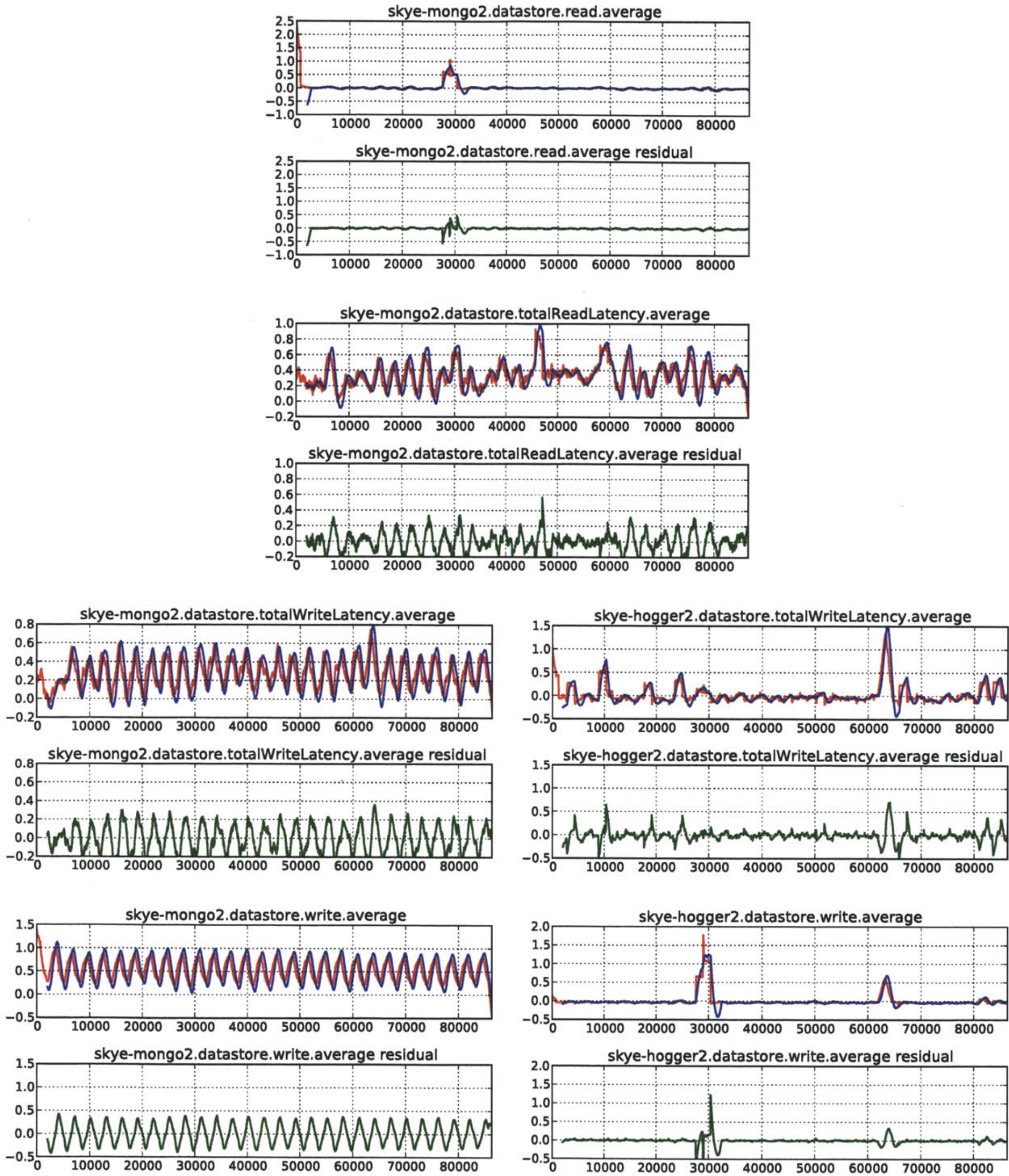


Figure B-1: Actual vs. predicted counters (continued)

