Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters

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

Abstract

Horizontally-scalable Internet services on clusters of commodity computers appear to be a great fit for automatic control: there is a target output (service-level agreement), observed output (actual latency), and gain controller (adjusting the number of servers). Yet few datacenters are automated this way in practice, due in part to well-founded skepticism about whether the simple models often used in the research literature can capture complex real-life workload/performance relationships and keep up with changing conditions that might invalidate the models. We argue that these shortcomings can be fixed by importing modeling, control, and analysis techniques from statistics and machine learning. In particular, we apply rich statistical models of the application's performance, simulation-based methods for finding an optimal control policy, and change-point methods to find abrupt changes in performance. Preliminary results running a Web 2.0 benchmark application driven by real workload traces on Amazon's EC2 cloud show that our method can effectively control the number of servers, even in the face of performance anomalies.

1 Introduction

Most Internet applications have strict performance requirements, such as service level agreements (SLAs) on the 95th percentile of response time. To meet these requirements, applications are designed as much as possible to be *horizontally scalable*, meaning that they can add more servers in the face of larger demand. These additional resources come at a cost, however, and especially given the increasing popularity of utility computing such as Amazon's EC2, applications are incented to minimize resource usage because they incur cost for their incremental resource usage. A natural way to minimize usage while meeting performance requirements is to automatically allocate resources based on the current demand. However, despite a growing body of research on automatic control of Internet applications [2, 11, 6, 5, 4, 10], application operators remain skeptical of such methods, and provisioning is typically performed manually.

In this paper, we argue that the skepticism of datacenter operators is well founded, and is based on two key limitations of previous attempts at automatic provisioning. First, the performance models usually employed, such as linear models and simple queueing models, tend to be so unrealistic that they are not up to the task of controlling complex Internet applications without jeopardizing SLA's. Second, previous attempts at automatic control have failed to demonstrate robustness to changes in the application and its environment over time, including changes in usage patterns, hardware failures, changes in the application, and sharing resources with other applications in a cloud environment.

We claim that both problems can be solved using a suite of modeling, control, and analysis techniques rooted in statistical machine learning (SML). We propose a control framework with three components. First, at the basis of our framework are *rich statistical performance models*, which allow predicting system performance for future configurations and workloads. Second, to find a control policy that minimizes resource usage while maintaining performance, we employ a *control policy simulator* that simulates from the performance model to compare different policies for adding and removing resources. Finally, for robustness we employ *model management* techniques from the SML literature, such as online training and change point detection, to adjust the models when changes are observed in application performance. Importantly, control and model management are model-agnostic: we use generic statistical procedures that allow us to "plug and play" a wide variety of performance models.

Our new contributions are, first, to demonstrate the power of applying established SML techniques for rich models to this problem space, and second, to show how to use these techniques to augment the conventional

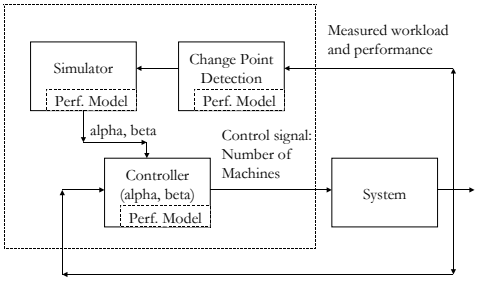

Figure 1: Architectural diagram of the control framework.

closed-loop control framework, making automatic control robust and practical to use in real, highly variable, rapidly-changing Internet applications.

2 Modeling and Model Management

Controllers must adapt to multiple types of variation in the expected performance of a datacenter application. "Expected" variations, such as diurnal or seasonal workload patterns and its effects on performance, can be measured accurately and captured in a historical model. However, unexpected sustained surges ("Slashdot effect") cannot be planned for, and even "planned" events such as bug fixes may have unpredictable effects on the relationship between workload and performance, or may influence the workload mix (e.g., new features). Furthermore, the controller also must handle performance anomalies and resource bottlenecks.

