DRAPE: Dynamic Resource Allocation for Power Efficiency

Kevin Canini
kevin@cs.berkeley.edu

with Peter Bodík and Michael Armbrust
Advised by David Patterson and Armando Fox

CS 294-10: Practical Machine Learning
Final Project Report

December 12, 2006

1 Introduction

Electricity consumption is a major expense of today's datacenter operations, and the need for air conditioning to remove the resulting buildup of heat is a bottleneck for full utilization of physical space. Energy concerns like power efficiency have become as important as performance and reliability for maintaining a successful computing center. Recognizing the need for more attention in this area, we have built a resource allocation system that dynamically redistributes available computing resources in order to minimize power consumption while maintaining a specified service level agreement.

DRAPE takes advantage of two key properties of typical datacenters. First, datacenters must necessarily be provisioned for the maximum possible load they are expected to handle, while actual workloads tend to be periodic in nature, especially for web-based applications. In practice, this means that the average workload handled by the datacenter is much lower than the maximum, and therefore the average utilization is low as well. According to anecdotal data, the average utilization is around 10% to 20% of a datacenter's computing resources.

Second, there is a large difference in power consumption between an powered-up but idle sever and a powered-down server. This means that temporarily powering down idle servers could eliminate an enormous amount of unnecessary electricity consumption.

Previous work in load-balancing has mostly focused on maximizing some performance metric. However, as performance increases beyond a certain point, the additional returns (measured in user satisfaction, revenue, or otherwise) are drastically diminished. For example, a user will generally be just as satisfied if a search engine query takes 10ms or 100ms to finish. Therefore, we tried to minimize power consumption while maintaining a specified service level agreement (SLA). We semi-arbitrarily chose our SLA to be that at least 90% of the requests in any given minute will be satisfied within 100ms.

2 Problem Description

Our goal was to create a system to manage the computing resources of a server cluster to minimize power consumption without violating the SLA described above. At first,

we envisioned a reinforcement learning algorithm that could allocate resources between different applications running in the same physical server, migrate applications between servers, and shut down unused servers to minimize overall power consumption. However, due to the limited time we had to complete the project and the large overhead of setting up all the architecture, we settled on the simpler problem of dynamically switching on and off servers that were dedicated to a single application, leaving the more interesting versions of the problem as future work.

We made the simplifying assumption that the power consumption of each machine was binary, i.e., it consumed a constant amount of power when it was turned on, and zero power when it was turned off. While this isn't really true, the resulting policy decisions probably aren't affected by the difference. Ongoing RAD Lab research is exploring this topic in more detail.

The hard part of this problem is that it takes time to turn servers on and off. During this delay, the workload can change dramatically, so it's important to be able to predict workload for a short time into the future. We didn't actually turn servers on and off (other people using the machines might have objected!), but we simulated a 2-minute delay between the time a server was told to be turned on and the time it started servicing requests.

2.1 Environment

The environment we consider, depicted in Figure 1, is a server cluster consisting of 6 physical servers, each having a dual-core 1GHz CPU and running VMWare ESX server [10]. The virtual machines made it easy to suspend a machine's state when it needed to be shut down. Each VM is limited to 1500MHz and is running a copy of Apache web server and PHP dynamic content loader. The cluster is load-balanced in software by HAProxy [6], which can change the workload weights of each VM on the fly. The requests are generated by our homemade workload generator, which is capable of playing back a trace of requests from a log file or generating requests according to any of several pre-defined functions, e.g. constant workload and sinewave workload.

Statistics about the workload and the response times are aggregated and reported by HAProxy, and statistics about server resource utilization, e.g. CPU load and disk activity, are reported by the VM monitoring software. This information is passed to the policy engine, which is able to effect changes in the load balancing weights, change virtual machine CPU allocations, and turn on and off virtual machines entirely.

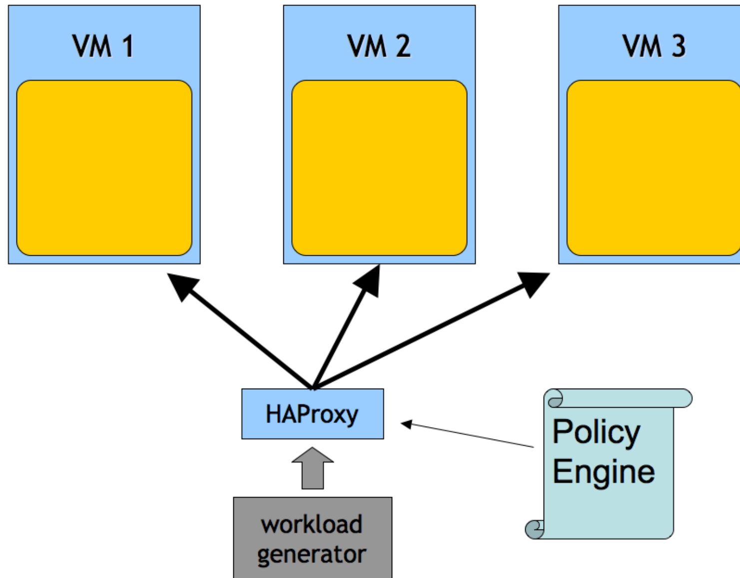

Figure 1: The environment consists of multiple servers, each running VMWare ESX server, which handle requests assigned by the HAProxy load balancer.