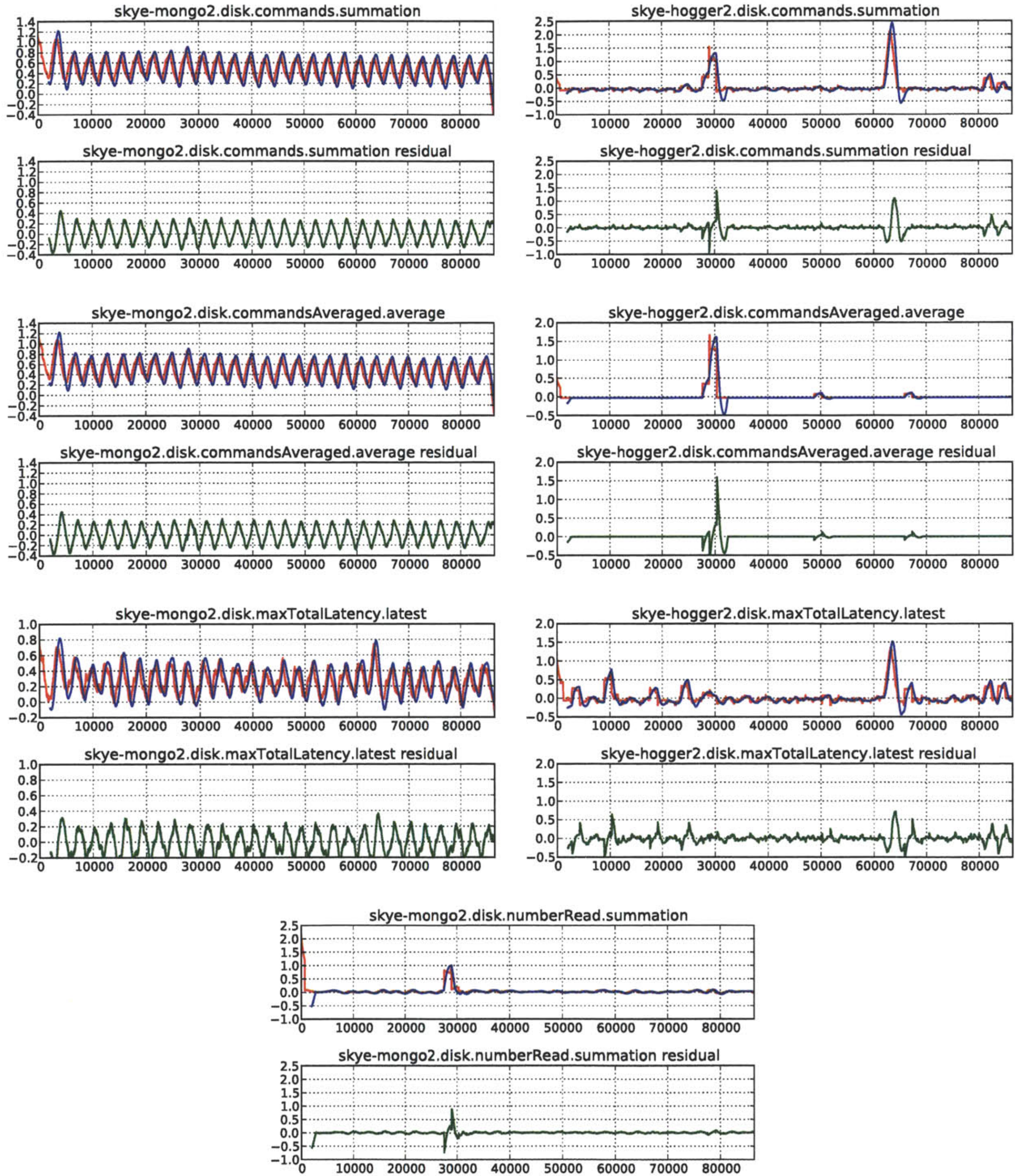


Figure B-1: Actual vs. predicted counters (continued)

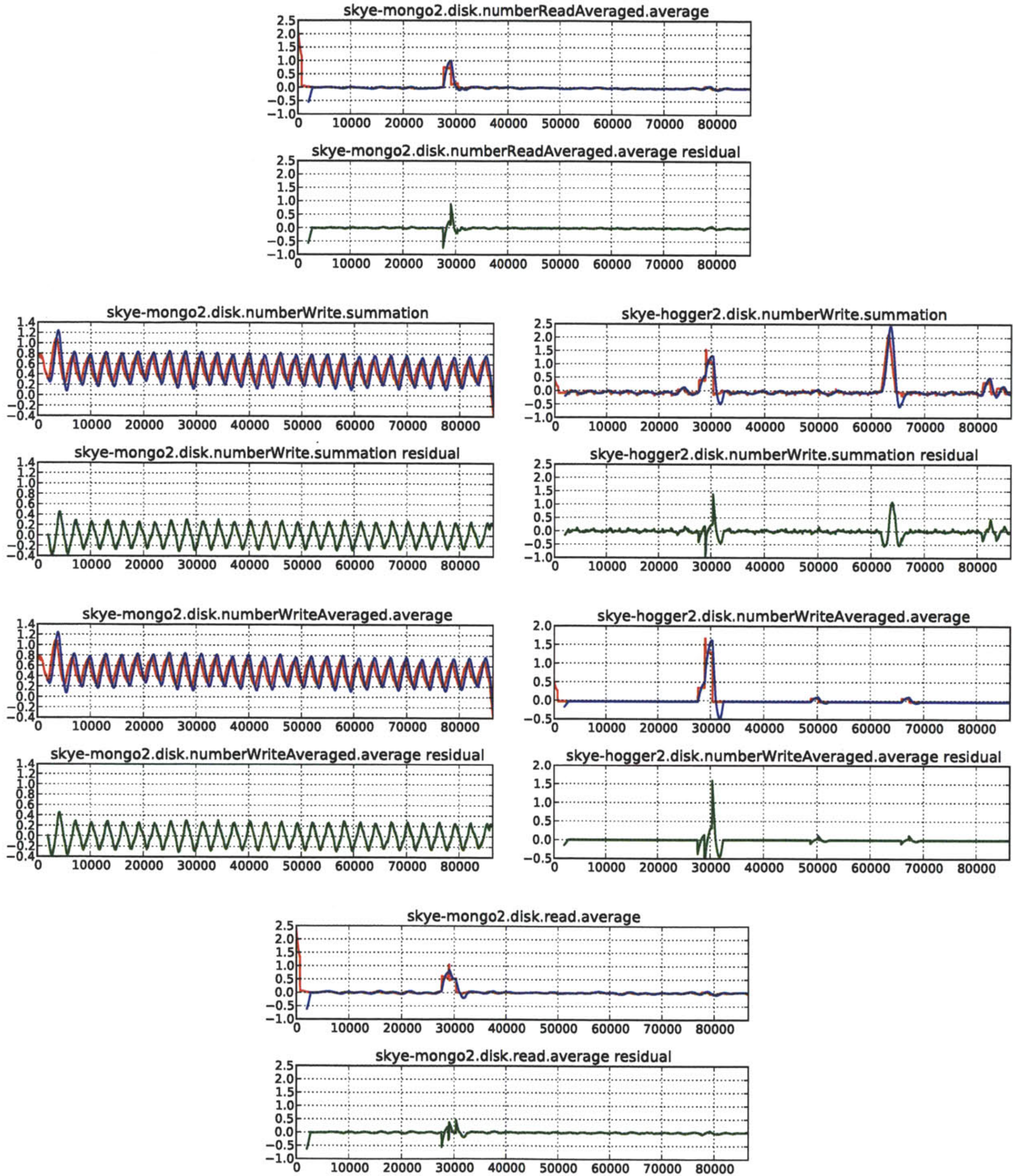


Figure B-1: Actual vs. predicted counters (continued)

