Automatic Exploration of Datacenter Performance Regimes

Peter Bodík, Rean Griffith, Charles Sutton,
Armando Fox, Michael I. Jordan, David A. Patterson
RAD Lab, EECS Department
UC Berkeley
Berkeley, CA

ABSTRACT

Horizontally scalable Internet services present an opportunity to use automatic resource allocation strategies for system management in the datacenter. In most of the previous work, a controller employs a performance model of the system to make decisions about the optimal allocation of resources. However, these models are usually trained offline or on a small-scale deployment and will not accurately capture the performance of the controlled application. To achieve accurate control of the web application, the models need to be trained directly on the production system and adapted to changes in workload and performance of the application. In this paper we propose to train the performance model using an *exploration policy* that quickly collects data from different performance regimes of the application. The goal of our approach for managing the exploration process is to strike a balance between not violating the performance SLAs and the need to collect sufficient data to train an accurate performance model, which requires pushing the system close to its capacity. We show that by using our exploration policy, we can train a performance model of a Web 2.0 application in less than an hour and then immediately use the model in a resource allocation controller.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems; C.5.5 [**Computer System Implementation**]: Servers

General Terms

Experimentation, Measurement, Performance

Keywords

Automatic control, resource allocation

1. INTRODUCTION

Horizontally scalable Internet applications would appear to be a great fit for automatic control, and indeed a growing literature exists on applying models of application performance for resource allocation [5, 6, 13, 14, 7, 11, 3]. Much of this work falls under the framework of *model-based control*, in which first a performance model is trained to map from the workload of the system and the current number of servers to the expected application performance, and then during production, the model is used to select the optimal number of servers to meet the current workload, while meeting the desired application performance.

However, in previous work on model-based control, performance models are trained offline, using data from performance of the application on a small-scale benchmark test bed, from historical performance data, or from application performance under low workload. Offline training of performance models creates serious difficulties in applying model-based control in real-world settings, for two reasons. First, operators of data center applications report that experiments on small-scale test beds do not reflect the capacity of applications in production [1]. Second, the performance profile of Internet applications changes frequently, because of changes in how the application is used, changes in the datacenter environment, and changes in the application itself. Both of these difficulties can be circumvented by training the model *online*, that is, continually retraining the model based on the latest workload and performance data from the production system.

A difficulty of applying online training is the need to gather data from different regimes of workload and system configuration while avoiding violating the Service Level Agreements (SLAs) of the application. Building an accurate performance model can require data from several different regimes of application performance, for example performance under low workload with few machines, under high workload with many machines, and so on. In this paper, we propose a solution to this data-gathering problem, namely, the use of an *exploration policy*. An exploration policy is an automatic control strategy that is specifically designed to collect a broad range of training data for a more complex model-based control strategy, that is, to *explore* different performance regimes. To minimize the probability of violating the SLA, the exploration policy constantly monitors the performance of the application and immediately adds more servers if the latency exceeds the specified *exploration safety threshold*. This idea of "performing experiments in the production environment" is not as radical a departure from current practice as it may seem: it is in fact a common practice to measure system capacity in production, by increasing load on system components in a controlled way [1, 15].

An exploration policy must meet four requirements. First, it must quickly explore different regimes of the application. Second, the amount and type of data collected during exploration must be sufficient to train an accurate performance model. Third, even during the exploration phase, the policy must be careful not to violate performance requirements. And finally, the exploration policy must

ACDC'09, June 19, 2009, Barcelona, Spain.

include rules for automatically switching to model-based control once the performance model is accurate.

We have found that an exploration policy that meets these requirements must satisfy the following four principles. First, *the exploration policy must push the system close to its capacity*, because otherwise we cannot reliably estimate that capacity. However, as a second principle, *exploration must not push the system past its capacity*, because the performance requirements for interactive datacenter applications are strict. Operators will not tolerate an automatic control system that violates performance requirements. The trade off between collecting data close to the system capacity and violating the performance SLA is controlled by the *exploration safety threshold*. Third, *during exploration, the actions themselves should run as quickly as possible*. In our application, we achieve this using a pool of "hot standby" machines that are fully ready to serve requests, but are not yet provisioned. Adding these machines to the application is much faster than requesting and initializing new machines. This makes exploration both safer, because the standby servers can be added quickly in an emergency, and faster, because the exploration controller can see the effects of its actions without a long delay. Finally, the exploration policy needs to quickly estimate an approximate *capacity model* of the application to prevent entering configurations where the system cannot handle the incoming workload. Without such a model, the policy often removes too many servers and causes an SLA violation.

