Modeling Virtualized Application Performance from Hypervisor Counters

by

Lawrence L. Chan

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

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2011

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

Accepted by .....
Leslie A. Kolodziejski
Chair of the Committee on Graduate Students

Modeling Virtualized Application Performance from Hypervisor Counters

by

Lawrence L. Chan

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2011, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Managing a virtualized datacenter has grown more challenging, as each virtual machine's service level agreement (SLA) must be satisfied, when the service levels are generally inaccessible to the hypervisor. To aid in VM consolidation and service level assurance, we develop a modeling technique that generates accurate models of service level. Using only hypervisor counters as inputs, we train models to predict application response times and predict SLA violations.

To collect training data, we conduct a simulation phase which stresses the application across many workloads levels, and collects each response time. Simultaneously, hypervisor performance counters are collected. Afterwards, the data is synchronized and used as training data in ensemble-based genetic programming for symbolic regression. This modeling technique is quite efficient at dealing with high-dimensional datasets, and it also generates interpretable models. After training models for web servers and virtual desktops, we test generalization across different content.

In our experiments, we found that our technique could distill small subsets of important hypervisor counters from over 700 counters. This was tested for both Apache web servers and Windows-based virtual desktop infrastructures. For the web servers, we accurately modeled the breakdown points and also the service levels. Our models could predict service levels with 90.5% accuracy on a test set. On a untrained scenario with completely different contending content, our models predict service levels with 70% accuracy, but predict SLA violation with 92.7% acccuracy. For the virtual desktops, on test scenarios similar to training scenarios, model accuracy was 97.6%.

Our main contribution is demonstrating that a completely data-driven approach to application performance modeling can be successful. In contrast to many other works, our models do not use workload level or response times as inputs to the models, but nevertheless predicts service level accurately. Our approach also lets the models determine which inputs are important to a particular model's performance, rather than hand choosing a few inputs to train on.

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

Acknowledgments

First and foremost, I would like to thank my thesis adviser Una-May O'Reilly. She was extremely knowledgeable and experienced with the EB-GPSR modeling technique, and offered insightful advice on how to structure the experiments. Her excitement and enthusiasm for EB-GPSR always kept me excited to use the technique, and her lighthearted personality brought joy to all who worked with her. I also thank her for her dedication and patience during the thesis writing.

I am also grateful for the support and mentorship of Saman Amarasinghe, who worked closely with me on this project as another adviser. His expertise in the virtualization systems was immensely helpful, and his high-level vision for the project guided the work.

During the summer research portion, I worked closely with Geoffrey Thomas and Kalyan Veeramachaneni, and I would like to thank them for their helpful discussions and collaboration in developing some of the software used in the project. During the year, I also worked with Victor Yarlott, and I am thankful for his help in the simulations. The Evolutionary Design and Optimization group (of which Una-May and I are a part) was also immensely supportive, and I am thankful for their camaraderie.

I gratefully acknowledge the funding from VMware, Inc. that supported this work. None of this would have been possible without the funding and the help of all the people at VMware that supported and advised this project. In particular, I would like to acknowledge Ravi Soundararajan, Carl Waldspurger, Haoqiang Zheng, Arkady Kanevsky, Julia Austin, and Rita Tavilla for their support.

I also thank Evolved Analytics for allowing our research group to beta test the DataModeler package free of charge. I am grateful for their responsive support in making sure our issues were addressed, and for making sure their package was as polished as possible. Their excellent software made our modeling very streamlined, and it allowed us to focus on how to use EB-GPSR to model rather than spending our time developing a modeling system.

Finally, I would like to thank my family and friends for their constant encourage-

ment. I could not have accomplished this without their support.

Contents

List of Figures

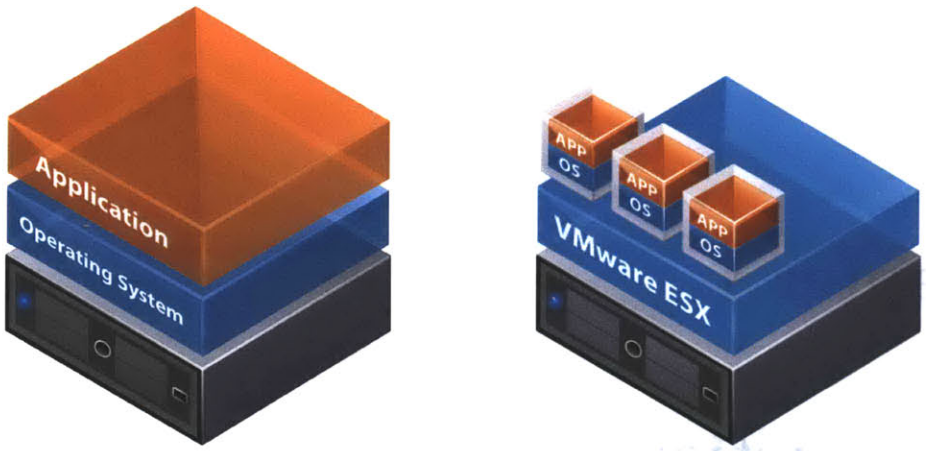
List of Tables

Chapter 1

Introduction

The resource demands of today's datacenters have grown significantly in the last decade. To meet these needs in a scalable way, datacenters have adopted virtualization for server consolidation. In essence, virtualization is the abstraction of hardware from software. In an unvirtualized system, the operating system communicates with the hardware directly (see Figure 1-1(a)). In a virtualized environment, the operating system communicates with the hardware through what is known as the hypervisor, which in turn relays and schedules the instructions to the hardware (see Figure 1-1(b)). The primary result of this is that many virtual machines can run on a single physical system, because the hypervisor emulates a physical host to each virtual machine while managing the instruction flow to the actual physical host. Prior to virtualization, system administrators would run an individual physical host for each operating system that was needed. A datacenter might manage the power, cooling, and physical host maintenance for the client, but otherwise the physical resources were not shared. Today, a vast majority of datacenters use virtualization. Consolidation figures in the tens or hundreds are not uncommon, saving energy in hardware, real estate, and cooling. In addition, the management of virtual machines can now be centralized and streamlined, leading to reduced administration costs.

Most current virtualization systems distribute resources according to a systems-level proportional-share algorithm. The central idea is that each virtual machine is given a certain number of shares, and the amount of resources that that particular



(a) Native operating system (b) Virtualized operating systems

Figure 1-1: Native operating system versus a virtualized set of operating systems

virtual machine is given is proportional its shares relative to the total shares of competing virtual machines. The virtualization system does its best to respect these guidelines, but it is important to understand that the sharing mechanisms measure and distribute raw resources.

Using resource consumption as the explicit service level agreement (SLA) to a datacenter customer is computationally efficient and resource-wise efficient when it is clear how the raw system resource allocation affects performance and the customer can specify what is needed to satisfy expectations. This is true of high performance computing (HPC) scenarios, where it is known that when the VM is being used, it will be used to nearly 100% of its capacity.

In many cases, however, the customer is mostly concerned with application performance, and this performance is a nonlinear function of resource allocation. Because every application can be different, datacenters allocate and guarantee resources, not service levels. It is hence the responsibility of the customer to designate the proper resource level. Many customers will do this in an ad-hoc way, and almost always, there is generous overprovisioning to ensure that the system has enough resources to handle normal workload and a bit of workload variation.

However, to the customer, the nominal resource allocation isn't really the end goal; they expect a certain service level agreement (SLA), and as long as this is met,

the resource allocation is merely a side effect. If the datacenter could guarantee SLA instead of resources, the customer would have to do less testing to determine the appropriate resource level to purchase. Further, there could be much more flexibility for the datacenter to exploit application performance nonlinearities and further consolidate servers dynamically.

One of the main challenges in implementing such a system is getting around the isolation of the virtual machine paradigm. Virtual machines are expected to be isolated, private environments; the hypervisor should avoid looking inside the OS to and see what is being run[1]. Is there a way to ensure SLA without communicating with the application in some way?

1.1 Our Approach

In this work, we model application performance by looking at hypervisor-level performance counters. Using only these counters as inputs, we train models to predict application response times and predict SLA violations.

To train each application, we first do a simulation phase and collect large quantities of data. The simulation stresses the application across many workloads levels, and collects each response time. Simultaneously, hypervisor performance counters are collected. The data is then preprocessed to synchronize timestamps and aggregate the fine-grained requests into larger time slices.

The synchronized data is then used as training data in ensemble-based genetic programming for symbolic regression. This modeling technique is quite efficient at dealing with high-dimensional datasets, and it also generates interpretable models (see Chapter 2.2) for more details). After training models, we test generalization across different data from the same host configuration, and even across unseen scenarios.

Our main contribution is demonstrating that a completely data-driven approach to application performance modeling can be successful. In contrast to many other works,

[1]This is not entirely true; for example, VMware tools can communicate with applications in the OS. In fact, some VM customers might *choose* to allow the hypervisor to look at its performance if it means better, more efficient performance. More on this in Chapter 5

our models do not use workload level or response times as inputs to the models. Our approach also lets the models determine which inputs are important to a particular model's performance, rather than hand choosing a few inputs to train on. Indeed, from over 700 performance counters, the modeling can select a small set of less than 5 counters and generate models that achieve over 95% accuracy.

1.2 Roadmap

Chapter 2 will cover some basic background information on virtualization and VMware-specific details. Chapter 2 also includes a general overview of ensemble-based genetic programming for symbolic regression (EB-GPSR). Chapter 3 will show and discuss results of our experiments. Chapter 4 discusses a selection of related works on modeling and control in virtualized systems. Chapter 5 will discuss future work and open questions. Finally, Chapter 6 will provide a high-level discussion of results and closing remarks.

Chapter 2

Background

2.1 Virtualization Infrastructure

For our experiments, we use VMware ESX 4.1. The next few section will discuss relevant topics such as resource scheduling in VMware ESX, visible hypervisor-level performance counters, and the higher-level Distributed Resource Scheduler.

2.1.1 ESX Resource Scheduling

In order for us to build models on top of the virtualization system, it is important to understand the algorithms behind the low-level resource scheduling. Since we are primarily attempting to saturate CPU (and memory, to some extent), we will cover CPU scheduling first. The algorithms introduce complexity which makes us believe that nonlinear modeling is required.

CPU Scheduling

The ESX CPU scheduler operates at the instruction level, choosing where and when to execute instructions from each VM. Analogous to process or threads at the OS level, ESX defines the concept of a vCPU. At each timestep, the scheduler must decide where to execute the vCPU, or which (if any) of the already running vCPUs to displace.

ESX uses a proportional-share based algorithm to determine which vCPUs take priority. For contending vCPUs, the entitled CPU usage is calculated as a simple ratio of the VMs CPU shares to the sum of the total active CPU shares (i.e. those from the VMs contending for vCPU executions). A VM that has received less than its entitled usage will have a higher priority, with increasing priority as the discrepancy between usage and entitlement increases.

ESX also defines reservations and limits, which are lower bounds and upper bounds on CPU utilization, respectively. In practice, however, limits are not used, because they place a hard limit on the CPU usage of a VM. When shares are used, contending VMs will use up all possible CPU cycles as designated by the CPU shares. When limits are used, CPU cycles can go to waste if the total CPU level falls below the configured levels. Limits are thus hard to maintain, and most datacenters work with shares[1].

Memory, Network, and Disk Scheduling

Memory is shared much like CPU, with shares, reservations, and limits. Analogous to an OS, ESX can swap memory to disk if memory is overcommitted, although this will lead to drastic performance impact. In this work, we try to avoid actively saturating memory as the primary bottleneck because memory is stateful and exhibits hysteresis, so modeling is more challenging.

Network and disk have schedulers also, but generally speaking the configuration is static and unsaturated, and we will assume that these are not bottlenecks in our applications.