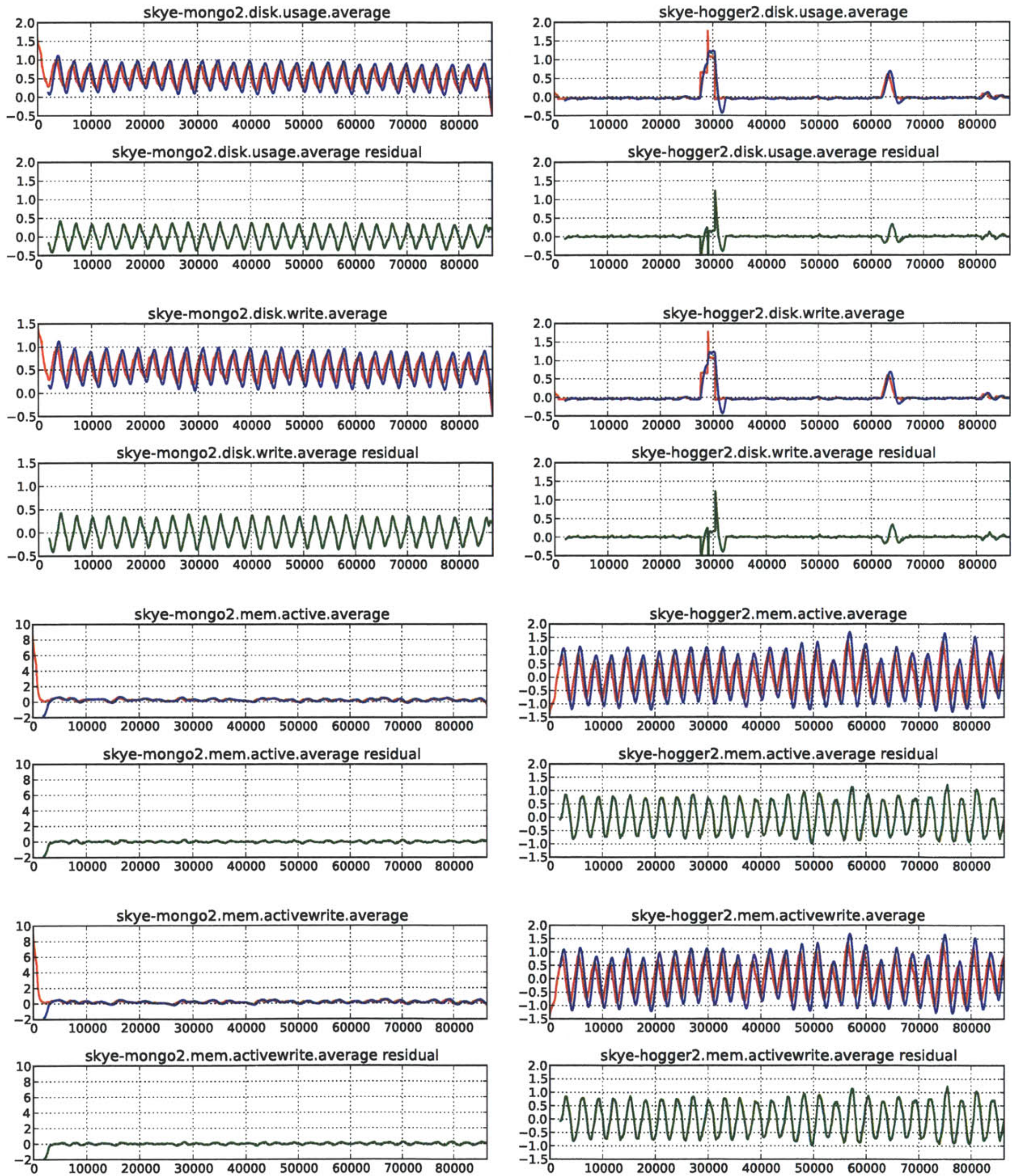


Figure B-1: Actual vs. predicted counters (continued)

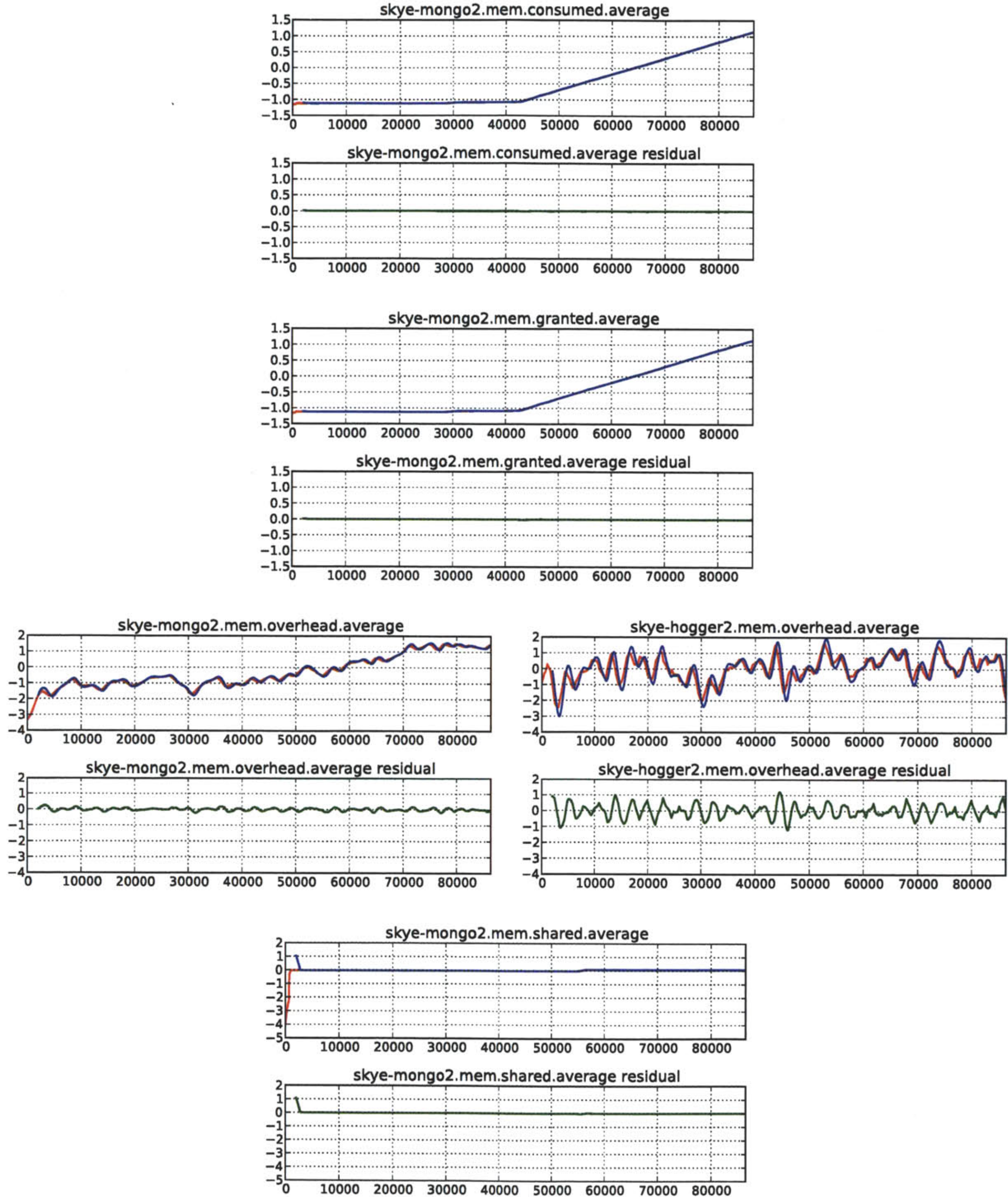


Figure B-1: Actual vs. predicted counters (continued)