2. RELATED WORK

2.1 Offline exploration

Training of statistical performance models using exploration of the configuration space was previously described in [9, 8, 2]. The authors use *active sampling* to select system configurations to benchmark which leads to fast exploration of most performance regimes of the system. The number of system parameters and their values is often large and exhaustive benchmarking of all possible combinations would take weeks or months. The goal of active sampling is to select a small set of experiments that will provide sufficient data to train an accurate performance model. The benchmarking of the system is performed offline, not in a production environment; selecting a configuration that results in high latency or low throughput of the system thus does not impact the experience of users of the system. In this paper we present exploration that is performed in a production environment where using an incorrect configuration has potentially catastrophic consequences.

2.2 Exploration in Reinforcement Learning

Reinforcement learning algorithms are designed to automatically explore the environment, estimate a control model, and then slowly switch to the *exploitation* mode where this control model is used to achieve a certain objective without further exploration [12]. However, the standard exploration algorithms simply perform *random* actions and do not consider their potentially high cost. Applying these algorithms to our problem would certainly cause too many SLA violations.

3. MODEL-BASED RESOURCE ALLOCATION

We use the following framework for model-based control of horizontally scalable Internet applications with frequently changing performance characteristics consisting of three components. We first train a *statistical performance model* to capture the relationship between workload, number of servers, and the request latency. We propose to train the model in the production environment using exploration and later frequently retrain it to adapt to gradual changes in performance. We use performance models based on smoothing splines or local regression [16] – established techniques for nonlinear regression that do not require specifying the shape of the curve in advance. The model estimates the *fraction of requests slower than the SLA threshold*, given input of the form { workload, # servers } observed over a period of twenty seconds.

The performance model is then used in the *optimal, model-based controller*. The proposed controller first predicts the next five minutes of workload using a linear regression on the most recent 15 minutes. The predicted workload is then used as input to the performance model to find the smallest number of machines that can handle the workload. Finally, the controller employs a hysteresis strategy to avoid oscillations.

To detect abrupt changes in relationship between workload, number of machines, and performance, we use *change point detection* – a technique based on statistical hypothesis testing. The change point detection algorithm compares the current performance of the application relative to an accurate model of the performance a few hours ago. After detecting a change point, the controller enters the exploration phase, quickly trains a new model, and eventually returns to the optimal, model-based control.

4. EXPLORATION CONTROL POLICY

The goal of the exploration control policy is to quickly collect enough data to train an accurate and stable performance model while avoiding the SLA violations. The exploration policy has to cope with an inherent trade off between the safety of the exploration and the "quality" of data being collected. If the policy does not push the system close to its capacity, it will not be able to train an accurate performance model, while running the system above the capacity will likely result in SLA violations.

4.1 Baseline Exploration

To quickly explore a wide range of the application behaviors, the exploration policy sets a random request latency target between 0 and M, the *exploration safety threshold*, and adds or removes machines to reach that target. If the current latency is lower than the selected latency target, the policy removes a single machine from the application, otherwise it adds a machine. The policy sets a new random latency target after adding or removing two machines. The effect of such policy is that it explores both the low-latency and high-latency regimes of the system. To quickly respond to possible future SLA violations, when latency increases above the safety threshold, the policy immediately requests another server for the application. The safety target thus serves two purposes: it ensures that the policy never selects a very high latency target and is also used to trigger a *safety action* in case of emergency.

To collect enough workload and performance data for a given configuration, after removing a machine from the application, the policy waits for D^- minutes before executing another action. After adding a machine, the policy waits for D^+ minutes, which includes the average delay of requesting and initializing a new machine.

4.2 Safety Mechanisms

While experimenting with this exploration policy, we found that it tends to violate the performance SLA and we have thus implemented two safety mechanisms. We maintain a pool of hot standby machines that can be added to the application immediately without waiting for new machines to boot up and we quickly estimate an ap-