2.1.2 ESX Performance Counters

ESX implements the vSphere API to allow the datacenter to collect low level performance counters. Analogous to the `/proc/$(pid)/` counters in a UNIX system, they

[1]VMware ESX also defines the concept of resource pools, which are hierarchical groups of VMs that share some set of resources. Each resource pool can also has its own CPU shares, reservations, and limits, and these entitlements propagate up the hierarchy. If we use resource pools, it becomes more manageable to apply resource limits on the pools to micromanage resource allocation.

are quick statistics collected by the resource scheduler periodically. By default, ESX can collect statistics at 20 second intervals and can be configured to collect faster with slight performance penalties.

In most of our experiments, we dump the entire set of counters at a 5 second resolution. This amounts to 799 counters (with some variation depending on the host hardware and the VM configurations). Our strategy will be to search for correlation between counters and response time. The low-level nature of the counters and the complex nature of the CPU, memory, network, and disk scheduling seem to imply that the modeling will be challenging. Which counters are appropriate, and the relationships between the counters is not readily apparent and it suggests that that best way to learn them is from the data.

2.1.3 VMware DRS

VMware vCenter provides a service known as the Distributed Resource Scheduler (DRS). DRS supports resource pools across multiple physical hosts. It observes resource utilization by VMs and places them in a complementary way on ESX servers via migration. Migration is a means of encapsulating the state of a VM in hot standby mode and moving the VM with its updated data seamlessly to another ESX server.

2.2 Ensemble-based Genetic Programming for Symbolic Regression

Symbolic regression is a method of modeling a system of observations and actions via a set of explanatory variables (a.k.a. inputs, predictors) [Keijzer, 2003, 2008]. In this kind of regression, arithmetic operators are combined with the variables into a syntactically correct arithmetic expression that is intended to predict an unobserved behavior. The model can be represented as a parse tree or a nested and bracketed expression (see Figure 2-1). Variables and constants are leaves of the expression tree, and mathematical operators such as plus, minus, divide, or multiply are internal nodes.

To obtain a prediction from a model with an observed set of variable states, these states are bound to the variables and the expression is evaluated in-order.

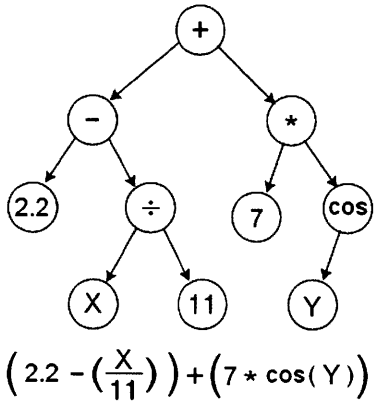


Figure 2-1: An example of a symbolic regression tree.

Unlike some regression methods like neural networks, symbolic regression has the particular advantage of being transparent to interpretation: the output is a human-readable expression that can be understood and modified as necessary. Even compared to simpler models such as linear regressions, the models are often more interpretable, especially in high-dimensional situations. This is because symbolic regression often only uses a subset of the input variables in the tree, and hence we have built-in variable selection [Smits et al., 2005].

One variant of the symbolic regression idea is the use of an ensemble of models instead of a single model. An ensemble allows us to consider multiple models that each offer alternate explanations for the observed responses. This makes sense because there is always more than one explanation for a given response. By using an ensemble of models, we can consider alternate explanations (which vary in accuracy and complexity[2]) and factor in all of them when making a prediction about the response. For instance, we can look at the mean/median or the standard deviation of the outputs to determine the ensemble consensus and the uncertainty of that particular output [Kotanchek et al., 2007, Vladislavleva, 2008].

[2]Complexity is defined as the length of the in-order path through the expression tree

Evolutionary Algorithms for Symbolic Regression

Because the space of possible operator/variable expressions is enormous, it is usually difficult to solve directly for the symbolic expression. Search algorithms are far more effective, and one particularly powerful method that pairs particularly well with symbolic regression is genetic programming [Vladislavleva, 2008]. In this technique, a set of candidate models (the population) is evaluated for fitness. Fitness can be defined in different ways, but often the mean squared error (MSE) or coefficient of determination (R^2 value) is used. R^2 is defined as

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2},$$

and we will use R^2 fitting for our results. Models that fit the function well have higher fitness values. In our particular style of genetic programming, we use multi-objective optimization to address the tradeoffs between model complexity and model accuracy.

Fitness values are used to determine the likelihood of the individual producing offspring for the next generation of solutions [Kotanchek et al., 2008, Smits and Kotanchek, 2004]. Once selected for mating, a pair of solutions undergoes biology-inspired crossover. Parts of the child genotype are randomly chosen from those of the parents, and in this way, there is some chance that the "good" parts of the parents genotypes will be fused into an even "better" child genotype. The genotype also undergoes mutation, which adds random perturbations that may improve solutions that cannot be found via crossover. Over time, the population fitness improves, and it has been shown empirically that such genetic algorithms are often very competitive with other global optimization algorithms.

Evolutionary algorithms are also well suited to symbolic regression due to the tree-like nature of a symbolic regression expression. Crossovers and mutations are straightforward because trees can be spliced together or reconfigured to form related, possibly better trees. Furthermore, it is straightforward to extract an ensemble from a population of models. One way to do so is to bound the complexity and the minimum fitness value and choose the best models satisfying these criteria.

Chapter 3

Experimental Results

We ran experiments using VMware ESX 4.1 as our hypervisor. For the web application experiments, the physical host used was a Dell PowerEdge SC1435, with two quad-core AMD Opteron Processor 2384s at 2.693GHz each and 32GB memory. For the VDI experiments, the physical host used was a Dell PowerEdge R410 with two hyperthreading quad-core Intel Xeon CPU E5620s at 2.393GHz each and 32GB memory.

For the stress testing, external clients running a custom libevent-based python server were used to simulate users and collect data. In each of the experiments, the cycle time (i.e. delay between successive requests in one thread) for each operation was sampled from an exponential distribution with a mean of 5 seconds.

For our EB-GPSR modeling, we used a proprietary Mathematica package called DataModeler [Evolved Analytics, 2011, Kotanchek, 2010], which encapsulates the EB-GPSR algorithm and includes a set of utilities to handle model archiving and visualization.

First we will discuss web application modeling and examine the generalizability of these models. Then we use the same methods to model database server performance, and finally we shift to a virtual desktop infrastructure and test the performance of common desktop applications.

3.1 Web Application Modeling

We will focus primarily on the Apache web server for our experiments. There are certainly other web servers with perhaps better performance, but Apache is still the predominant web server in use today [Netcraft, 2011], with over 62% market share.

We built a small social media (SM) web application that associated short strings with a particular user (like Twitter). When queried, the application would query a subset of the strings and sort them. The application was written using Django, a python framework for web development. Little time was spent optimizing the application for performance; we wrote a straightforward implementation and left it as such. In a datacenter, the customers have free reign over their VMs, and even unoptimized code should meet the SLA, so we should make no assumption about code performance and should in fact ensure that unoptimized code is also accounted for. Also, unoptimized code will perform slower and reach breakdown with less stress, making it easier for us to stress test the server.

3.1.1 Limit-based Modeling

Initially, we wanted to verify that server response was indeed nonlinear by varying the load on a web server with a range of CPU allocations (limits). Using this SM server, each run consisted of setting a CPU limit on the VM and ramping up the request rate linearly. Intuitively, we would expect the server to respond with minor performance impact as request rate is increased, since the system will just consume more of the available resources. At some point, however, the server will reach saturation and start exhibiting exponentially slow response times.

Indeed, in Figure 3-1, we can see that response time is flat until the breakdown point, and then the response times jump significantly. One detail to note is that response times flatten out after reaching saturation. This is because of the way we are simulating clients. If a client normally waits one second between requests, and the request takes longer than a second, then the effective rate will be less than it should be. Thus, after saturation, the request rates actually saturate as well and not reach

the target.

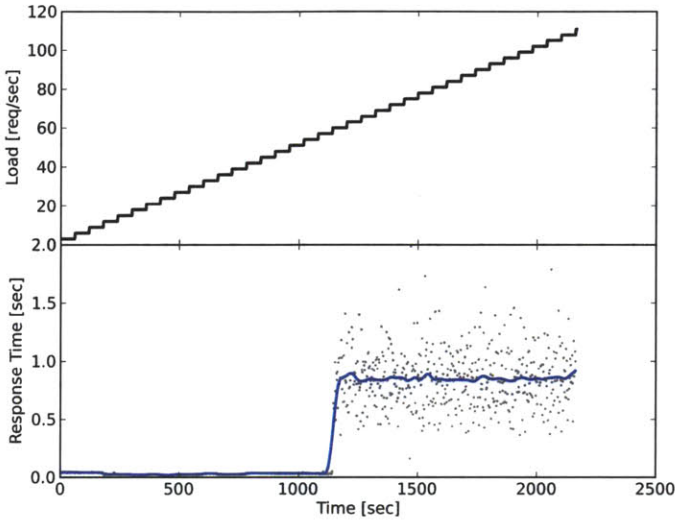


Figure 3-1: Response times on social media Apache application given a linearly increasing load rate. A moving time average line is shown for visualization purposes (blue line).

This response curve is collected for each of the CPU limits from 300MHz up to 1800MHz in increments of 100MHz. Intuitively, the breakdown point should decrease as the server is allocated less resources, because it takes less to saturate the resources. The breakdown severity should also be more extreme with lower limits because the VM capability is lower. The data we collected closely follows this intuition, as shown in Figure 3-2.

We wish to predict how close we are to breakdown. If we model the response time, it may not give us enough warning time to react, since response time is fairly nonlinear and it is unclear what the proper response time threshold should be. Ideally, we would be able to predict how close we are to breakdown. To do this, we introduce a metric called "percentage to breakdown" (PtB). Because the request rate is ramped up linearly from zero, it is straightforward to define this metric: we find the breakdown point (in request/sec), and the percentage to breakdown is the ratio of the current request rate to the breakdown request rate. We also bound this metric at 1, because we are only interested in request rates below breakdown and any breakdown response

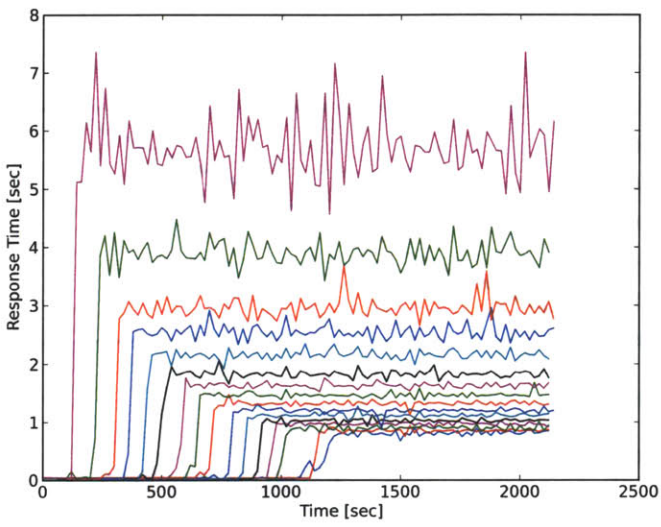


Figure 3-2: Response times on social media Apache application given a linearly increasing load rate. Each line represents a different CPU limit from 300MHz to 1800MHz in 100MHz increments. The lines with lower breakdown points correspond to lower CPU limits.

is the same from a user perspective. Expressed mathematically,

$$\text{PtB} = \min\left\{\frac{r}{r_{sat}}, 1\right\},$$

where r is the request rate and r_{sat} is the saturation/breakdown request rate.

We have defined PtB, but r_{sat} is still an ad-hoc threshold or a visual label. To more algorithmically define breakdown, we model the response times as a mixture of Gaussians (one for the saturated regime and one for the unsaturated regime). The saturation point is the point when the maximum likelihood estimate of the generating Gaussian switches from one to the other (i.e. from the unsaturated regime Gaussian to the saturated regime Gaussian). From this breakdown point, we follow the linear request rate curve down to zero to create the PtB metric. This metric is shown in Figure 3-3.