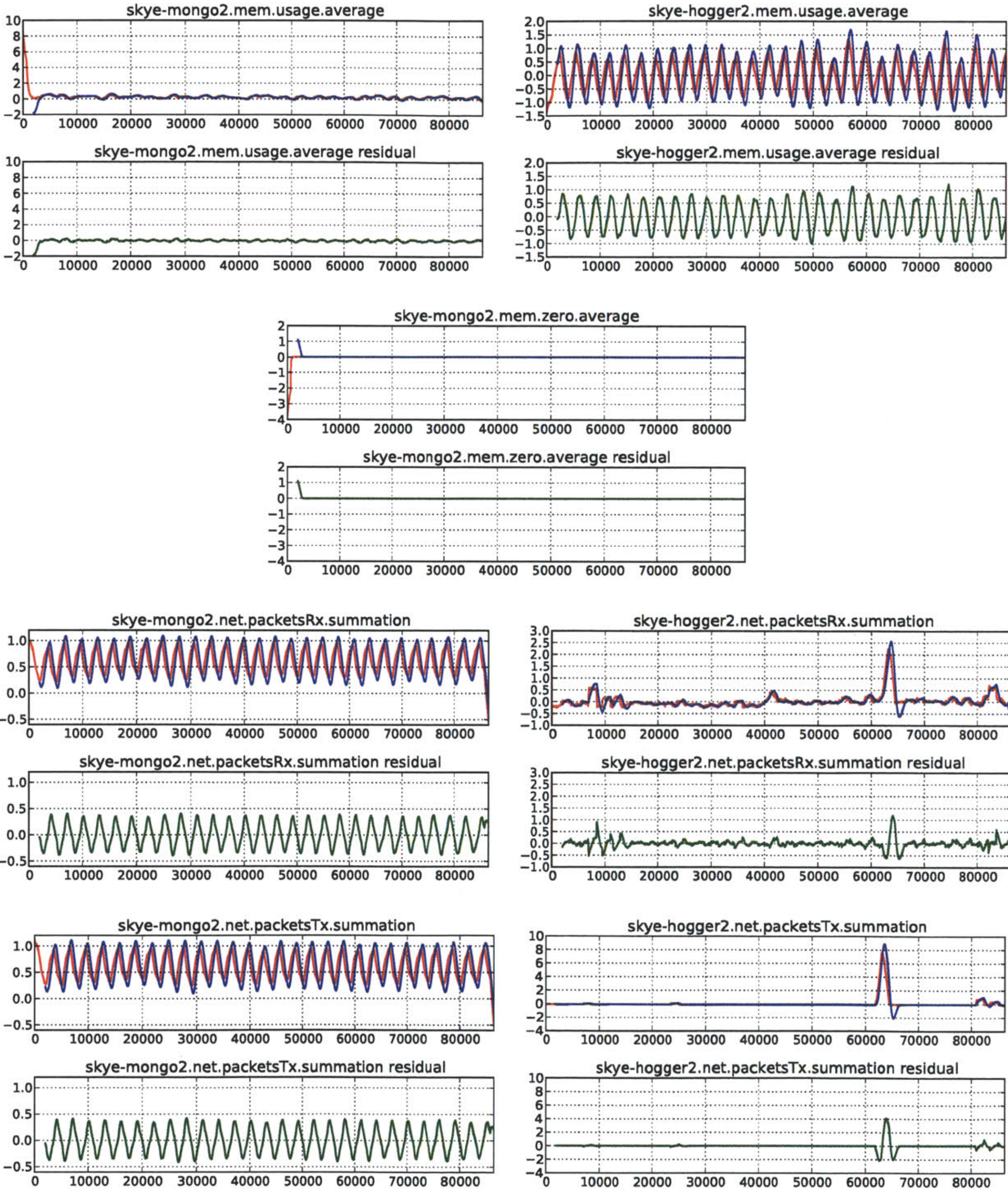


Figure B-1: Actual vs. predicted counters (continued)