We propose a resource controller (see Figure 1) that uses an accurate performance model of the application to dynamically adjust the resource allocation in response to changes in the workload. We find the optimal control policy in a control policy simulator and adapt the performance model to changes via change-point detection techniques. The control loop is executed every 20 seconds as follows:

Step 1. First, predict the next 5 minutes of workload using a simple linear regression on the most recent 15 minutes. (More sophisticated historical workload models could easily be incorporated here.)

Step 2. Next, the predicted workload is used as input to a *performance model* that estimates the number of servers required to handle the predicted workload (Section 2.1). However, many complex factors affect application performance, such as workload mix, size of the database, or changes to application code. Rather than trying to capture all of these in a single model, we detect when the current performance model no longer accurately models actual performance, and respond by estimating a new model from production data collected through exploration of the configuration space. We explain this *model management* process in Section 2.2.

Step 3. Servers are added or removed according the recommendation of the performance model, which we call s_{target}. To prevent wild oscillations in the controller, we use hysteresis with gains α and β. More formally, we maintain s, a continuous version of the desired number of servers that tracks s_{target} by

$$s_{\text{new}} \leftarrow s_{old} + \begin{cases} \alpha(s_{\text{target}} - s_{\text{old}}) & \text{if } s_{\text{target}} > s_{\text{old}} \\ \beta(s_{\text{target}} - s_{\text{old}}) & \text{otherwise.} \end{cases} \quad (1)$$

Here α and β are *hysteresis parameters* that determine how fast the controller is to add and remove servers. Section 2.3 explains how optimal values of α and β are found using the simulator.

The proposed change-point and simulation techniques are model-agnostic, meaning that they work with most existing choices of statistical performance model; any model that predicts the mean and variance of performance could be used. This makes the proposed framework very flexible; progress in statistical machine learning can be directly applied to modeling, control, or model management without affecting the other components.

2.1 Statistical Performance Models

The performance model estimates the *fraction of requests slower than the SLA threshold*, given input of the form {workload, # servers}. Each point in the training data represents the observed workload, number of servers, and observed performance of the system over a twenty-second interval. We use a performance model based on smoothing splines [3], an established technique for nonlinear regression that does not require specifying the shape of the curve in advance. Using this method, we estimate a curve that directly maps workload and number of servers to mean performance (for an example, see Figure 3). In addition to predicting the mean performance, it is just as important to predict the variance, because this represents our expectation of "typical" performance spikes. After fitting the mean performance, we estimate the variance by computing for each training point the squared difference to the model's predicted mean performance, resulting in one training measurement of the variance. Finally, we fit a nonlinear regression model (in particular, a LOESS regression [3]) that maps the mean performance to variance. This method allows us to capture the important fact that higher workload not only causes a higher mean in performance, but also a higher variance.

2.2 Detecting Changes in Application Performance

Our performance model should be discarded or modified when it no longer accurately captures the *relationship among* workload, number of servers and performance. In practice, this relationship could be altered by software upgrades, transient hardware failures, or other changes in the environment. Note that this is different from detect-

ing changes in performance alone: if the workload increases by 10%, this may cause a performance decrease, but we do not want to flag it as a change point.

The accuracy of a model is usually estimated from the residuals, i.e., the difference between the measured performance of the application and the prediction of the model. Under steady state, the residuals should follow a stationary distribution, thus a shift of the mean or increase of variance of this distribution indicates that the model is no longer accurate and should be updated. Online change-point detection [1] techniques use statistical hypothesis testing to compare the distribution of the residuals in two subsequent time intervals of possibly different lengths, e.g., 9 AM to 9 PM and 9 PM to 11 PM. If the difference between the distributions is statistically significant, we start training a new model. The magnitude of the change will influence the detection time; an abrupt change should be detected within minutes, while it might take days to detect a slow, gradual change.