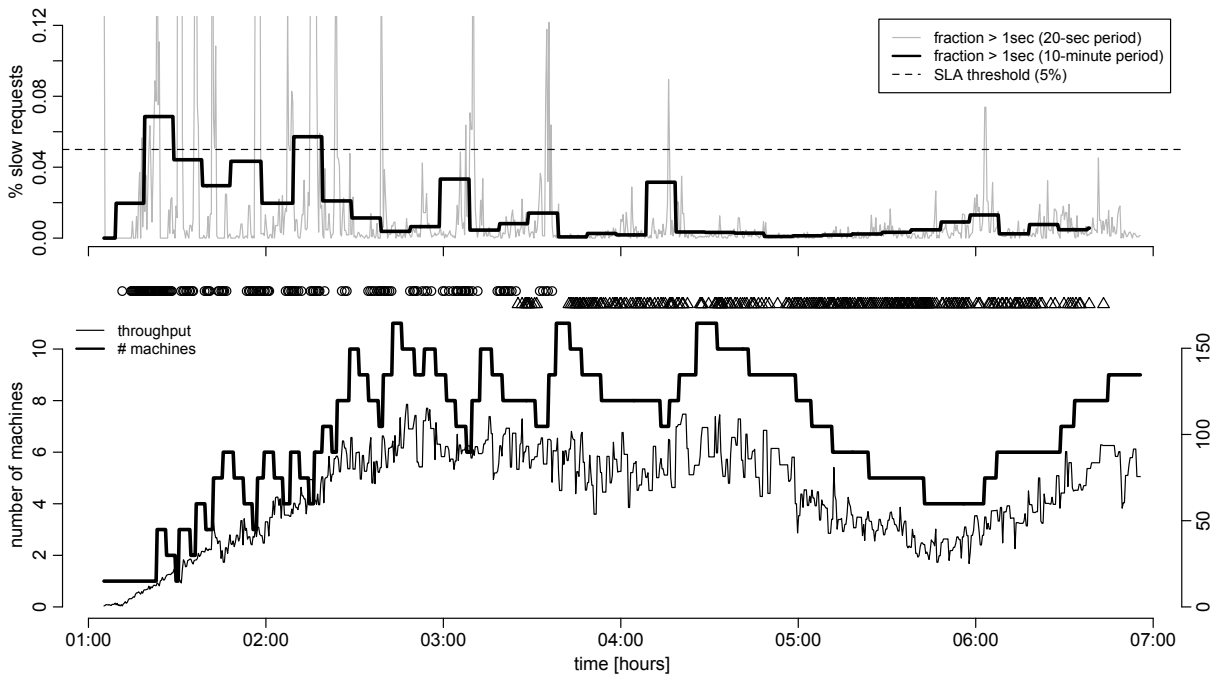

Figure 1: A typical six hour experiment with parameters $M = 5\%$ and $\lambda = 2\%$. In the top graph, the thin gray line represents the fraction of requests slower than 1 second measured every 20 seconds. The thick black line represents the fraction of slow requests over a 10-minute period that corresponds to our SLA; a SLA violation occurs when the black line crosses the dashed horizontal line. There were a total of two SLA violations in this experiment. The symbols in the center differentiate between exploration periods (circles on the top) and optimal control (triangles on the bottom). During period with no symbol, the controller was waiting for the previous action to complete. In the bottom graph, the thin line shows the throughput of the application, the thick line represents the number of running application servers. Notice that during exploration the number of servers changes very rapidly, while during optimal control, the hysteresis smoothes out oscillations of the controller.

proximate capacity of a single machine which prevents the policy from removing too many machines. Finally, we improve the stability of the performance model by discarding data collected during the transient behavior of the application after adding or removing a server.

Machines in hot standby

Booting up a new machine and adding it to the application takes several minutes, which makes the exploration policy very slow to respond to sudden increases in workload. We therefore maintain a small number of hot standby machines that are running the application, but are not serving any requests. When the exploration policy requests a new machine, a hot standby machine is added to the configuration of the load balancer and starts serving application requests within seconds. After adding a machine to the application from the hot standby pool, the exploration policy requests a new hot standby machine to ensure a certain capacity of the pool.

Estimating server throughput using a linear model

The baseline exploration policy often removes a server even when the application is very close to its capacity which causes a significant increase in request latency. To avoid this behavior, we estimate the maximum throughput of a single machine and then derive the minimum number of machines required for the current workload.

Let w_t and n_t be the workload and number of servers at time t, respectively, and let S be the set of time intervals when the latency exceeded the threshold defined in the SLA. The set $T = \{w_t/n_t\}_{t \in S}$ thus represents the workload per server during time intervals when the latency was too high. We approximate the maximum throughput of a single server, $w_{\max}$, as the median of values in T. At time t, the exploration policy will remove a server only if there are more than $\lceil w_t/w_{\max} \rceil$ servers running. We could obtain a more conservative estimate of $w_{\max}$ by computing a lower quantile of values in T, such 10^{th} or 25^{th}. Using the *minimum* of T should be avoided because this statistic is very sensitive to outliers, while quantiles are more robust.