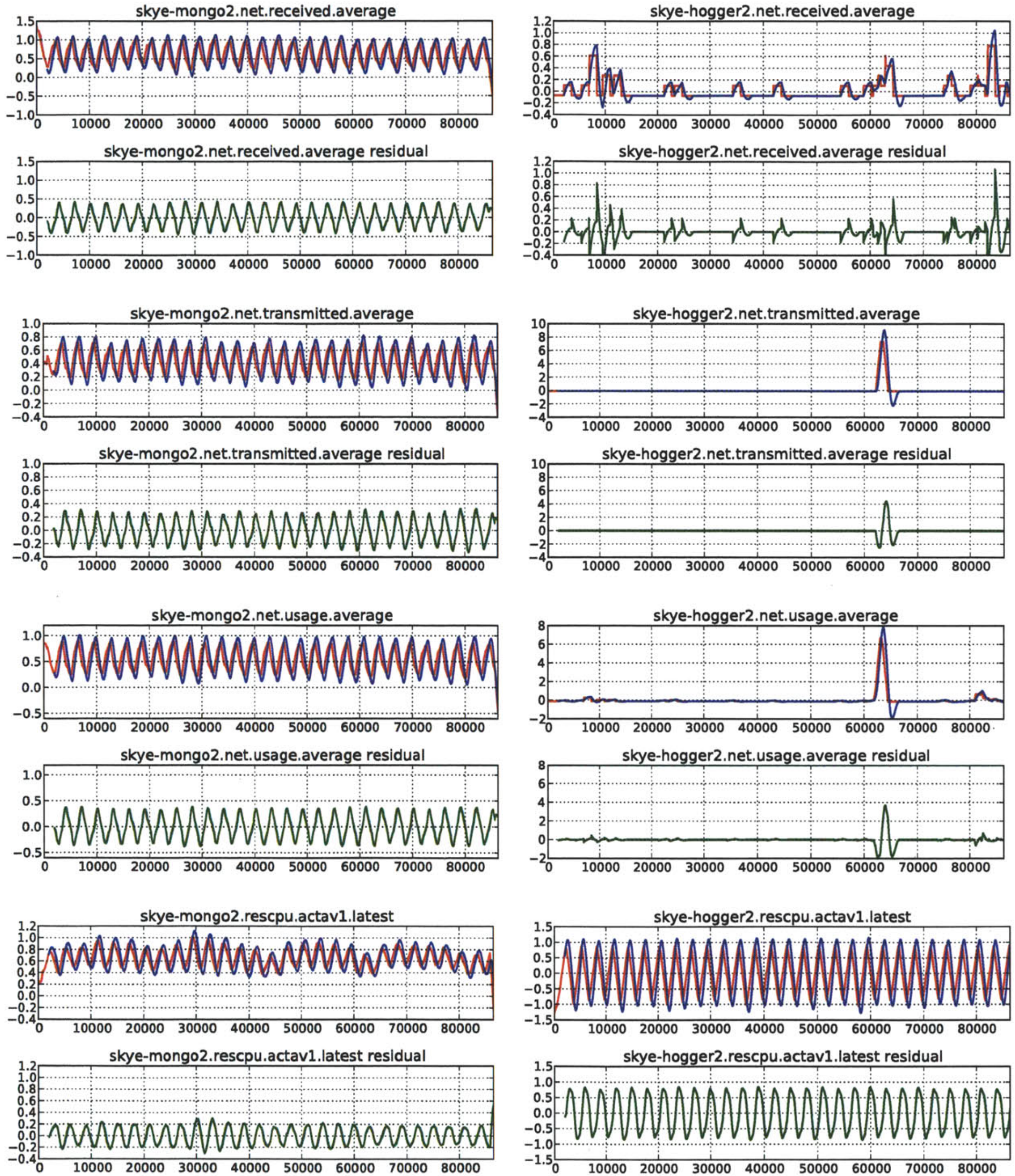


Figure B-1: Actual vs. predicted counters (continued)

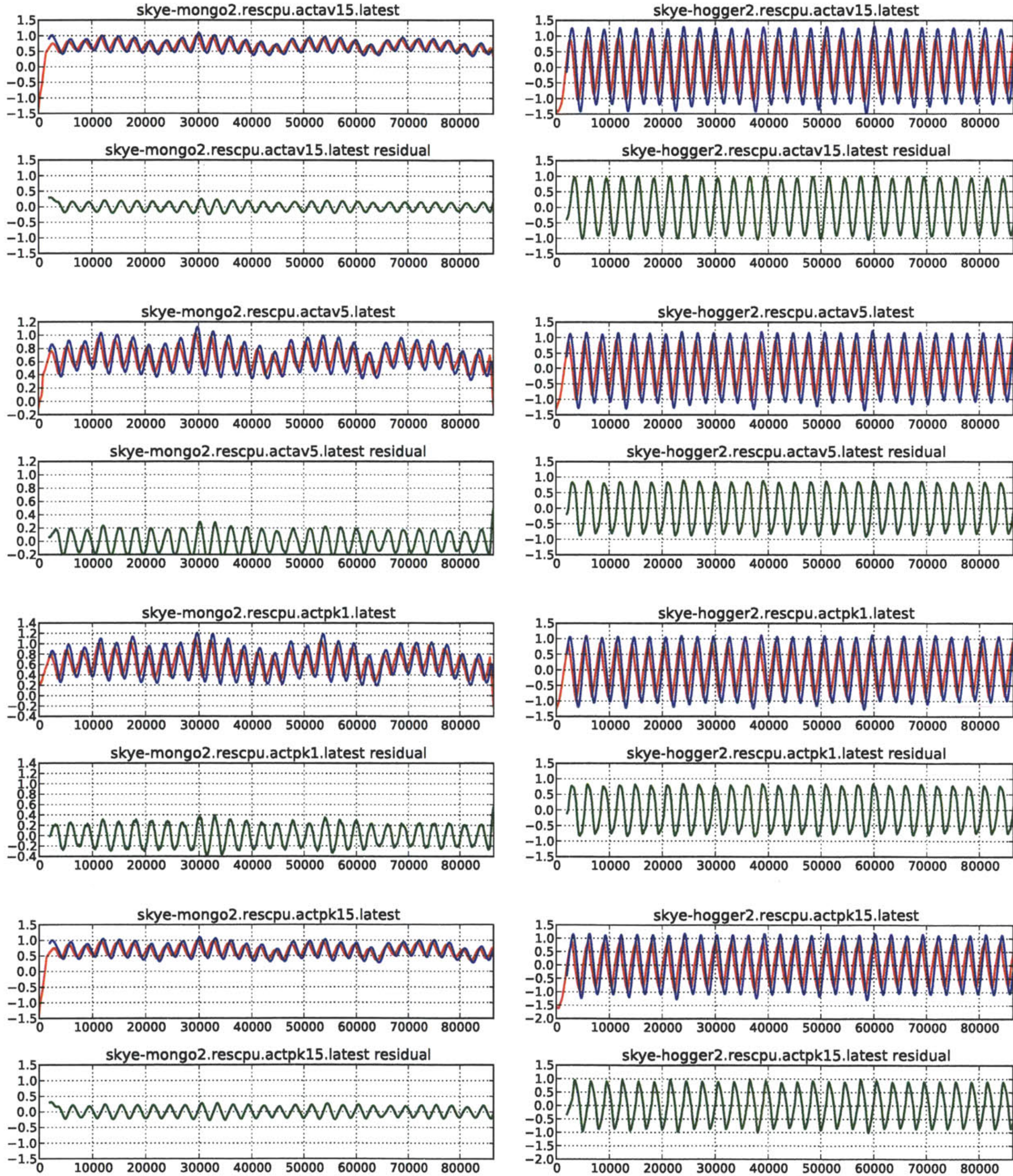


Figure B-1: Actual vs. predicted counters (continued)