Because we train the model on-line from production data, rather than from a small-scale test deployment, our approach is under pressure to quickly collect necessary training data for the new model. To address this constraint, we use an active exploration policy until the performance model settles. In *exploration mode*, the controller is very conservative about the number of machines required to handle the current workload; it starts with a large number of machines to guarantee good performance of the application and then slowly removes machines to find the minimum required for the current workload level. As the accuracy of the performance model improves, the controller switches from exploration to optimal control.

2.3 Control Policy Simulator

An accurate performance model alone doesn't guarantee good performance of the control policy in the production environment because the various parameters in the control loop, such as the hysteresis gains α and β, significantly affect the control. This is a standard problem in control theory called *gain scheduling*, however, it is very difficult to find the optimal values of the parameters in complex control domains such as ours because of delays in actions and different time scales used in the controller and the cost functions. To solve this problem, we use Pegasus [8], a policy search algorithm that compares different control policies using simulation. We use a coarse-grained simulator of the application to evaluate various values of α and β parameters using the workload and performance models. The simulator uses real, production workloads as observed in the past several days or any other historic workloads such as spikes. Given particular values of the α and β parameters, the simulator executes the control policy on the supplied workloads, estimates performance of the application using the performance model, and returns the amount of money spent on machines and the number of SLA violations. By using a local search heuristic such as hill-climbing, we can find the optimal values of α and β that minimize the total cost of running the application.

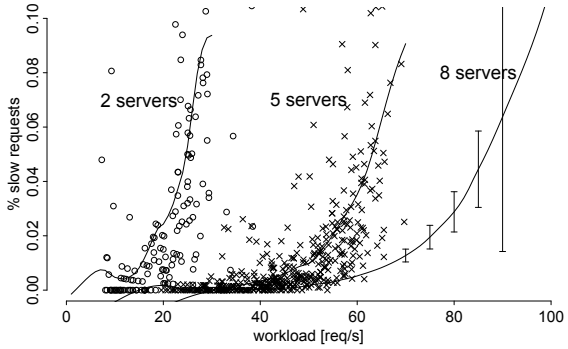

Figure 3: The smoothing spline performance model estimated from observed data; the circles and crosses represent observed data for two and five servers, respectively. Each curve represents the mean performance estimated by the model. The error bars in the eight-servers line represent our estimate of the standard deviation of performance.

3 Preliminary Results

In this section we present early results demonstrating our approach to automatic resource allocation. We first present a resource management experiment using a smoothing-splines based performance model as described in Section 2.1. Next, we demonstrate that the simulator is effective for selecting the optimal value of the β hysteresis parameter. Finally, we use change-point detection techniques to identify a change in performance during a performance anomaly.

In all of the experiments, we used the Cloudstone Web 2.0 benchmark [9], written in Ruby on Rails and deployed on Amazon EC2. Cloudstone includes a workload generator called Faban [7], which we use to replay three days of real workload data obtained from Ebates.com. We compress the trace playback time to 12 hours for the sake of efficiency.

3.1 Automatic Resource Allocation

First we demonstrate that an SML performance model can be used for automatic resource allocation.[1] We model the business cost of violating an SLA as a \$10 penalty for each 10-minute interval in which the 95^{th} percentile of latency exceeds one second. We trained an initial performance model using data from an offline

[1] Although Amazon EC2 costs 10 cents per server per hour, we adjust this to 10 cents per 10 minutes to account for our faster replay of the workload.