Our objective is to model PtB using the hypervisor-level performance counters only. From vSphere, we exported the performance counters in every rollup type. We then selected the counters related to CPU usage to use as inputs to the models. The

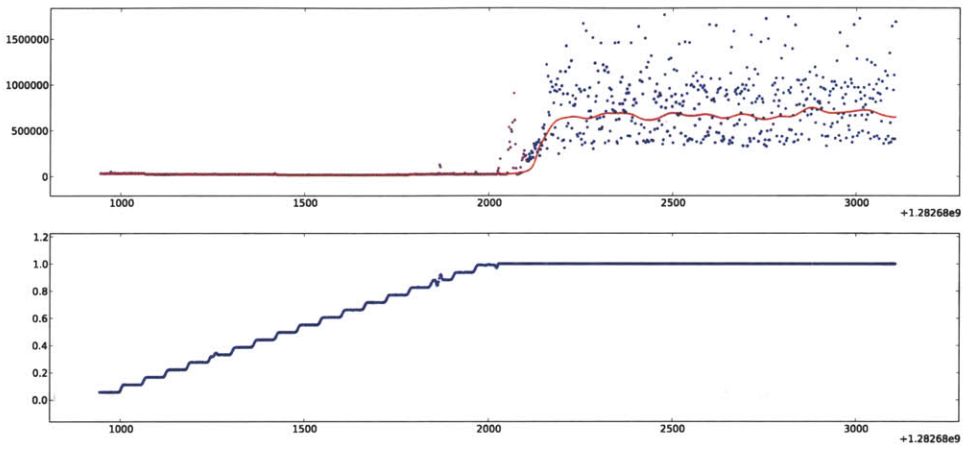
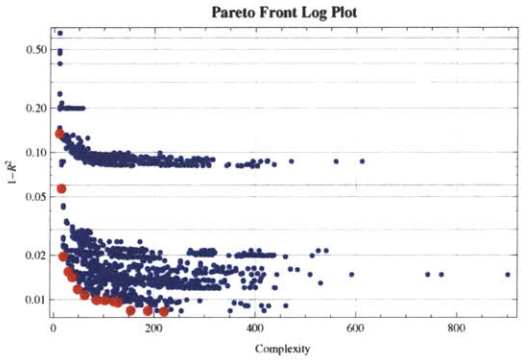
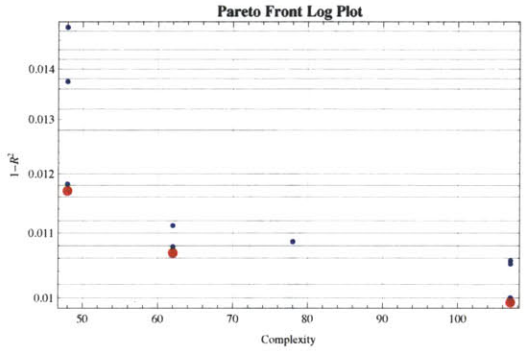


Figure 3-3: Percentage to breakdown at 1800MHz. The upper plot shows the response times, with a moving time average shown with a red line.

results of the modeling are shown in Figure 3-5.



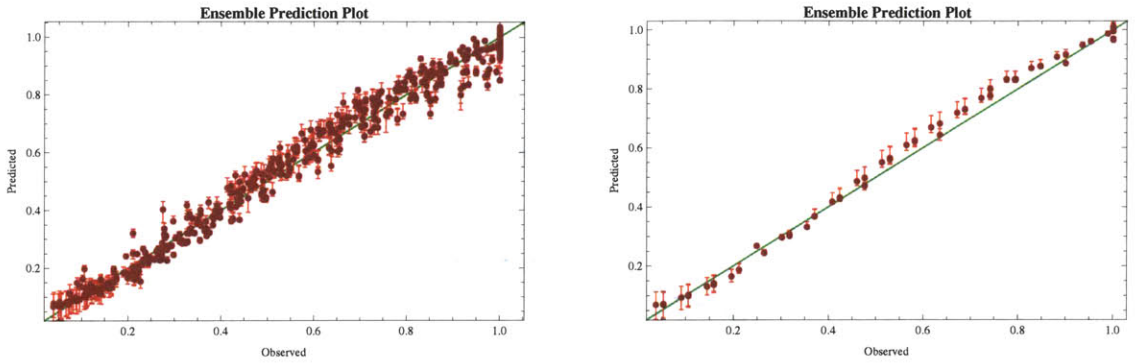
(a) Pareto front log plot of GPSR models



(b) Pareto front log plot of GPSR models, boxed around $1 - R^2 = 0.02$ and $C = 200$

Figure 3-4: Fitness of models for SM web application in limit-based simulation

The GPSR models can then be used to predict the breakdown point, just before it happens. For illustration purposes, let us say that we want a warning at 95% to breakdown. Using these models, we can monitor the PtB and raise a flag when the PtB is above 95%. In Figure 3-6, we can see the predicted 95% breakdown point as well as the PtB response. Another example at 900MHz is shown in Figure 3-7. The other CPU limits from 300MHz to 1800MHz all have similar outputs, and every single one accurately preempts the breakdown point. What's critical to being able to act is having ample time to react. Because we predict the percentage to the breakdown workload, the model user is free to choose the level at which reaction should happen.



(a) Ensemble predictions vs actual inputs

(b) Ensemble test predictions

Figure 3-5: Model ensemble prediction results for Percentage to Breakdown (PtB) prediction for SM web application in limit-based simulation

If workload changes quickly, we might want to threshold at 80%, but if it is slow moving, 90-95% may be sufficient. Forecasting techniques can dynamically detect the rate and change this threshold depending on how fast the workload is changing.

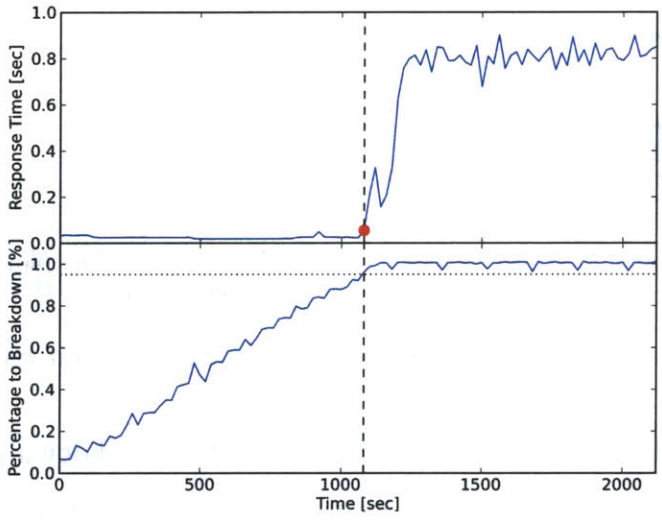


Figure 3-6: Percentage to breakdown estimate for 1800MHz

These results indicate that we can effectively model application performance given only the hypervisor counters as inputs. Using these models, it is possible to form thresholds and flags to pass into a control module.

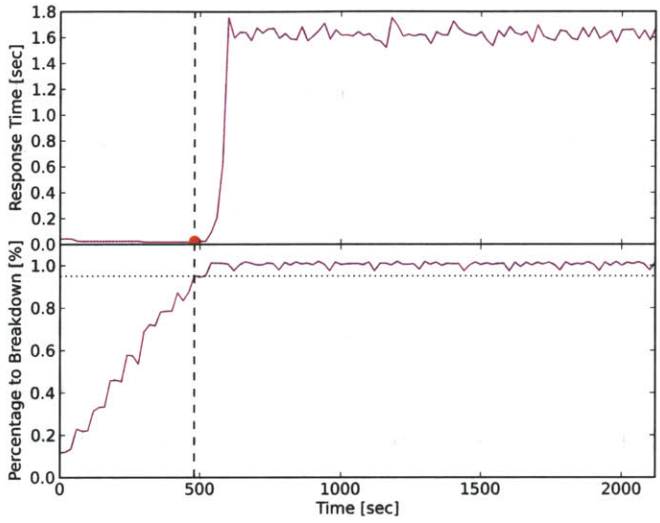


Figure 3-7: Percentage to breakdown estimate for 900MHz

3.1.2 Share-based Modeling

In the previous scenario, CPU contention was simulated by limiting the CPU usage of the VM. In industry, however, CPU limits are not often used; instead, datacenters opt to use shares to designate server priorities. This way, the VMs have more flexibility in stealing resources from each other.

To more realistically simulate a contention scenario in which VMs share resources, we placed another VM on the same machine and measured the performance results. Several content types were placed on the contending VM, and the next few sections detail the results of each combination.

Identical Social Media Web Application

In this experiment, we created a replica of the SM web application and varied the request levels on the two servers (call them SM1 and SM2 for the original and the replica, respectively). At the hypervisor level, each VM is given the same amount of shares for CPU, memory, network, and disk.

In Figures 3-8(a) and 3-8(b), we see that the response time of SM1 server degrades as its own request rate increases (horizontal axis). The breakdown point also decreases

when the SM2 request rate is increased. This makes sense because the VMs are now contending for CPU cycles. However, the breakdown point is not as well-defined as before. It is unclear how to define a PtB without a clear boundary between saturation and normal operation. Thus, we model the response times instead. Since SLAs typically measure the 95th percentile of response time, we parallel this and model the 95th percentile of response time. To make the modeling more closely match human expectations, we take the log of the 95th percentile response time before we perform modeling. A 50ms slowdown between 50ms and 100ms is not the same as a 50ms slowdown between 1.0s and 1.05s; by using the log response times, the slowdowns we measure are multiplicative slowdown factors. This is also better for modeling, since the model will not focus too much effort on the additive slowdowns in the breakdown regime.

We take the performance counters of both the SM1 VM and the ESX host and synchronize them with the response times of SM1 by binning the response times into the 10 second intervals used by ESX (the ESX counters are already synchronized). For each bin, we take the set of response times and calculate the 95th percentile of response time. Each 10 second interval becomes a training point, with a set of hypervisor counters and a 95th percentile of response time.

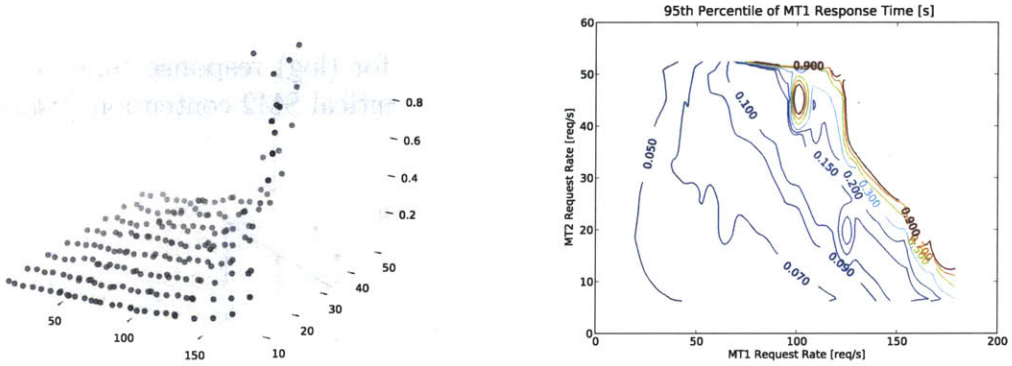
We then split the data into 1283 training points and 321 test points. The training data is then passed into the modeler with the full set of hypervisor counters as input. In contrast to the limit-based modeling in the previous section where we chose a set of counters, we input all of the counters (794 counters). For our response variable, we used the 95th percentile of response time.