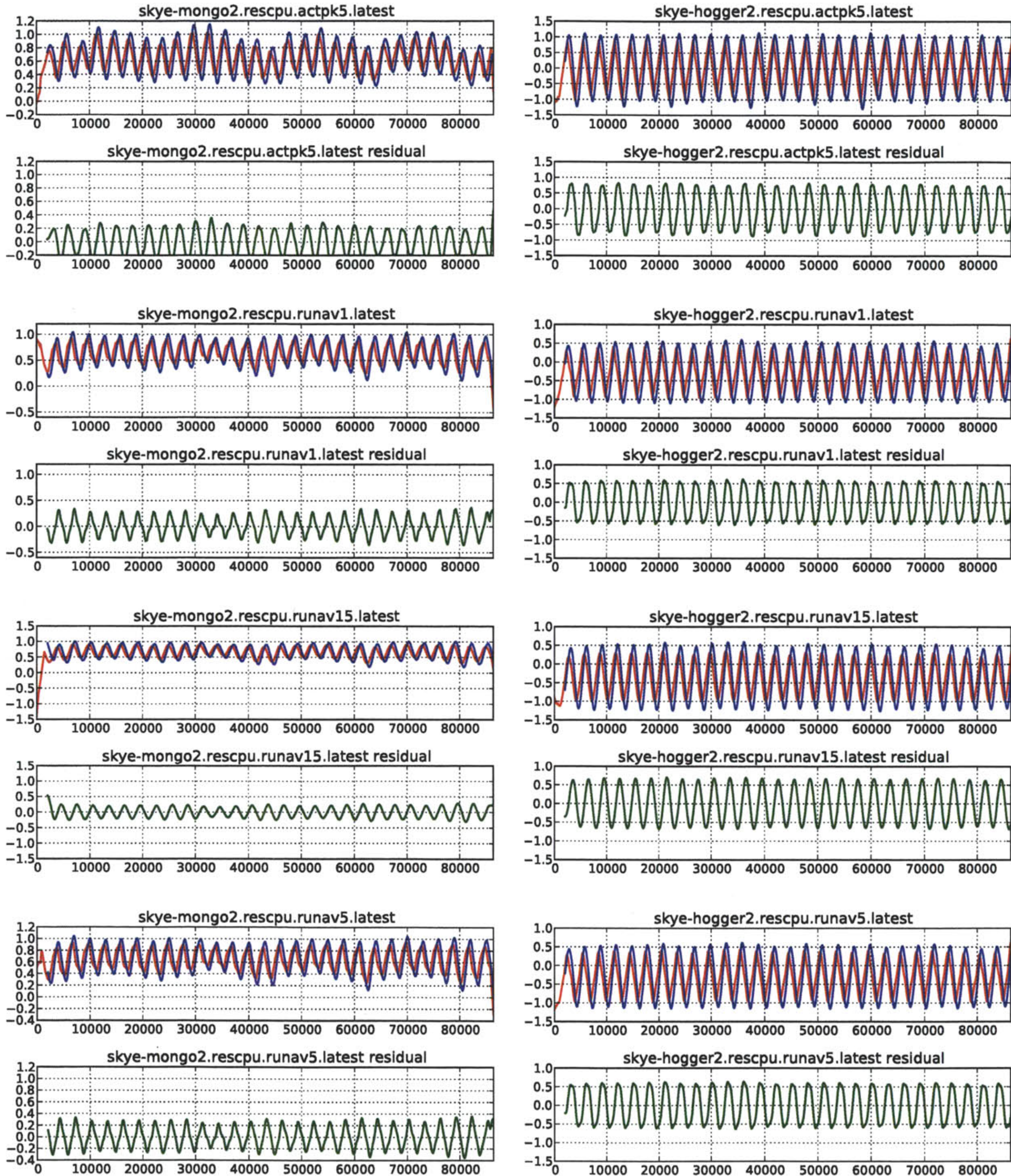


Figure B-1: Actual vs. predicted counters (continued)