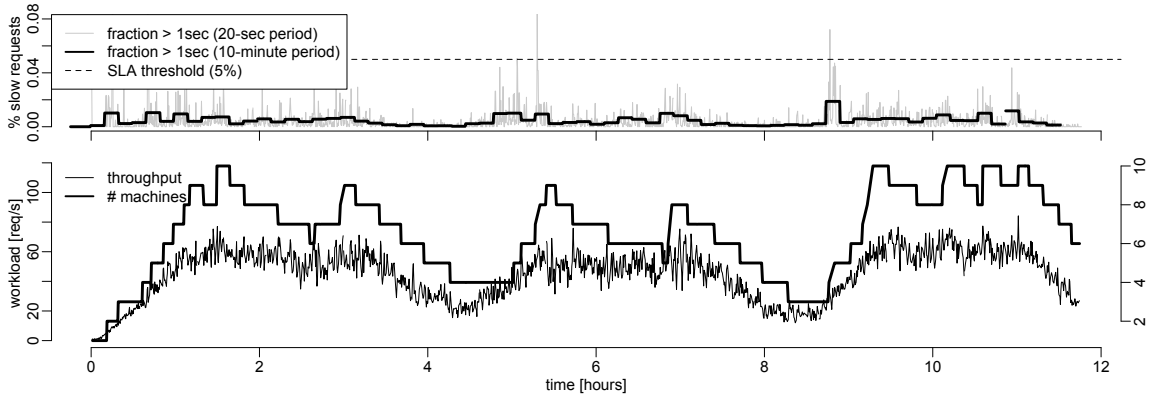

Figure 2: Results of a 12 hour long experiment replaying a three-day workload. The bottom graph shows the workload and the number of servers requested by the controller (thick, step curve). The top graph shows the fraction of requests slower than one second during 20-second intervals (thin, gray line) and during the ten-minute SLA evaluation intervals (thick, step curve). Because the fraction of slow requests during the ten-minute intervals is always below 5%, the SLA was not violated. During the whole experiment 0.52% of the requests were slower than one second. The controller performed 55 actions.

benchmark and used it to derive the relationship between the workload and the optimal number of servers. (We leave online training of the performance model to future work.) The controller attempts to minimize the total cost in dollars that combines a cost for hardware with a penalty for violating the SLA. We set $\alpha = 0.9$ to quickly respond to workload spikes and $\beta = 0.01$ to be very conservative when removing machines. Figure 2 shows that these choices led to a successful run, with no SLA violations and few controller actions to modify the number of servers. However, as Figure 4 shows, the controller is very sensitive to the values of α and β, so we next describe how we use a simulator to automatically find optimal values for these parameters.

3.2 Control Policy Simulator

In this section, we demonstrate using the control policy simulator (Section 2.3) to find the optimal value for the parameters of the controller. In this experiment, we use the simulator to find the optimal value of β, the hysteresis gain when removing machines, while keeping $\alpha = 0.9$. For each value of β, we executed ten simulations and computed the average total cost of that control policy (dashed line in Figure 4). The minimum average total cost of $78.05 was achieved with $\beta = 0.01$. To validate this result, we measured the actual performance of the application on EC2 with the same workloads and the same values of β (solid line in Figure 4). The results align almost perfectly, confirming that the simulator finds the optimal value of β.

3.3 Detecting Changes in Performance

In this section we demonstrate the use of change-point detection on a performance anomaly that we observed while running Cloudstone on Amazon's EC2. Although

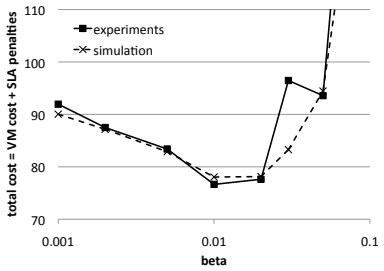

Figure 4: Comparison of the total cost for different values of the β parameter. The solid line represents actual measurements, the dashed line represents the average simulated values (Section 2.3). The simulation runs for all values of β were completed in less than an hour.

the workload and the control parameters were identical to the experiment described in Figure 2, we observed a three hour long performance anomaly during which the percentage of slow requests significantly increased (hours 6 to 8 in the top graph of Figure 5). The result of the change-point test for each t is a p-value; the lower the p-value, the higher the probability of a significant change in the mean of the normalized performance signal. The bottom graph on Figure 5 shows the computed p-values in logarithmic scale; the dips do indeed correspond to the beginning and the end of the performance anomaly period. Furthermore, outside of the performance anomaly, the p-values remain flat. This result shows promise that the dips in p-value could indeed be used in practice to direct model management.