DataModeler was used to perform the EB-GPSR algorithm. The EB-GPSR model fitnesses are shown in Figures 3-9(a) and 3-9(b). Ensemble prediction results for the training set are shown in Figures 3-10(a) and 3-10(b), with the ensemble prediction ranges for each data point. A similar plot for the test set predictions is shown in Figure 3-11. Finally, the Pareto front models are shown in Table 3.1.

Prediction results are above 90%. Even on the test set, we have 90.5% accuracy. This is good, especially if we consider that the inputs to the model are all

generic vSphere performance counters, and there is nothing monitoring request rate or throughput. It is also important to notice how nonlinear the models, especially for higher R^2 values.

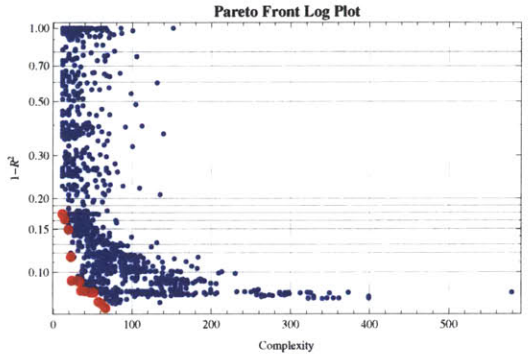
One of the main advantages of GPSR is that it is capable of handling very high-dimensional inputs. In this experiment, the input is 794-dimensional. We know that most of these are irrelevant to the application performance, but we included them to test the robustness of the EB-GPSR modeling method. It turns out that the models can reduce the dimensionality quite quickly to the most important subset. The most frequent variables are shown in Table 3.2. The first few counters are CPU ready counts. We know intuitively that these are some of the most important counters because they count the moments where a CPU instruction cannot be executed because no CPU cores are available. There are also host CPU utilization counters, from which the models can calculate the relative CPU usage of the VM versus the whole host. The network traffic also ends up in 18.8% of the models; this also makes sense because the network traffic can be used to infer the request rate. Heuristically, these explanations all make sense, but the advantage of a GPSR system is that we can include all the data and have it learn behavior, rather than us choosing counters and/or model structures.



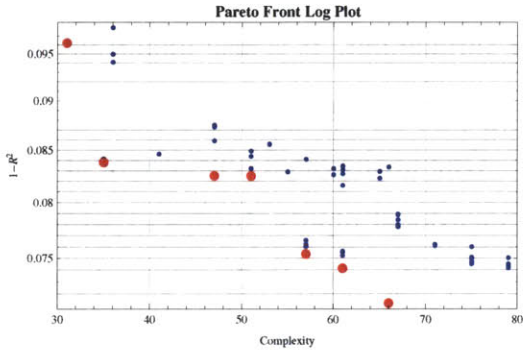
(a) 3D scatterplot of 95th percentiles of SM1 response time. The axes are the same as in (b)

(b) Contour plot of 95th percentiles of SM1 response time

Figure 3-8: SM1 response times for varying request rates on SM1 and SM2

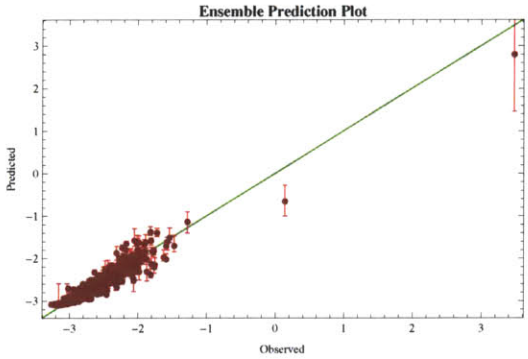


(a) Pareto front log plot of GPSR models

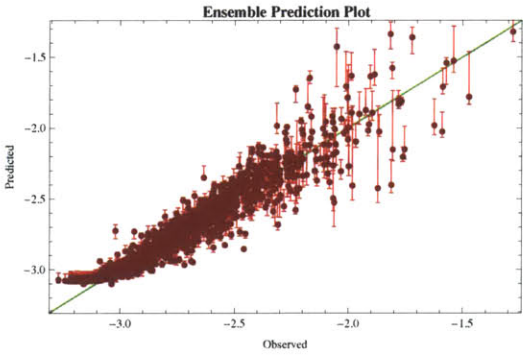


(b) Pareto front log plot of GPSR models, boxed around $1 - R^2 = 0.1$ and $C = 100$

Figure 3-9: Fitness of models for SM web application in share-based simulation



(a) Ensemble predictions vs actual inputs



(b) Ensemble predictions vs actual inputs (zoomed in)

Figure 3-10: Model ensemble prediction results for (log) response time of the SM1 web application in share-based simulation with identical SM2 contention (training set)

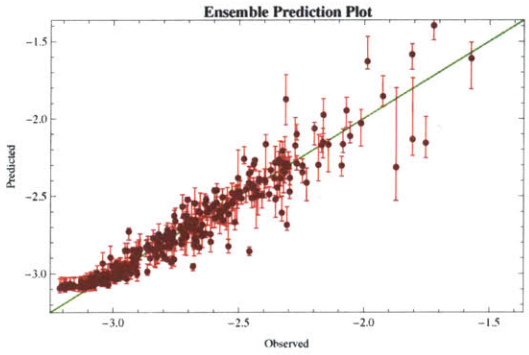


Figure 3-11: Model ensemble prediction results for (log) response time of the SM1 web application in share-based simulation with identical SM2 contention (test set)

Complexity	$1 - R^2$	Function
11	0.173	$-3.056 + (2.685 \times 10^{-4})\, x_{673}$
14	0.165	$-2.984 + (1.834 \times 10^{-8})\, x_{440}\, x_{673}$
19	0.149	$-3.061 + \frac{0.487\, x_{673}}{x_{745}}$
22	0.115	$-3.010 + (2.188 \times 10^{-7}) x_{673}(931.659 + x_{729})$
23	0.092	$-3.048 + \frac{(1.225 \times 10^{-5})\, x_{440}\, x_{673}}{x_{433}}$
33	0.091	$-3.047 + \frac{(1.280 \times 10^{-5})(x_{440} - x_{433}) x_{673}}{x_{433}}$
35	0.084	$-3.115 + (3.286 \times 10^{-7})\, x_{673}(1053. + x_{36} - x_{725} + x_{729})$
40	0.083	$-3.207 + (3.349 \times 10^{-7})\, x_{440} \left(9.614 + \frac{x_{673}}{x_{433}}\right)^2$
47	0.082	$-3.101 + (3.056 \times 10^{-7})(-3.172 + x_{673} + x_{729})(1053. + x_{36} - x_{725} + x_{729})$

Table 3.1: Model table for SM1 models

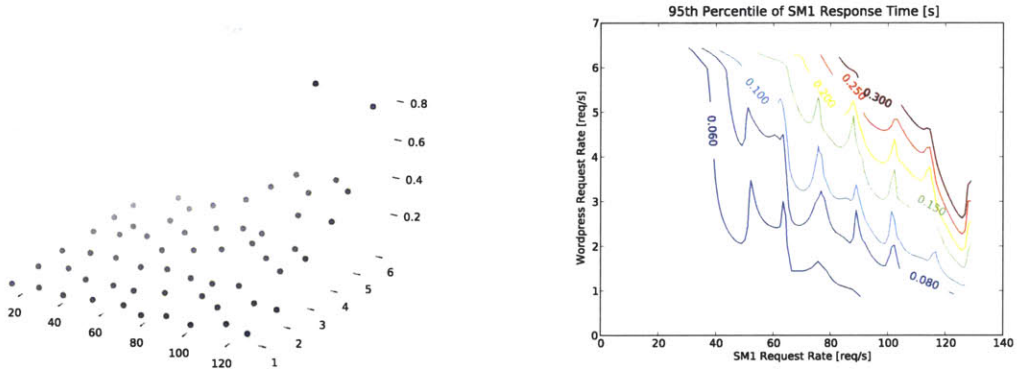
# Models	% of Models	Variable	Meaning
2270	64.8	x_{673}	vm.cpu.ready_summation
1523	43.5	x_{729}	vm.cpu6.ready_summation
1198	34.2	x_{725}	vm.cpu2.ready_summation
814	23.3	x_{724}	vm.cpu1.ready_summation
657	18.8	x_{433}	host.net.vmnic0packetstx_summation
544	15.5	x_{746}	vm.cpu0.usagemhz_minimum
540	15.4	x_{440}	host.cpu.usagemhz_none
531	15.2	x_{301}	host.sys.hostvimresourcememmapped_latest
510	14.6	x_{36}	host.mem.swapused_average
366	10.5	x_{728}	vm.cpu5.ready_summation
344	9.8	x_{490}	host.cpu.utilization_average
333	9.5	x_{730}	vm.cpu0.system_summation
303	8.7	x_{455}	host.cpu1.usage_minimum
235	6.7	x_{722}	vm.cpu0.ready_summation

Table 3.2: Variable frequency table for SM1 models

Different Contention Content

To test generalization of our modeling framework, we also placed different content alongside the SM server.

In the first case, we placed a Wordpress server in a separate VM, with 1000 random posts, each with random chosen tags from 26 categories and up to 8 generated comments. As before, we varied the request rates on both the SM1 server and the Wordpress server. Each of the response times from the SM1 server was collected and synchronized with the full set of ESX performance counters as before. The response time curves are shown in Figures 3-12(a) and 3-12(b).



(a) 3D scatterplot of 95th percentiles of SM1 response time. The axes are the same as in (b)

(b) Contour plot of 95th percentiles of SM1 response time

Figure 3-12: 95th percentile SM1 response times in share-based simulation with Wordpress contention

Using the same model ensembles from before, we use this out of sample data to test prediction performance in untrained scenarios. DataModeler is again used to form predictions from the ensembles, and the results are shown in Figure 3-13.

The prediction error here is noticeably worse, and we have $R^2 = 70.1$. Because the models were trained on only one particular case (the identical SM server alongside), the simulations may not have explored the entire input space. A variable that the models threw out in the previous case may become important in the Wordpress case due to changes in the bottleneck resource, and hence the models may overlook scenarios they were not trained to model. As we can see here, the variance of breakdown regime

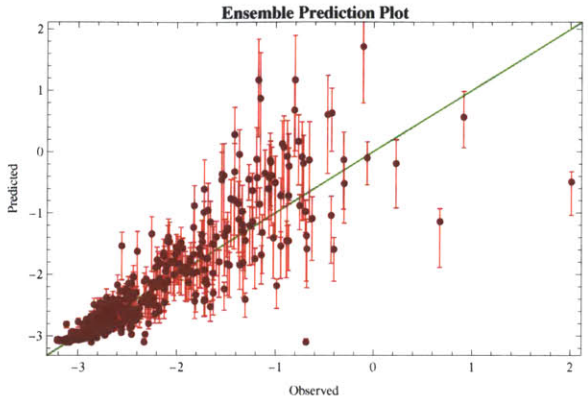


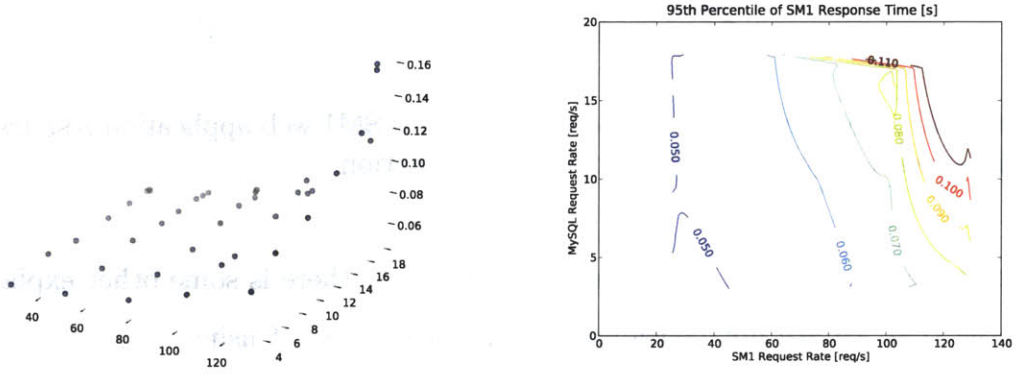
Figure 3-13: Model ensemble prediction results for SM1 web application response time in share-based simulation with Wordpress contention

response time prediction increases, indicating that there is some other explanatory variable that we are not considering. One advantage of using an ensemble-based approach, however, is that the ensemble prediction ranges grow with uncertainty. In many of these data points, the uncertain range actually overlaps the actual response time. The prediction uncertainty would be larger if we included more models, so the error may also be due to our aggressive model selection (boxing) scheme. With a more generous model selection, we would increase robustness in detecting new regimes, at the cost of lower training/test performance.