This throughput estimate is also used to compute the number of machines to add as a safety action after detecting a potential SLA violation. Assuming that a machine serving $w_{\max}$ requests per second is at 100% utilization, the safety action aims to decrease the utilization to 75%, or more formally, the safety action adds $\lceil w_t/w_{\max}/0.75 \rceil - n_t$ machines.

Discarding data from transients

During initial testing we observed significant spikes in latency when adding machines to the application. We believe that the increase in latency is due to the new machines being "cold" and thus responding slower than the remaining machines. Because the performance model should model the steady-state performance of the application, we discard data observed within one minute after adding or removing a machine. If used for training, these data points could negatively impact the accuracy of the performance model.

The duration of the these transients could be estimated either offline by changing the number of machines under steady workload and analyzing the spikes in latency, or online by comparing the observed workload to the prediction of an accurate model.

5. SWITCHING TO OPTIMAL CONTROL

As the accuracy of the performance model increases during exploration, the controller has to decide when to switch from the exploration policy to the optimal, model-based control. The controller makes this decision at each time step by evaluating the *stability* of the performance model at the current workload. We first use *bootstrapping* [16] to estimate the variance of the performance model $\mathcal{M}$ at different values of workload w and number of servers n and then use it to decide whether to continue exploring or switch to optimal control.

Bootstrapping is a technique for estimating properties of a model (such as its variance) by fitting the same model to datasets obtained by resampling the original training dataset. The original dataset D is used to create k *resampled* datasets $D_1, \ldots, D_k$; each D_i is the same size as D and contains data points sampled from D with replacement (some data points are selected multiple times, while others are not selected). The variance of the performance model $\mathcal{M}$ for workload w and number of servers n is estimated as the variance of predictions of the k models obtained from the respective resampled datasets at point (w, n). $\mathcal{M}$ is considered stable at (w, n) if the standard deviation (square root of the variance) is less than the *model stability threshold* λ. See Figure 2 for the evolution of model stability during exploration.

We use the following rule to decide whether to explore at the current workload w_{now}. Let n_0 be the smallest n such that the model is stable at (w_{now}, n_0) and $\mathcal{M}(w_{now}, n_0)$ is less than the performance threshold specified in the SLA; in other words, we expect n_0 servers to handle the current workload with good latency. However, to ensure we can rely on predictions of the model for n_0 servers, we also check the stability of the model at points (w_{now}, n_0) through (w_{max}, n_0), where w_{max} is the maximum workload that n_0 servers can handle according to the current model. If the model is not stable at these points, the controller remains in exploration. Otherwise it switches to optimal control and will allocate n_0 servers to the application.

6. EVALUATION

The experiments in this section illustrate the four principles described in the introduction. First, they show that using both a hot standby pool of servers and the simple capacity model is necessary for a exploration policy to avoid SLA violations and quickly train an accurate performance model (see Section 6.1). Second, without pushing the application close to its capacity, the exploration policy does not observe enough data in the high-latency regime of the application and is unable to train an accurate performance model. This situation occurs when the exploration safety threshold M is set to a very low value. Also, setting the safety threshold too high causes a significant number of SLA violations and would thus make the exploration policy unacceptable for production environments (see Section 6.2). Finally, we demonstrate the effects of the model stability threshold λ on the accuracy of the model and the duration of the exploration (see Section 6.3).

Experimental Setup

We evaluate the exploration policy on Amazon EC2 using CloudStone [10], a recently proposed benchmark for Web 2.0 applications. We use a 36 hour workload compressed into six hours obtained from Ebates.com [4]. We assume a cloud computing environment where running a single virtual machine (VM) for ten minutes costs $0.10. We use the local regression library `locfit` in the statistical package R to estimates the *fraction of requests slower than one second* given the current workload and number of servers. During bootstrapping, we found that using $k = 10$ bootstrap samples are sufficient to estimate the stability of the performance model. We used a hot standby pool of two machines. Before executing another action, the baseline exploration policy waits for $D^+ = 7$ minutes after adding a machine and $D^- = 3$ minutes after removing a machine. Requesting and initializing a new machine takes approximately four minutes which are included in D^+. With hot-standby machines, both D^+ and D^- are set to 3 minutes.