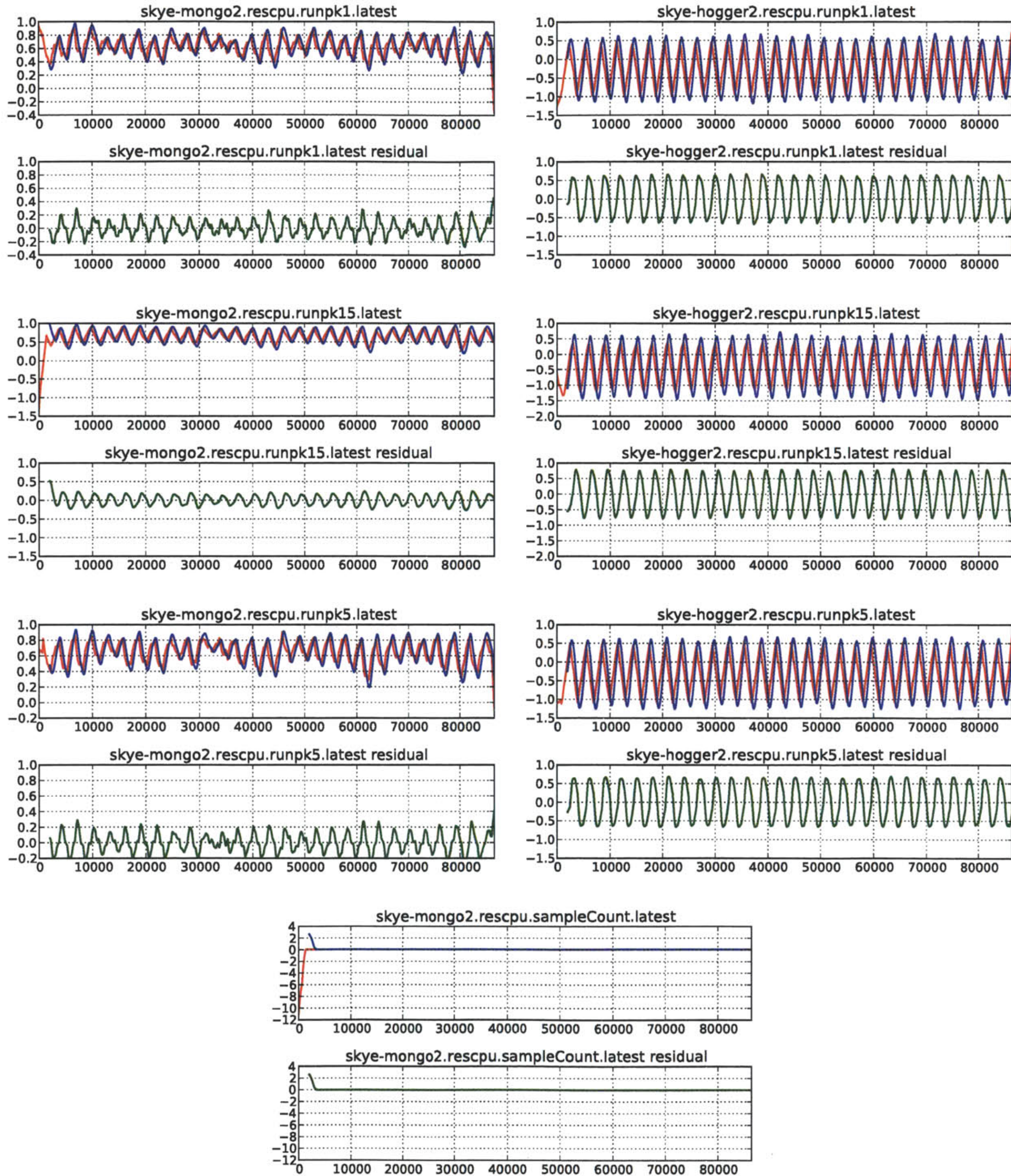


Figure B-1: Actual vs. predicted counters (continued)

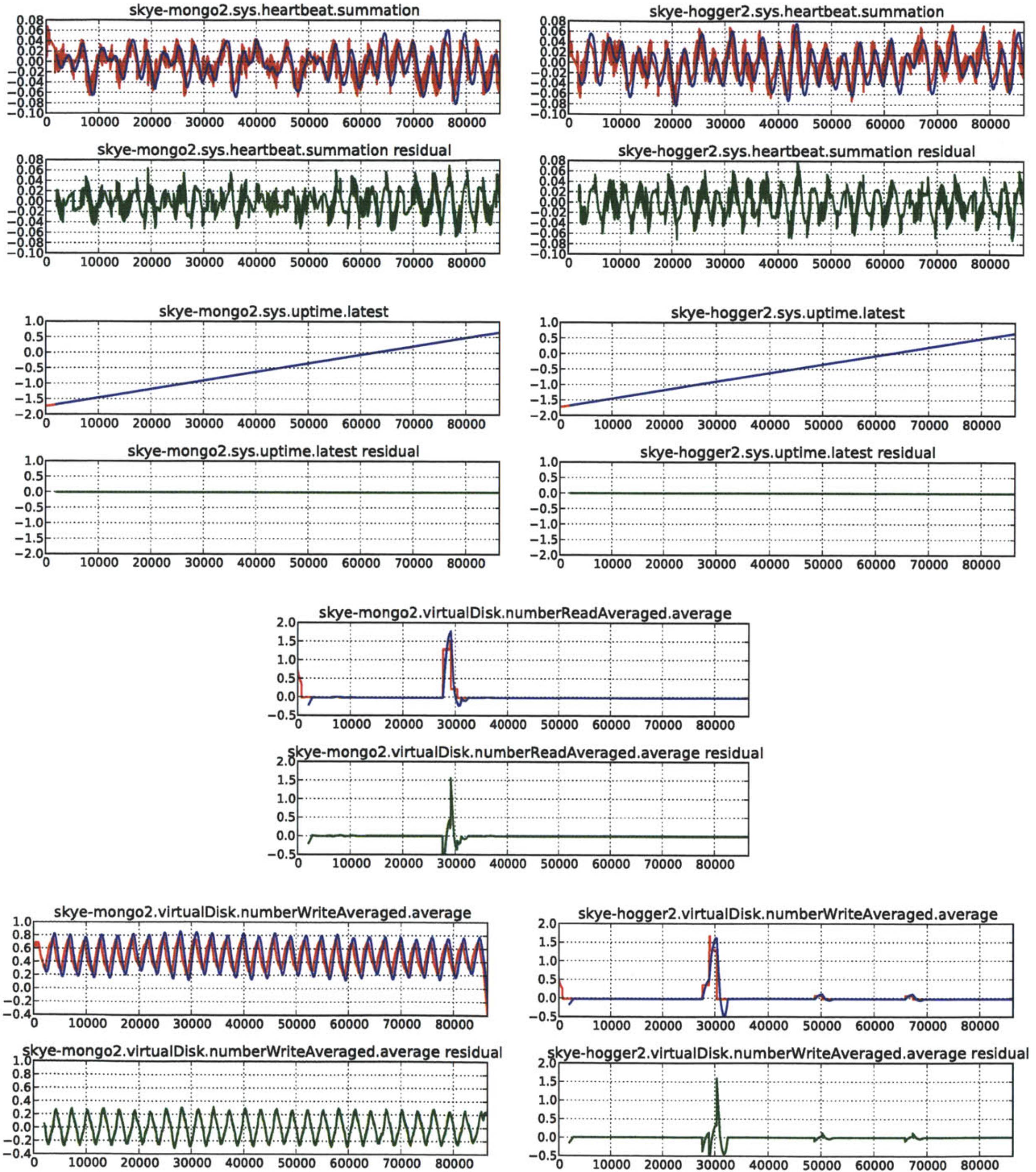
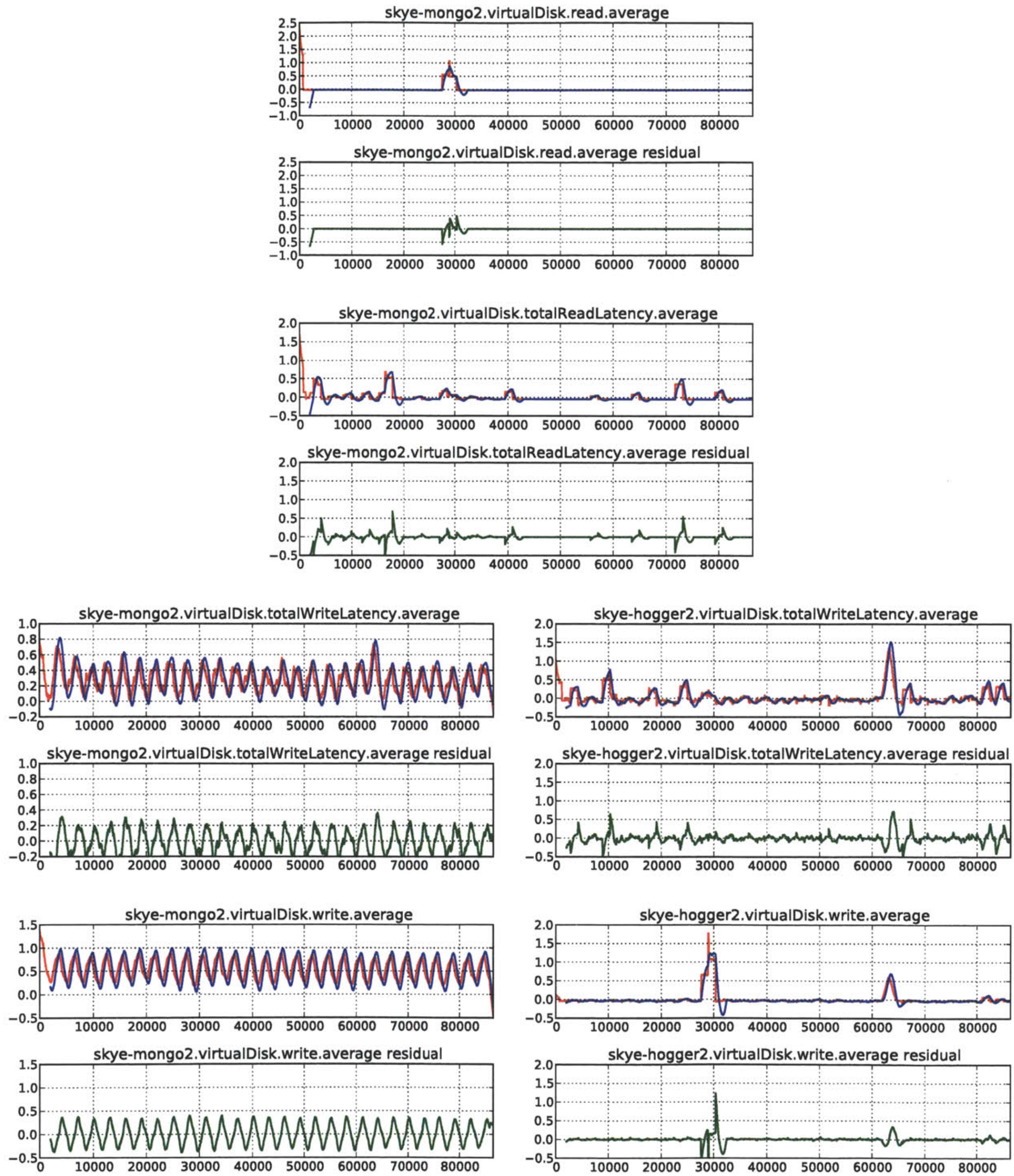