Despite the high prediction error from a R^2 point of view, the model is still useful even without training with this contending VM. Using a threshold-based SLA definition, the model will still be correct most of the time. For example, if we define the SLA to be a 4x latency increase from the fastest possible response time, our threshold is at 160ms (-1.8 on the log plot). Using a thresholding SLA definition, the prediction of whether or not the system is experience breakdown is actually correct 92.7% of the time. Thus, a model that may not perform terribly well from a machine learning perspective can still provide useful information in practice. Keep in mind that it is making predictions for regimes it has not experienced before, so the fact that it can still predict breakdown is noteworthy.

We also tested the model robustness using a MySQL as the contending content. The SysBench benchmark was used as the content, and the request rate was varied

similar to the previous scenario. All other details are the same, so they are omitted for brevity. Simulation results are shown in Figures 3-14(a) and 3-14(b). Again, we used the same models from the previous section and tested model accuracy on this out of sample data to assess prediction performance in untrained scenarios. The results from this are shown in Figure 3-15.



(a) 3D scatterplot of 95th percentiles of SM1 response time. The axes are the same as in (b)

(b) Contour plot of 95th percentiles of SM1 response time

Figure 3-14: 95th percentile SM1 response times in share-based simulation with MySQL contention

From a R^2 point of view, we achieve 69.1% accuracy. This is acceptable, considering it is a novel scenario with very different contending behavior. However, if we look at the models as a predictor of SLA violation, we achieve 96.6% accuracy. Thus, we can see that the models are quite robust at predicting breakdown, even in regimes that

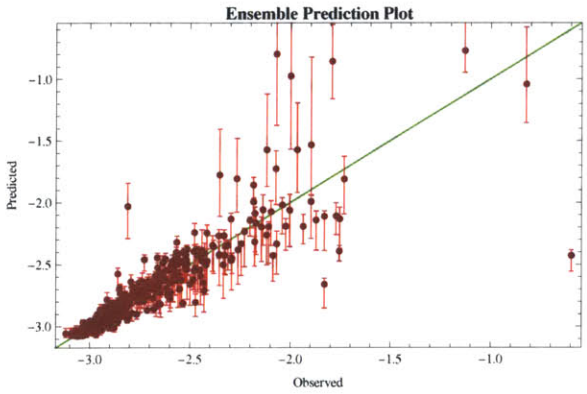


Figure 3-15: Model ensemble prediction results for SM1 web application response time in share-based simulation with MySQL contention

have not been trained for.

3.2 Virtual Desktop Infrastructure Modeling

We also ran an experiment with a VMware ESX-based virtual desktop infrastructure. Thirty Windows 7 desktops were deployed to the ESX host and configured according to the best practices guide [VMware, 2010]. Each host ran a XML-RPC server as an interface to execute commands. This was used rather than a remote desktop based interface so that bandwidth is reduced and guaranteed not to be the bottleneck. It also reduces the computation burden on the stress client, since we can now execute commands directly rather than interacting with a screen image and performing image checksums.

Internally, the VDI server utilizes the Microsoft COM to execute commands on Microsoft Office applications. We used this set of operation sequences:

- **Microsoft Word:**
    1. Launch
    2. Open a large document
    3. Scroll to the bottom
    4. Scroll to the top
    5. Close
    6. Terminate
- **Microsoft Word:**
    1. Launch
    2. Create a new document
    3. Type some sample text (Lorem ipsum...)
    4. Scroll down 30 lines
    5. Scroll up 30 lines
    6. Close
    7. Terminate
- **Microsoft Excel:**
    1. Launch

2. Create a new spreadsheet
3. Input 100 cells of data
4. Scroll down 100 cells
5. Close
6. Terminate

- **Microsoft Powerpoint:**

1. Launch
2. Open a large presentation
3. Page through the slides in edit mode
4. Close
5. Terminate

- **Microsoft Powerpoint:**

1. Launch
2. Open a large presentation
3. Start the presentation and advance through slides
4. Close
5. Terminate

During the run, up to 30 client threads simulate desktop users. Each client connects to a different VM, and executes a series of actions and times the operation times. 30 runs were performed, each with a different number of clients active (i.e. 1, 2, 3, ..., 30 clients). While a desktop is unused, it remains powered on.

To analyze the data, we choose to look at one particular user (say the VM1 user) and look at the 95th percentile of Powerpoint response time. Again, we choose to model the 95th percentile of response time because SLAs are typically defined in terms of 95th percentiles. Results of the simulation are shown in Figures 3-16(a) and 3-16(b). Median response times are also shown in the plots to show that the variance also increases as we increase the number of users; they are not used for modeling, however. Notice that response time is fairly flat up until the 9th user connects, at which point the response times jump and begin to climb, reaching a slowdown factor of 3-4x at 30 users.

We can model these response times using the same GPSR procedure as before. We collected 55 data points from the 30 runs and split it into a 44 point training set and a 11 point test set, and ran the training set through the modeler. The input data is the hypervisor counters from VM1 (the selected user whose response times we are modeling) and from the ESX host.

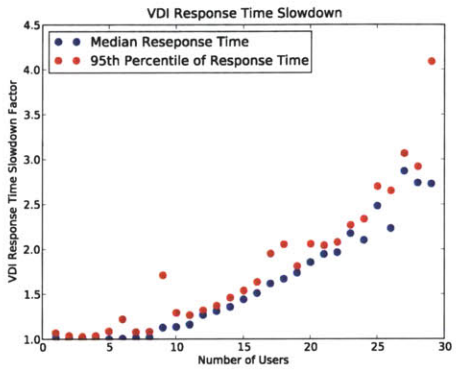
The model results, shown in a Pareto front log plot, are shown in Figures 3-17(a) and 3-17(b), and Pareto front individuals are shown in Table 3.3. Model predictions are shown in Figure 3-18, and test results in Figure 3-19. The results are once again quite good, achieving 96-97% R^2 on the training set with modest complexity. On the test set, the model achieves a 97.63% R^2 value.

It is interesting to note that the most important variables for this VDI simulation, shown in Table 3.4 are very different from the variables used in the SM application (Table 3.2). On the one hand, this is an advantage, because this means the modeling procedure is adaptive and learns specifically how a particular application will perform without any prior assumptions about resource usage or application class. On the other hand, this almost surely means the models will not generalize across application classes given this training procedure.

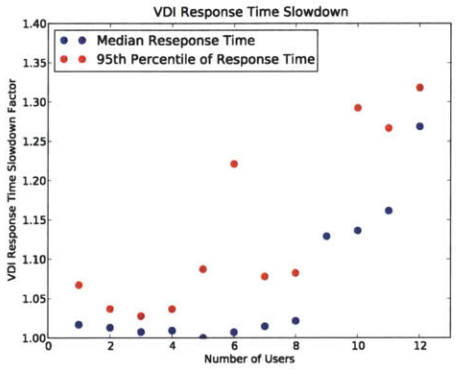
These models were not plugged into a control module. One way in which they could be used is a model could be trained for each application/operation in a representative basket. The models could then produce real time estimates of application performance, and if any of them violate a SLA, action could be taken.

Complexity	$1 - R^2$	Function
23	0.030	$2.097 + \frac{(7.996 \times 10^{-6}\, x_{160}}{\sqrt{x_{663}}}$
35	0.029	$2.072 + \frac{(3.378 \times 10^{7}\, x_{160}}{x_{427}^2 \sqrt{x_{663}}}$
42	0.028	$2.138 + (-8.282 \times 10^{-6}) \left(x_{243} - \frac{x_{160}}{\sqrt{x_{663}}}\right)$
50	0.027	$1.904 + \frac{(-7.162 \times 10^{-6})(-6 - 2\, x_{160} - x_{183} + x_{381} - x_{571})}{\sqrt{x_{663}}}$
63	0.026	$1.900 + \frac{(-7.162 \times 10^{-6})(-2\, x_{160} - x_{183} + x_{381} - x_{571} - x_{663})}{\sqrt{x_{663}}}$
77	0.025	$1.880 + \frac{(14.768 \times 10^{-6})(2\, x_{160} + x_{183} - x_{381} + x_{663} + x_{93})}{x_{427} \sqrt{x_{663}}}$

Table 3.3: Model table for VDI models

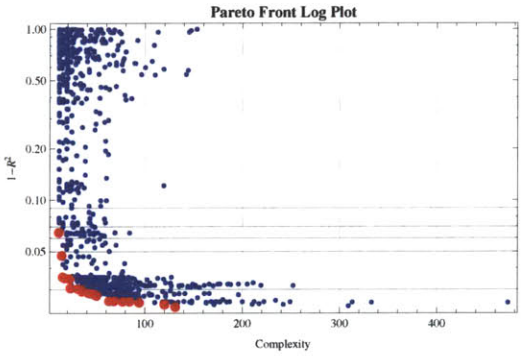


(a) VDI Response Time

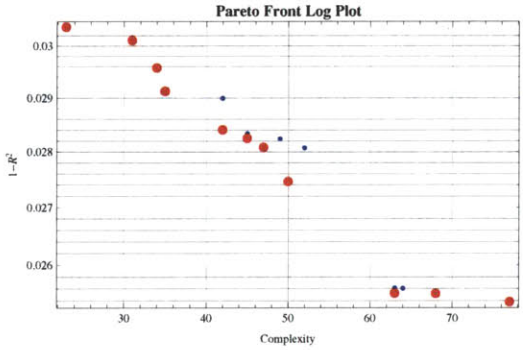


(b) VDI Response Time (zoomed in to the breakdown point)

Figure 3-16: VDI Response Times



(a) Pareto front log plot of GPSR models



(b) Pareto front log plot of GPSR models, boxed around $1 - R^2 = 0.04$ and $C = 80$

Figure 3-17: Pareto front log plots of GPSR models

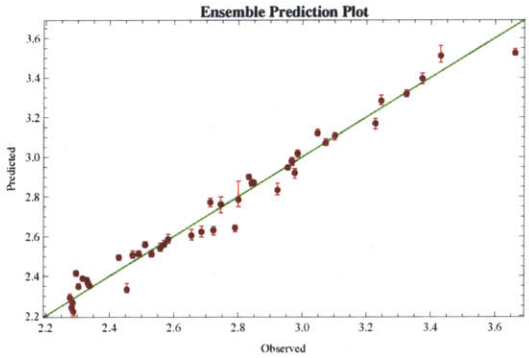


Figure 3-18: Ensemble predictions results in the VDI simulation (training set)

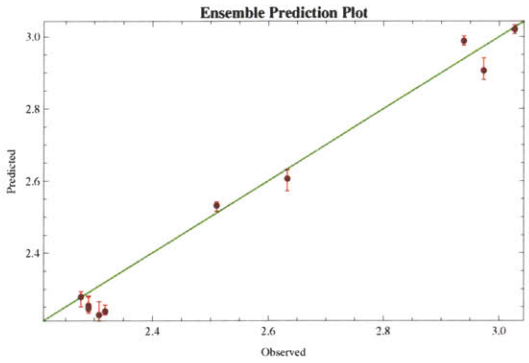


Figure 3-19: Ensemble prediction results in the VDI simulation (test set)

# Models	% of Models	Variable	Meaning
99	90.8	x_{663}	vm.rescpu.runav1_latest
90	82.6	x_{160}	host.mem.active_none
30	27.5	x_{93}	host.cpu9.usage_minimum
24	22.0	x_{381}	host.mem.activewrite_average
20	18.3	x_{84}	host.cpu7.usage_maximum
16	14.7	x_{427}	host.sys.uptime_latest
16	14.7	x_{163}	host.mem.active_maximum
11	10.1	x_{379}	host.sys.hostuserresourcememtouched_latest
11	10.1	x_{183}	host.mem.zero_minimum
10	9.2	x_{327}	host.rescpu.actpk1_latest

Table 3.4: Variable frequency table for VDI models

Chapter 4

Related Work

The growing importance of virtualization in the last few years has attracted substantial research into the modeling and control of virtualized systems. Systems are growing to the point where human management and heuristics are reaching the limits of their scale, and automated management systems are becoming more critical.