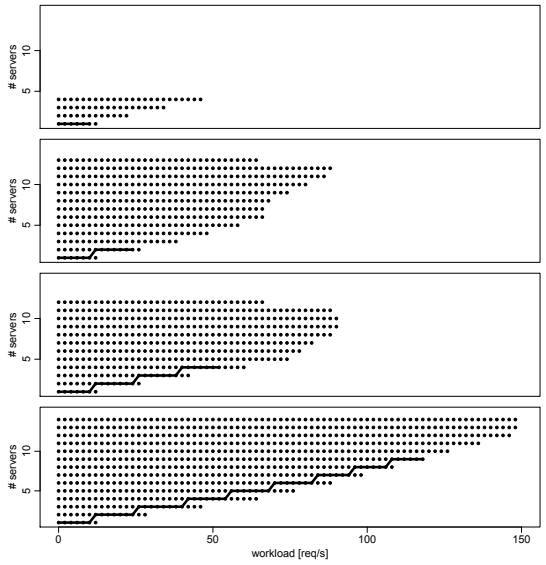

Figure 2: The stability of the performance model after 27, 31, and 64 minutes and at the end of the six hour experiment. In each plot, a dot at workload w and number of servers n means that the performance model is stable at (w, n); i.e. the variance is less than λ. The thick black line represents the number of machines used during optimal control. For workloads where the optimal number of machines is not available, the controller remains in exploration.

We define two performance SLAs: at most 5% of requests should be slower than one second during periods of ten and two minutes. The two-minute SLA is a more stringent criterion, because it is affected even by brief spikes in latency.

A good exploration policy would minimize the number of SLA violations over the ten and two minute periods (*SLA10* and *SLA2* in Tables 1, 2, and 3), the length of the exploration period in minutes (*explor.*), and the total cost of the VMs during the experiment in dollars (*VM cost*). We performed three runs for each setting of the parameters and report the average over these three runs.

6.1 Effects of the safety mechanisms

We first evaluate the baseline exploration policy and compare the effects of the safety mechanisms. We performed four experiments; starting with the baseline exploration policy, then adding hot standby servers, adding the simple capacity model, and finally adding the discarding of data collected during transient behavior. We used exploration safety threshold $M = 0.05$ and model stability threshold $\lambda = 0.02$. The results are summarized in Table 1.

The baseline exploration policy performed worst, regularly violating the SLA and taking too long to converge to an accurate model. Adding hot standby servers decreased the number of SLA violations and the exploration time, while using the simple capacity model reduced the number of SLA violations. As expected, discarding data points from transients helped the model to train faster.

mean

type	SLA10	SLA2	explor.	VM cost
baseline	10.33	40.33	170.8	63.5
+ hot standby	4.33	15.00	100.6	67.6
+ capacity est.	0.50	4.00	124.6	69.2
+ discard tran.	0.33	4.33	94.5	68.6

standard deviation

type	SLA10	SLA2	explor.	VM cost
baseline	5.86	25.93	89.7	1.99
+ hot standby	3.21	5.00	17.5	1.05
+ capacity est.	0.58	3.21	35.5	1.86
+ discard tran.	0.82	2.07	19.3	1.03

Table 1: Comparison of the baseline and exploration policies created by adding the safety measures. The last policy uses hot standby machines, simple capacity estimate, and discards data points during transients. The top table shows the mean over the three experiments, the bottom one shows the standard deviation. The columns, from left to right, represent the type of exploration algorithm, number of SLA violations when averaging over ten and two minutes, duration of the exploration phase in minutes, and total cost of the VMs.

6.2 Effects of the exploration safety threshold

To understand the effect of the exploration safety threshold M on the behavior of the exploration policy we performed a series of experiments with values of M ranging from 0.002 to 0.10 (see Table 2). We used $\lambda = 0.02$.

Small values of M force the exploration policy to add too many servers to achieve the low latency specified by the performance target and thus increase the VM cost. More importantly, the exploration takes too long because the policy never observes the high-latency regime of the application. The policy with $M = 0.01$ performed much better, however, in one of the runs the simple capacity model was very inaccurate because of only a single instance of latency exceeding the SLA threshold. As a consequence, the policy twice added 15 machines as a safety action when a few machines would suffice.

Increasing the value of M allows the policy to push the application closer to its capacity and train the performance model faster. The number of SLA10 violations is very small, while the number of SLA2 violations doesn't increase much even for higher values of M. Values of the safety threshold above the 5% SLA threshold do not automatically mean that the policy will constantly violate the SLA. The SLA is usually evaluated on longer time periods, while the exploration policy can quickly respond to spikes in latency that occur at higher values of M. We found that the SLA is significantly violated only at $M = 0.20$.