Figure B-1: Actual vs. predicted counters (continued)



Bibliography

[1] MongoDB. http://www.mongodb.org.

[2] Network Time Protocol Version 4: Protocol and Algorithms Specification. http://tools.ietf.org/html/rfc5905.

[3] Rain git repository. https://github.com/yungsters/rain-workload-toolkit.

[4] stress project page. http://weather.ou.edu/~apw/projects/stress/.

[5] vSphere 5 Documentation Center: Managed Object - PerformanceManager. http://pubs.vmware.com/vsphere-50/topic/com.vmware.wssdk.apiref.doc_50/vim.PerformanceManager.html.

[6] Davide Albanese, Roberto Visintainer, Stefano Merler, Samantha Riccadonna, Giuseppe Jurman, and Cesare Furlanello. mlpy: Machine learning python. http://mlpy.sourceforge.net/, 2012.

[7] Aaron Beitch, Brandon Liu, Timothy Yung, Rean Griffith, Armando Fox, and David A. Patterson. Rain: A workload generation toolkit for cloud computing applications. Technical Report UCB/EECS-2010-14, University of California at Berkeley, February 2010.

[8] Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[9] Peter Bodik, Rean Griffith, Charles Sutton, Armando Fox, Michael I. Jordan, and David A. Patterson. Automatic exploration of datacenter performance regimes. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, ACDC '09, pages 1–6, New York, NY, USA, 2009. ACM.

[10] Lawrence L. Chan. Modeling virtualized application performance from hypervisor counters. Master's thesis, Massachusetts Institute of Technology, May 2011.

[11] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[12] Nello Cristianini and John Shawe-Taylor. *An introduction to support Vector Machines: and other kernel-based learning methods.* Cambridge University Press, New York, NY, USA, 2000.

[13] Gartheeban Ganeshapillai. Machine learning techniques to provide application specific performance guarantees. Work done in collaboration with MIT CSAIL and VMware, Inc., August 2011.

[14] Zhenhuan Gong and Xiaohui Gu. Pac: Pattern-driven application consolidation for efficient cloud computing. In *MASCOTS*, pages 24–33. IEEE, 2010.

[15] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *CNSM*, pages 9–16. IEEE, 2010.

[16] Ajay Gulati, Anne Holler, Minwen Ji, Ganesha Shanmuganathan, Carl Waldspurger, and Xiaoyun Zhu. VMware Distributed Resource Management: Design, Implementation and Lessons Learned. Technical report, VMware, Inc., 2012.

[17] Hiep Nguyen, Yongmin Tan, and Xiaohui Gu. Pal: Propagation-aware anomaly localization for cloud hosted distributed applications. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, SLAML '11, pages 1:1–1:8, New York, NY, USA, 2011. ACM.

[18] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 13–26, New York, NY, USA, 2009. ACM.

[19] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[20] Yongmin Tan, Xiaohui Gu, and Haixun Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 173–182, New York, NY, USA, 2010. ACM.

[21] Steve Zhang, Ira Cohen, Moises Goldszmidt, Julie Symons, and O Fox. Ensembles of models for automated diagnosis of system performance problems. In *In DSN*, pages 644–653, 2005.