4.1 Performance Modeling

In [Stewart and Shen, 2005], modeling and datacenter management is achieved primarily through application profiling. Each VM or application layer is modeled individually to determine the CPU/memory/disk/network usage for a given workload. The modeling is performed offline in the linear (unsaturated) regime, and the assumption is that the placement of the VMs will not violate the resource requirements estimated in this modeling phase. This is similar to the assumptions underlying the VMware Distributed Resource Scheduler (DRS). DRS estimates VM resource requirements based on current resource usage, and the optimizer uses those estimates to consolidate VMs as a bin packing problem (see [VMware, 2006a] for more details).

These works make the assumption that resource oversubscription is to be avoided. However, it has been shown that oversubscription is not necessarily always or strongly detrimental to application performance [Urgaonkar et al., 2009]. There is thus much more flexibility in server consolidation than resource utilization may indicate. Thus, a

SLA-oriented optimization approach can offer room for more aggressive consolidation.

[Tickoo et al., 2010] raises the question of performance effects from unobservables such as cache misses. They propose heuristics to estimate the cache miss rate based on CPU core sharing, and the results seem to indicate that even a relatively simple model can capture much of the information. On a broad scale, this work shows that although unobservable performance counters can impact application performance, they can often be estimated using a model based on observable counters. By using statistical machine learning, one is attempting to capture some of these hidden effects by a data-driven approach rather than relying upon a deep technical understanding of a complicated system (like cache managements) and an indication from the understanding as how to craft the models.

In [Kundu et al., 2010], the objective is to model system performance on a set of benchmarks given a particular set of resource allocations. For each data collection run, they set the CPU caps (CAP), memory allocations (MEM), and IO priority (IONICE), and they collect the number of competing IO operations per second (CDIOPS). The system performance is measured with several benchmarks: SysBench to measure CPU performance, a custom memory benchmark to test memory performance, PostMark to test IO performance, and SysBench to test overall system performance as an OLTP server. Linear/quadratic regressions and artificial neural networks (ANNs) are used to model benchmark performance. To more effectively collect data, an iterative training procedure is used to adaptively collect more data when the modeling error is above a threshold (see Figure 4-1). The best prediction error is achieved by ANNs (see Figure 4-2). This shows that nonlinear models are much better estimators of system performance.

One key difference between our work and this work is the use of benchmarks versus real applications. Benchmarks essentially push the system to saturation with respect to its resources, and assess the best-possible performance using each resource configuration. The focus in [Kundu et al., 2010] is thus modeling the boundary of possible application performance. In many cases, modeling benchmark performance is a close proxy to application performance, but in some scenarios, it is important to model the application

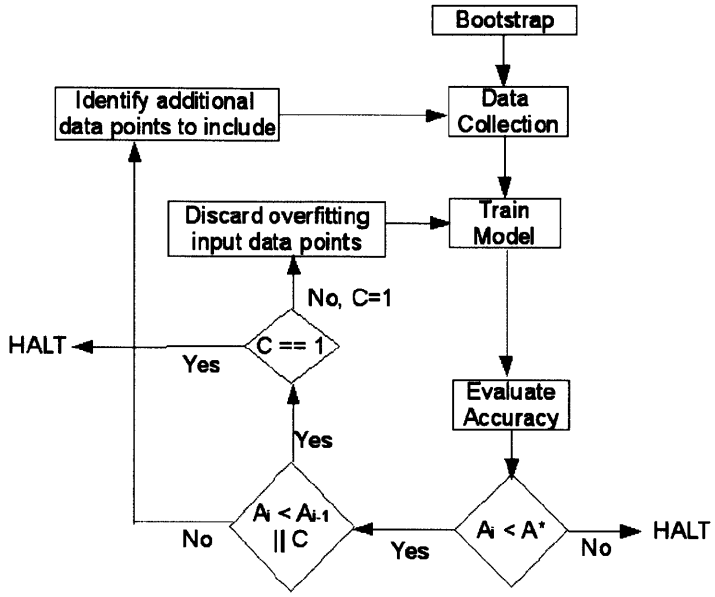


Figure 4-1: Iterative training procedure used in [Kundu et al., 2010] (Figure taken directly from work).

Benchmark	CPU				Memory				Postmark				OLTP			
Modeling Technique	% prediction error				% prediction error				% prediction error				% prediction error			
	avg.	med.	stdev.	90p.	avg.	med.	stdev.	90p.	avg.	med.	stdev.	90p.	avg.	med.	stdev.	90p.
Regress-L	24.90	20.12	20.11	54.88	19.87	20.24	12.98	34.87	6.04	4.73	5.60	11.42	23.95	17.91	19.29	50.54
Regress-LQ	21.69	17.81	17.83	48.88	8.66	6.47	8.12	19.36	6.27	5.09	5.51	11.19	73.51	53.12	74.75	195.49
Regress-LI	21.89	19.35	16.36	49.31	19.80	16.71	14.80	37.19	6.58	5.71	5.53	12.60	71.36	46.31	75.04	213.53
ANN-Linear	18.72	14.02	17.41	46.46	18.57	18.53	12.45	34.43	4.36	3.57	5.25	8.53	24.87	16.72	22.93	57.53
ANN-Gauss	27.42	21.13	23.84	57.16	47.34	46.53	29.73	96.15	25.58	21.00	24.54	33.34	40.00	20.89	48.22	120.04
ANN-Elliot	11.50	6.65	14.02	29.60	2.50	1.16	3.68	6.10	7.31	3.34	10.95	16.24	8.48	4.24	12.36	21.95

Figure 4-2: Prediction error results from [Kundu et al., 2010] (Table taken directly from work).

directly because it is difficult to interpolate generic application performance from benchmark performance. This is especially important when consolidating VMs, because the use of resource sharing instead of hard resource caps makes it harder to translate benchmark performance to application performance. Furthermore, the resource acting as the bottleneck can change as resources demands of other VMs change over time. Thus, the relationship between benchmark performance and application performance is nonlinear and varies with the behavior of the other VMs as well.

Another key difference from our work is that the modeling inputs are the control parameters (allocation levels). In our work and other works in this section, the inputs are usually resource usage, workload, or other performance counters. Modeling using allocations is appropriate for benchmark performance, but in modeling application performance, where the VM may not be at saturation, it is important to also include actual usage and other counters.

4.1.1 Probabilistic Response Time Modeling

Many works on performance modeling estimate the mean (and sometimes variance) of response time. According to [Watson et al., 2010], if we examine the response time distributions, it turns out that they are usually not Gaussian. The response times are lower bounded by the computational requirements of the request. Above this lower bound, response times vary due to instruction queueing and IO waits. Intuitively, very long response times are much more rare than short response times due to the FIFO nature of request queues. Thus, the distribution is usually lower bounded and clustered close to the lower bound, but also heavy-tailed with some chance of very long response times. Queueing theory models the response times with exponential distributions, and this turns out to be a good approximation.

Some works account for the non-normal distribution shapes by modeling medians and percentiles, which have been shown to be more appropriate for queue modeling because they make less assumptions about the shape of the distribution [Bodik et al., 2009b]. From a practical point of view, modeling percentiles is also advantageous since service levels are often provided in terms of the 99th percentile of response time (e.g.

on [Amazon, 2011]).

Another way to tackle skewed distribution is to model the distributions themselves. [Watson et al., 2010] chooses to take this approach and model response time distribution using quantile regression. Quantile regression is a modeling technique that finds the times T_q such that $P[t \leq T_q] = q$ for a set of quantiles q. From these estimates, an empirical CDF estimate can be formed, and from the CDF a PDF can be derived. Because quantile regression models quantiles instead of moments, the absolute value of the residual is used as the loss function. Linear programming is usually used to optimize the parameters. This is more computationally intensive than linear regression with a squared residual loss function, but allows us to model quantiles instead of moments.

The testbed in [Watson et al., 2010] consists of a 3-tiered RUBiS application: one for the web layer, one for the application layer, and one for the database layer. For each resource allocation configuration a, the utilizations u_i, u_j, u_k are measured (one for each application layer) while the application is loaded externally. For simplicity, the variables are all binned into M discrete units. They assume an exponential model of response time versus resource utilization, motivated by queuing theory and performance modeling work in [Bodik, 2010, Bodik et al., 2009b]. Using the fitted exponential model, they construct a CDF and compare that to the experimental CDF using a mean absolute difference loss function. This seems to work very well in practice, yielding low distribution estimation errors.

4.1.2 An SLA-oriented Perspective

It is possible to take a perspective centered on resource allocation and VM placement according to simply fulfilling a Service Level Agreement. This is advantageous as it allows much more flexibility in resource allocation and VM placement. [Turner et al., 2010] develops such an allocation controller. It tries to minimize allocated resources while meeting the SLA. The application used is a multi-tiered implementation of the TPC-W standard. First, an exponential model is used to estimate the effect of CPU contention on response time (see Figure 4-3). In this paper, CPU contention

is defined as "the total CPU utilization minus the amount used by the VM itself." Contention is simulated using a custom Apache server with CPU-intensive requests. For various contention levels, the mean response times then are collected for a range of allocation levels (see Figure 4-4). Since this is a multi-tiered application, there are too many resource contention combinations to exhaustively store a model for each. To effectively overcome this, this work models a small subset of the possible contention combinations, and uses linear interpolation to estimate the other points. After these models are trained, the controller can estimate response time based on allocation and contention levels, and dynamically adjust the allocations to reflect changes in contention while still respecting the SLA. The results indicate that dynamic resource allocation is less wasteful when the contention levels are low, and also more adaptive in high-contention regimes (see Figure 4-5). Indeed, the SLA violations in the static allocation (50% and 10% cases) are mitigated in the SLA-based cases. One point to note is that the resource allocation works well equally for both SLA levels, and more aggressively reduces allocation when the SLA is more lenient. In industry, this could translate to datacenters offering various tiers of SLA, as opposed to various tiers of resource allocation. Not only would this allow for the datacenter to manage resources more aggressively, but the end customer would also get a clearer picture of application performance.

This approach is at odds with our goals because it relies upon modeling the application which we presume is opaque to the ESX host. When the application is complex, more than model will be required.