6.3 Effects of the model stability threshold λ

The model stability threshold λ affects the length of the exploration and the accuracy of the performance model. We performed experiments with values of λ ranging from 0.005 to 0.5; the results are summarized in Table 3.

Smaller values of λ pose a strict requirement on model stability and require significantly more data points to train the model and thus extend the exploration. Consequently, longer exploration causes more SLA violations and higher VM cost.

As we loosen the restrictions on the stability of the performance model, the exploration period becomes much shorter while also decreasing the number of SLA violations. Somewhat surprisingly, even when λ – the threshold on the standard deviation of the predictions of the model – significantly exceeds the SLA threshold of 5%, the optimal control using this model is still accurate and doesn't incur any additional VM costs nor does it cause more SLA violations.

safety thr. M	SLA10	SLA2	explor.	VM cost
0.002	0.00	0.33	217.8	81.4
0.005	0.00	0.50	269.2	76.3
0.01	0.67	2.67	155.7	73.9
0.03	0.00	5.00	108.1	69.2
0.05	0.33	4.33	94.5	68.6
0.07	0.00	6.00	99.5	68.3
0.10	0.00	6.33	93.6	68.3
0.20	4.00	14.00	99.1	67.8

Table 2: Comparison of exploration policies with different values of the safety threshold M ($\lambda = 0.05$).

λ	SLA10	SLA2	explor.	VM cost
0.005	2.00	11.67	205.1	74.0
0.02	0.33	4.33	94.5	68.6
0.05	0.25	2.75	62.5	67.1
0.10	0.33	3.67	77.2	68.1
0.20	1.00	4.67	43.2	67.3
0.50	0.67	3.00	46.7	68.2

Table 3: Comparison of exploration policies with different values of the model stability λ ($M = 0.05$).

6.4 Setting the parameter values

In practice the values of the safety and model stability thresholds could be adjusted iteratively. One would start with low values of $M = 0.005$ and $\lambda = 0.02$ and then slowly increase both to shorten the duration of the exploration period. In case of a much stricter SLA defined over a shorter time interval, the value of the safety threshold has to be set much lower.

6.5 Exploration at large scale

The exploration policy has two parameters that would have to be adjusted to make it work at scale of thousands of machines. First, the current policy adds or removes a single machine every three minutes to reach the specified performance target. On a large scale, such behavior would be too slow and the policy should instead add or remove a fraction of the current servers (such as 5%). Similarly, the size of the hot standby pool should be a fraction of the current number of servers, such as 10 or 20%.

7. CONCLUSION

In this paper we argue that training a performance model in a production environment is a necessary first step in automatic resource allocation. We present a safe exploration policy that quickly explores different performance regimes of the application and causes very few SLA violations. We show that there is a relatively wide range of values of the safety and stability thresholds for which the exploration policy performs well.

8. REFERENCES

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

[2] S. Babu, N. Borisov, S. Duan, H. Herodotou, and V. Thummala. Automated experiment-driven management of (database) systems. In *HotOS*, 2009.

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

[4] P. Bodík, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. I. Jordan, and D. Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *ICAC*, 2005.

[5] J. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Symposium on Operating Systems Principles (SOSP)*, 2001.

[6] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang. Power and performance management of virtualized computing environments via lookahead control. In *ICAC '08: Proceedings of the 2008 International Conference on Autonomic Computing*, pages 3–12, Washington, DC, USA, 2008. IEEE Computer Society.

[7] X. Liu, J. Heo, L. Sha, and X. Zhu. Adaptive control of multi-tiered web applications using queueing predictor. *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 106–114, April 2006.

[8] P. Shivam, S. Babu, and J. Chase. Active sampling for accelerated learning of performance models. In *SysML*, 2006.

[9] P. Shivam, V. Marupadi, J. Chase, T. Subramaniam, and S. Babu. Cutting corners: Workbench automation for server benchmarking. In *USENIX*, 2008.

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

[11] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *NSDI*, 2005.

[12] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press, March 1998.

[13] G. Tesauro, N. Jong, R. Das, and M. Bennani. A hybrid reinforcement learning aproach to autonomic resource allocation. In *International Conference on Autonomic Computing (ICAC)*, 2006.

[14] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier internet applications. In *ICAC*, 2005.

[15] P. Vosshall. Amazon, Personal communication.

[16] L. Wasserman. *All of Nonparametric Statistics (Springer Texts in Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.