4 Related Work

Most previous work in dynamic provisioning for Web applications uses analytic performance models, such as

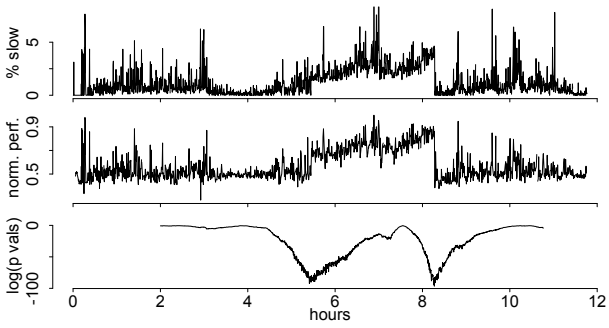

Figure 5: Top graph: performance of the application during a 12-hour experiment with a performance anomaly during hours six to eight. Center graph: observed performance normalized using the predictions of the model. Bottom graph: p-values reported by the hypothesis test.

queueing models, and doesn't consider adaptation to changes in the environment. In contrast, our statistical models are numerical in character, which allows us to more naturally employ statistical techniques for optimizing control parameters and for model management.

Muse [2] uses a control strategy that adds, removes, powers down or reassigns servers to maximize energy efficiency subject to SLA constraints on quality of service for each application in a co-hosted services scenario. Muse assumes that each application already has a utility function that expresses the monetary value of additional resources, which includes the monetary value of improved performance. In our work, we learn the performance impact of additional resources.

In [6] the authors devise an adaptive admission-control strategy for a 3-tier web application using a simple queuing model (a single M/G1/1/Processor Sharing queue) and a proportional integral (PI) controller. However, a single queue cannot model bottleneck effects, e.g., when additional application severs no longer help, that a statistical model can incorporate naturally.

[11] uses a more complex analytic performance model of the system (a network of G/G/1 queues) for resource allocation. [5] presents a controller for virtual machine consolidation based on a simple performance model and *lookahead control* – similar to our simulator. [10] applies reinforcement learning for training a resource allocation controller from traces of another controller and improves its performance. The system learns a direct relationship between observations and actions; however, because we model the application performance explicitly, our approach is more modular, interpretable, and allows us to simulate hypothetical future workloads.

[4] provides an example of using change point detection in the design of a thread-pool controller to adapt to changes in concurrency levels and workloads. In our control strategy, change-point detection is used to indicate when we need to modify our performance model.

5 Conclusion

We have demonstrated that the perceived shortcomings of automating datacenters using closed-loop control can be addressed by replacing simple techniques of modeling and model management with more sophisticated techniques imported from statistical machine learning. A key goal of our framework and methodology is enabling the rapid uptake of further SML advances into this domain. We are encouraged by the possibility of increased interaction among the research communities of control theory, machine learning, and systems.

References

[1] M. Basseville and I. V. Nikiforov. *Detectiong of Abrupt Changes*. Prentice Hall, 1993.

[2] 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.

[3] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Springer, August 2001.

[4] J. L. Hellerstein, V. Morrison, and E. Eilebrecht. Optimizing concurrency levels in the .net threadpool: A case study of controller design and implementation. In *Feedback Control Implementation and Design in Computing Systems and Networks*, 2008.

[5] 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.

[6] 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.

[7] S. Microsystems. Next generation benchmark development/runtime infrastructure. http://faban.sunsource.net/, 2008.

[8] A. Y. Ng and M. I. Jordan. Pegasus: A policy search method for large mdps and pomdps. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 406–415, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[9] 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.

[10] 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.

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