4.1.3 Online Modeling

Few works directly model application performance online (though many reference this as future work). Online modeling is tricky, because we are often concerned with the breakdown point in the response time curves, but during online runs we prefer to stay below the breakdown point to satisfy the SLA. Extrapolating from models of the unsaturated regime are often unreliable, because the unsaturated, linear regime is a poor predictor of the breakdown behavior. On the one hand, we want to collect data

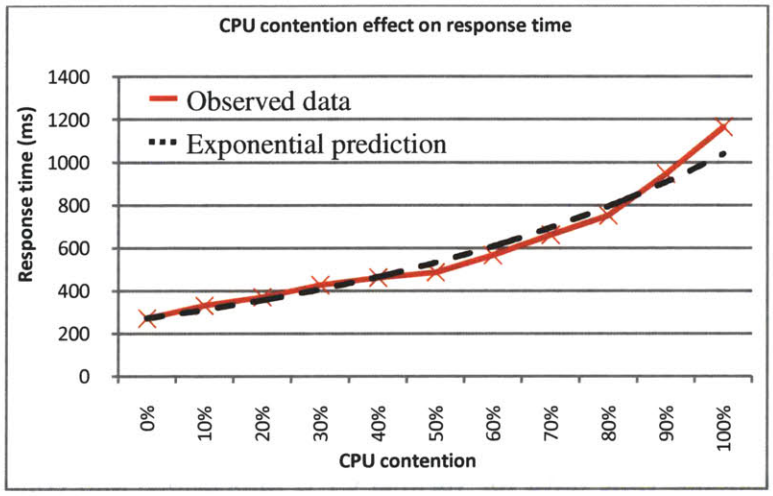


Figure 4-3: Model of response time from CPU contention in [Turner et al., 2010] (Figure taken directly from work).

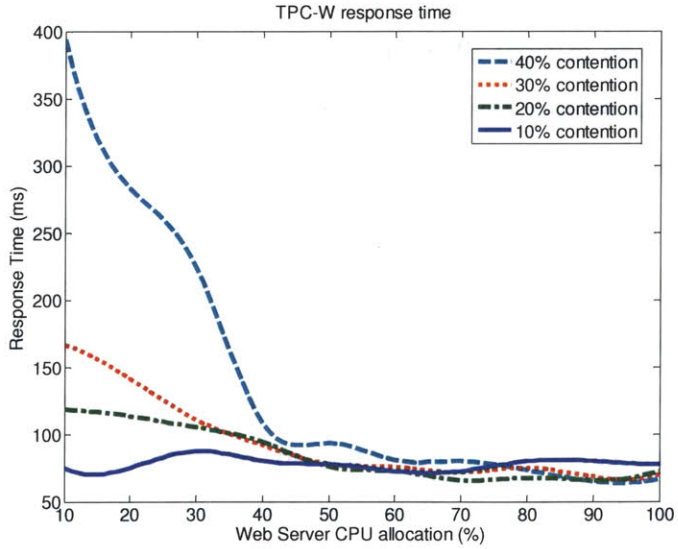


Figure 4-4: Response time data for various CPU allocations and contention levels in [Turner et al., 2010] (Figure taken directly from work).

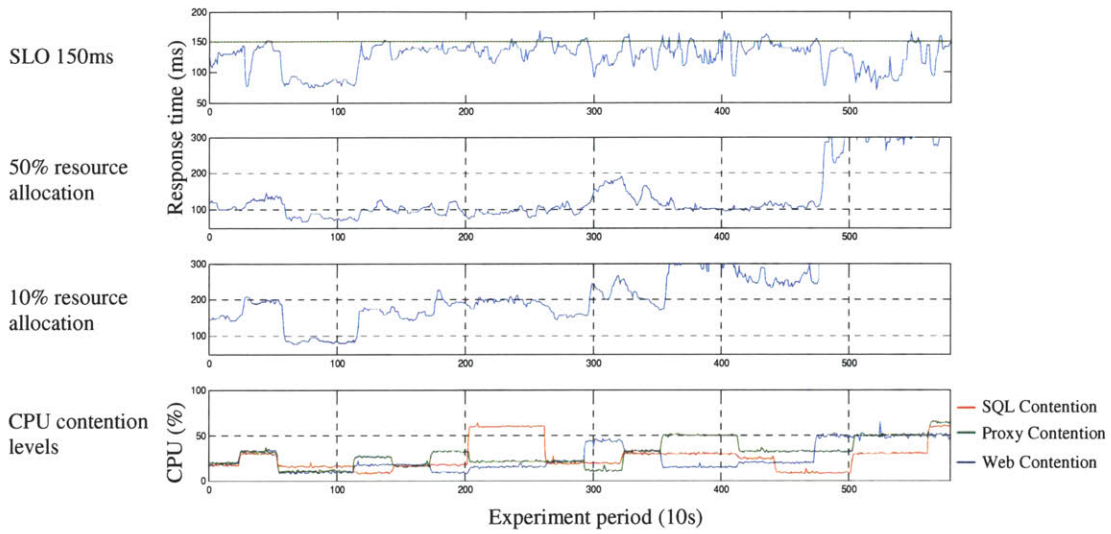


Figure 4-5: Response time data for various CPU allocations and contention levels in [Turner et al., 2010] (Figure taken directly from work).

about breakdown to know how to prevent it. On the other hand, we want to avoid such behavior in an online system.

[Bodik et al., 2009b] proposes one approach to addressing this difficulty. While exploration around the critical SLA violation threshold is happening, hot standby servers are running as backup servers in case response time crosses a safety threshold. A linear model estimates average server throughput and activates safety replicas if the linear model predicts unacceptable performance degradation. More complex models could certainly be plugged into this framework. However, the work focuses on a particular application within a large managed virtual infrastructure (Amazon EC2). In this case, contention is taken to be approximately a constant given the large scale of the infrastructure. Thus, powering on an additional VM essentially scales the throughput linearly and hence linear models are theoretically (and empirically) a good fit. The work focuses primarily on the user side and provisioning VMs in a large managed infrastructure. We focus more on a limited infrastructure attempting to consolidate more aggressively. Nevertheless, the idea of hot standby servers can be translated to a datacenter-side exploration algorithm in future work.

4.2 Datacenter Control

Most of the work thus far has focused on modeling, because many traditional control methods require a model of the system. There are, however, a few notable works on the control side.

4.2.1 VM Placement Control

The coarsest granularity for VM control is VM placement. Even without managing individual resource allocations too much, VM placement can drastically change the VM performance if the VMs are placed effectively. If VMs are placed efficiently, more VMs can be run on the same set of physical hosts and extra hosts can be powered off for power savings.

In [Petrucci et al., 2010], a control optimization algorithm is proposed which structures the VM allocation problem to resemble bin-packing with known application resource usages. The authors demonstrate that their dynamic control saves considerable energy from the normally bundled "Performance" and "On Demand" options on most systems. This work takes the model of application performance (in this case, the required resource allocation to satisfy a given performance objective) as a given and uses it to solve an interesting control problem.

As a special case of VM placement, [Bodik et al., 2009b] considers VM replication. However, the work focuses more on the user side and assumes a large cloud infrastructure capable of expanding without noticeable contention penalties.

4.2.2 Resource Allocation Control

Padala et al. [2009] implements an online controller that dynamically optimizes resource allocation via share manipulation. The application performance is estimated using local linear models around recent performance measurements. The control optimizer minimizes a cost function, which contains a squared term for deviation from target performance and another squared term for control action cost. The results are quite good, reducing overprovisioning when the SLA is lenient and reducing latency

when the SLA is tighter (see Figure 4-6). However, the approach requires looking at response times on line. If the models derived from the approach of this thesis were suffciently accurate to run online, or, if the response rate was recoverable as data without stepping inside the guest VM, and, if share manipulation is not too risky, this approach could be blended for the goals of this thesis.

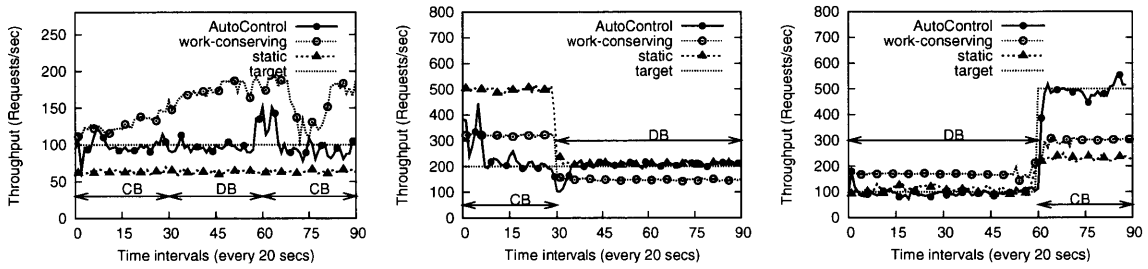


Figure 4-6: Controller results from [Padala et al., 2009] (Figure taken directly from work).

4.2.3 Resource Sharing Optimization

In some virtualization systems, resource shares are used in practice to allow for more efficient resource usage (see Chapter 2). While resource allocations are a good approximation to use for research, a resource sharing-based model would more effectively capture the complexities of the resource sharing used in practice.

In [Ardagna et al., 2010], a toy Apache Tomcat application and SPECweb2005 are modeled using linear parameter varying (LPV) models. LPV models describe systems as

$$x_{k+1} = A_k x_k + B_k u_k$$
$$y_k = C_k x_k + D_k u_k,$$

and the time-varying state space matrices A_k, B_k, C_k, D_k are modeled as affine combinations of constant matrices that are parameter-varying. It is unclear what the exact inputs to the model are (possibly resource counters of some sort), but the response variable is the application response time. The LPV technique is efficient enough to

work as a dynamic controller, and it is also flexible enough to model transients that traditional queueing network models cannot model.

[Wang et al., 2010] also manages VM resource allocations using shares. There are three application priority tiers (gold, silver, and bronze) to designate the relative importance of a VM's operations. One critical part of the infrastucture is a dispatcher that handles all requests and dispatches based on the VM shares and the incoming workload. A quadratic model is trained offline to relate workload to the response time given a CPU allocation. Local controllers then optimize the relative shares in each physical host. Finally, the dispatcher gets the controller decisions and updates its dispatching behavior based on the estimated of performance from the quadratic models. Experimentally, the Trade6 application is used to simulate a trading system, and it is deployed on all three tiers, with a simulated userbase for each tier. The results show that the system can handle changes in request rates and adjust the shares such that each tier continues to meet performance requirements.

4.3 Summary

Much of the virtualization research in recent years has investigated modeling and control as a means for increasing efficiency and scalability. A wide spectrum of modeling classes are represented, and the model inputs range from application specific request rates to application agnostic resource utilizations. In our work, we work towards a modeling system that uses a richer collection of performance counters (all that ESX can provide), using application-specific training in simulation.

Recent work has also shown that shifting towards SLA-based optimization rather than resource utilization-based optimization can lead to higher efficiency without sacrificing performance. We focus primarily on this paradigm because it is potentially the most flexible from the datacenter's point of view, and hence there is more room for cost savings.

Work	Application	RA Method	Model				Control
			Input	Output	Class	Online	
Stewart and Shen [2005]	RUBiS, StockOnline	-	Request Rate	CPU%	Linear	No	No
Kundu et al. [2010]	Benchmarks	Limits	Allocations	Resp. Time	Exp	No	No
Ardagna et al. [2010]	SPECweb2005	Shares	?	Resp. Time	LPV	Yes	No
Watson et al. [2010]	RUBiS	Limits	Utilizations	RT Dist.	QR	No	No
Petrucci et al. [2010]	WC98	Placement	-	-	-	-	Yes
Turner et al. [2010]	TPC-W	Limits	Allocations	Resp. Time	Linear	No	No
Wang et al. [2010]	DVDStore, RUBiS	Shares	Request Rate	Resp. Time	Quad	No	Yes
Bodik et al. [2009b]	Cloudstone	Replicas	Request Rate	Throughput	Linear	Yes	Yes
Padala et al. [2009]	RUBiS, TPC-W	Limits	Utilizations	Resp. Time	LocLin	Yes	Yes

Table 4.1: Summary of related works (see Table 4.2 for abbreviations)

Abbreviation	Meaning
LocLin	Local linear
RA Method	Resource allocation method
Resp. Time	Response time
RT Dist.	Response time distribution
QR	Quantile Regression
Exp	Exponential model
CPU%	CPU Usage Percentage
WC98	World Cup 1998 traces

Table 4.2: Abbreviations used in Table 4.1

Chapter 5

Future Work

Our results demonstrate a promising method of achieving SLA-based resource management. However, our current system is quite application-specific and training-data specific. For example, if our training data has a multi-vCPU VM spending most of its time on a particular CPU core, the models could end up using that particular vCPU's counters, rather than coming up with a general placement-agnostic model.

Genetic programming for symbolic regression works very well for extracting useful information from a deluge of unprocessed data, but in future work, human understanding of the CPU counters could be used to preprocess the data into more meaningful aggregate variables. For example, the simulation may happen to place the VM executions on a particular core, and the model will pick that core out, but in general, it is desirable to have a scheduling-independent model. The variables are also often correlated because many different ways of measuring the same resource consumption exist (e.g. percentage of CPU used vs MHz CPU usage). Principle component analysis (PCA) or hand-picking the variables of the models would also help reduce correlation. This would help the models learn faster by reducing the possible inputs to a smaller set.

Future work would include pursing an application-agnostic model. It would be trained using data from all application types and learns a general model that implicitly determines the type of application from features in the hypervisor counter behavior. The generality of this model would likely prohibit it from be very accurate across all

application types, but it would be robust to application differences.

This performance modeling naturally fits into a broader framework that includes control alongside the modeling. One could imagine a controller that determines VM placement based not on resource contention, but on the constraint that all SLAs must be met. The controller would use the model to predict impending SLA violation or service degradation in advance. Using using this advanced warning, the controller could take action in many ways. One way of mitigating minor SLA violations is to change the relative resource shares between VMs. Our preliminary work has already shown that this feedback mechanism can bring a system in breakdown back down to a level that satisfies the SLA. [Ardagna et al., 2010] also shows that changing relative CPU shares can lead to better overall performance. Another action could be voltage scaling to minimize power consumption when SLAs are fully met. Finally VM migration (i.e. vMotion in the VMware products) could be used for more drastic SLA violations where rebalancing shares will not be sufficient.

Finally, we could consider online modeling. One way to accomplish this is using EB-GPSR as variable selection. The EB-GPSR models would indicate which counters to record, and a more refined model could be trained online using only the selected counters. With a smaller variable set, more efficient modeling techniques could be used to train online models. For instance, one could use gradient descent to fit parameters in arithmetic expressions or structures discerned from the GP models. Genetic programming could be performed online, but usually it requires lots of model sampling, so it is perhaps not the most efficient choice for online learning.

To tie this into the controller framework, one would require a means of capturing response time data in a deployed system (i.e. not collecting data from the client side). One way to do so is to include a module in the hypervisor kernel that can listen to information from inside the VM (e.g. VMware Tools). The system could then be monitored for performance changes like in [Bodik et al., 2009b]. When the system detects that the underlying behavior is no longer stationary, an updated model could be learned online.

Finally, a model-free approach could be used for control. An application that

needs to meet a certain SLA would broadcast a health signal to the kernel based on its performance level, similar to the data that an online modeler would use. The VM could then control VM configurations directly from this signal. This is similar in concept to some work in [Hoffmann et al., 2010], except at the hypervisor level rather than the OS level. This has the drawback that migration is difficult to handle, since the system cannot predict behavior in completely novel environments. However, for minor adjustments to shares, this may be a viable approach.

Chapter 6

Summary

We have shown a fully data-driven approach to virtual application performance modeling. Through simulations, we generated training data for application performance, stress testing the VMs at various performance levels. During these simulations, hypervisor counters were collected as well as service levels (defined as 95th percentiles of response time).

Using EB-GPSR, we learned models of application service level. The full set of hypervisor counters were taken as inputs, and without a priori assumptions of model structure or variable selections, EB-GPSR successfully learned accurate models of application service level. Most test sets exceeded 90% accuracy. Generalization with different contending VM content was more challenging, with only 70% accuracy. However, given a SLA threshold, generalization accuracy rose up to 96%, indicating that the models are still useful in scenarios where an unknown environment is contending for resources.

Our models take inputs from visible hypervisor counters, and do not require direct interaction with the applications to determine response time. Thus we have demonstrated a successful method of modeling application performance from hypervisor-level information. The procedure is highly data driven, and can perform variable selection depending on the application.

Bibliography

Agarwal, A., Santambrogio, M., Wingate, D., and Eastep, J. (2009). Smartlocks: Self-aware synchronization through lock acquisition scheduling. *Strategy*, pages 1–15.

Amazon (2011). Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

Ardagna, D., Tanelli, M., Lovera, M., and Zhang, L. (2010). Black-box performance models for virtualized web service applications. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 153–164, New York, New York, USA. ACM.

Bodik, P. (2010). *Automating Datacenter Operations Using Machine Learning.* PhD thesis.

Bodik, P., Armbrust, M., Canini, K., Fox, A., Jordan, M., and Patterson, D. (2008). A case for adaptive datacenters to conserve energy and improve reliability. *University of California at Berkeley, Tech. Rep. UCB/EECS-2008-127*, (Vm):1–5.

Bodik, P., Goldszmidt, M., Fox, A., Woodard, D. B., and Andersen, H. (2010). Fingerprinting the datacenter: Automated classification of performance crises. In *ACM European Conference on Computer Systems EuroSys*, pages 111–124. Microsoft Research, ACM.

Bodik, P., Griffith, R., Sutton, C., Fox, A., Jordan, M., and Patterson, D. (2009a). Automatic exploration of datacenter performance regimes. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, ACDC '09, pages 1–6. ACM.

Bodik, P., Griffith, R., Sutton, C., Fox, A., Jordan, M., and Patterson, D. (2009b). Statistical machine learning makes automatic control practical for internet datacenters. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, pages 12–12. USENIX Association.

Downey, A. (2005). Lognormal and Pareto distributions in the Internet. *Computer Communications*, 28(7):790–801.

Du, J., Sehrawat, N., and Zwaenepoel, W. (2010). Performance profiling in a virtualized environment. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing*, page 2. USENIX Association.

Eastep, J., Wingate, D., Santambrogio, M., and Agarwal, A. (2010). Smartlocks: lock acquisition scheduling for self-aware synchronization. In *Proceeding of the 7th international conference on Autonomic computing*, ICAC '10, pages 215–224. ACM.

Evolved Analytics (2011). Evolved Analytics' DataModeler. `http://www.evolved-analytics.com/?q=datamodeler`.

Hoffmann, H., Eastep, J., Santambrogio, M., Miller, J., and Agarwal, A. (2010). Application heartbeats for software performance and health. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing*, pages 347–348, New York, New York, USA. ACM.

Hoffmann, H., Misailovic, S., Sidiroglou, S., Agarwal, A., and Rinard, M. (2009). Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures. Technical report.

Keijzer, M. (2003). Improving symbolic regression with interval arithmetic and linear scaling. In *Proceedings of the 6th European conference on Genetic programming*, EuroGP'03, pages 70–82, Berlin, Heidelberg. Springer-Verlag.

Keijzer, M. (2008). Symbolic regression. In *GECCO (Companion)*, pages 2895–2906.

Koenker, R. and Hallock, K. (2001). Quantile regression. *The Journal of Economic Perspectives*, 15(4):143–156.

Kotanchek, M. (2010). Real-world data modeling. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*, GECCO '10, pages 2863–2896, New York, NY, USA. ACM.

Kotanchek, M., Smits, G., and Vladislavleva, E. (2007). Trustable symbolic regression models. In Riolo, R. L., Soule, T., and Worzel, B., editors, *Genetic Programming Theory and Practice V*, Genetic and Evolutionary Computation, chapter 12, pages 203–222. Springer, Ann Arbor.

Kotanchek, M., Smits, G., and Vladislavleva, E. (2008). Exploiting trustable models via pareto GP for targeted data collection. In Riolo, R. L., Soule, T., and Worzel, B., editors, *Genetic Programming Theory and Practice VI*, Genetic and Evolutionary Computation, chapter 10, pages 145–163. Springer, Ann Arbor.

Kundu, S., Rangaswami, R., Dutta, K., and Zhao, M. (2010). Application performance modeling in a virtualized environment. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–10. Ieee.

Netcraft (2011). May 2011 Web Server Survey. `http://news.netcraft.com/archives/2011/05/02/may-2011-web-server-survey.html`.

Padala, P., Hou, K., Shin, K., Zhu, X., Uysal, M., Wang, Z., Singhal, S., and Merchant, A. (2009). Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, number HPL-2008-123, pages 13–26. ACM.

Pedram, M. and Hwang, I. (2010). Power and Performance Modeling in a Virtualized Server System. In *2010 39th International Conference on Parallel Processing Workshops*, pages 520–526. IEEE.

Petrucci, V., Loques, O., and Mossé, D. (2010). A dynamic optimization model for power and performance management of virtualized clusters. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, pages 225–233, New York, New York, USA. ACM.

Sangpetch, A., Turner, A., and Kim, H. (2010). How to tame your VMs: an automated control system for virtualized services. In *Proceedings of the 24th international conference on Large installation system administration*, pages 1–16. USENIX Association.

Sharf, M. (2005). On the response time of the large-scale composite Web services. In *Proceedings of the 19th International Teletraffic Congress (ITC 19), Beijing*. Citeseer.

Smits, G., Kordon, A., Vladislavleva, K., Jordaan, E., and Kotanchek, M. (2005). Variable selection in industrial datasets using pareto genetic programming. In Yu, T., Riolo, R. L., and Worzel, B., editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 6, pages 79–92. Springer, Ann Arbor.

Smits, G. and Kotanchek, M. (2004). Pareto-front exploitation in symbolic regression. In O'Reilly, U.-M., Yu, T., Riolo, R. L., and Worzel, B., editors, *Genetic Programming Theory and Practice II*, chapter 17. Springer, Ann Arbor.

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

Tickoo, O., Iyer, R., Illikkal, R., and Newell, D. (2010). Modeling virtual machine performance: challenges and approaches. *ACM SIGMETRICS Performance Evaluation Review*, 37(3):55–60.

Turner, A., Sangpetch, A., and Kim, H. (2010). Empirical virtual machine models for performance guarantees. In *Proceedings of the 24th international conference on Large installation system administration*, pages 1–15. USENIX Association.

Urgaonkar, B., Shenoy, P., and Roscoe, T. (2009). Resource overbooking and application profiling in a shared Internet hosting platform. *ACM Transactions on Internet Technology*, 9(1):1–45.

Vladislavleva, E. (2008). *Model-based Problem Solving through Symbolic Regression via Pareto Genetic Programming.* PhD thesis, Tilburg University, Tilburg, the Netherlands.

VMware (2006a). Resource management with VMware DRS. Technical report.

VMware (2006b). Virtualization overview. Technical report.

VMware (2009). Understanding Memory Resource Management in VMware ESX. Technical report. `http://www.vmware.com/pdf/usenix_resource_mgmt.pdf`.

VMware (2009). VMware vSphere 4 : The CPU Scheduler in VMware ESX 4. Technical report.

VMware (2010). VMware View Optimization Guide for Windows 7. Technical report.

VMware (2011). vSphere Resource Management Guide. Technical report.

Wang, R. and Kandasamy, N. (2009). A distributed control framework for performance management of virtualized computing environments: some preliminary results. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 7–12. ACM.

Wang, R., Kusic, D., and Kandasamy, N. (2010). A distributed control framework for performance management of virtualized computing environments. In *Proceeding of the 7th international conference on Autonomic computing*, pages 89–98. ACM.

Watson, B. J., Marwah, M., Gmach, D., Chen, Y., Arlitt, M., and Wang, Z. (2010). Probabilistic performance modeling of virtualized resource allocation. In *Proceeding of the 7th international conference on Autonomic computing - ICAC '10*, pages 99–108, New York, New York, USA. ACM Press.