2.2 Workload

The workload we used was based on the 1998 World Cup trace [1]. This freely-available resource allowed us to prove that our system could handle the eccentricities of real-world situations. The trace is 98% static content, so we could recreate the behavior of the web site almost exactly.

For reference, a plot of the requests per second over the period from day 60 to day 67 of the World Cup trace is shown in Figure 2.

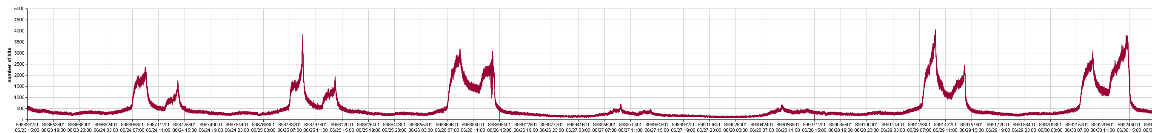

Figure 2: A 7-day sample of the 1998 World Cup Web Site trace, which the majority of our test cases was based on.

In addition to using the World Cup site, we wanted to test our system for robustness by running it on a variety of other workloads. To verify its steady-state behavior,

we used on a constant workload. To test its ability to adapt to quickly changing conditions, we used a sine wave with a period of only a few minutes. We also altered the World Cup site by increasing all the file sizes by 10 and by forcing the pages to be generated by PHP.

2.3 Input/Output

The decision-maker of DRAPE is the policy engine. It takes the following inputs every 20 seconds:

- Number of incoming requests
- Average CPU usage of each VM
- 90th percentile response time

Each input is a summary of the last 20 seconds' worth of data.

At each time step of 20 seconds, the policy engine is responsible for deciding how many VMs should be turned on. Our monitoring framework receives this decision, signals the appropriate VMs to turn on or off, and updates the HAProxy configuration to start sending requests to the new set of active VMs.

3 Solution

DRAPE is a heuristic-based solution using a predictive model of the workload and an adaptive model of the resource utilization of the system as a function of the workload. At each time step, the predicted request rate R is calculated, along with the threshold T, which is an estimate of the number of requests each machine can handle in one second. Then the number of needed servers is simply $N = \lceil R/T + b \rceil$, where b is a buffer which allows the system to err on the side of safety. We hard-coded the buffer value to 0.1, which was optimized on a small training set from the 1998 World Cup data (days 60-67, shown above), using a simplified Matlab simulation of the real environment.

3.1 Predictive Workload Model

With each new set of inputs, DRAPE incorporates the new number of incoming requests into its data and produces a prediction of the rate of requests at two minutes in the future. There are three types of prediction models we used.

1. Constant prediction - First, we used the simplest model that could possibly work: the predicted request rate is equal to the current request rate. That is, $R = C$, where C is the current request rate.
2. Linear regression - Next, in an effort to predict upward slopes, we performed a linear regression on the past w data points and extrapolated the data into the future. That is, $R = LinReg_w$.
3. Positive linear regression - Finally, we used the linear regression result to predict upward slopes, but ignored any predicted downward slopes. That is, $R = \min(C, LinReg_w)$.

For models involving a linear regression, we hard-coded the regression window to be the last 5 minutes of data. This choice was optimized on a 7-day training set from the 1998 World Cup data.

3.2 Adaptive Resource Model

The resource model is responsible for calculating the threshold value T in the above equation. Essentially, this is an estimate of the number of requests that a single machine can handle without violating the SLA. We started by choosing a reasonable value (500 requests per server per second) based on preliminary experiments. However, since this number depends heavily on the speed of the servers and the nature of the requests, it makes much more sense to dynamically calculate it based on previous data points. Figure 3 shows a plot of the average CPU utilization and the 90th percentile response time vs. number of requests per second per server. As you can see, the CPU utilization for a given request profile (small, static web pages, in this case) is clustered tightly around a straight line fixed at the origin. Taking advantage of this observation, our threshold calculation is based on a linear regression of the CPU data, fixed at the origin.

In order to conservatively calculate the threshold without risking overloading any particular server, we use the number of requests per second per server corresponding to an 80% CPU load on the regression line (the blue vertical line). As a fail-safe, in

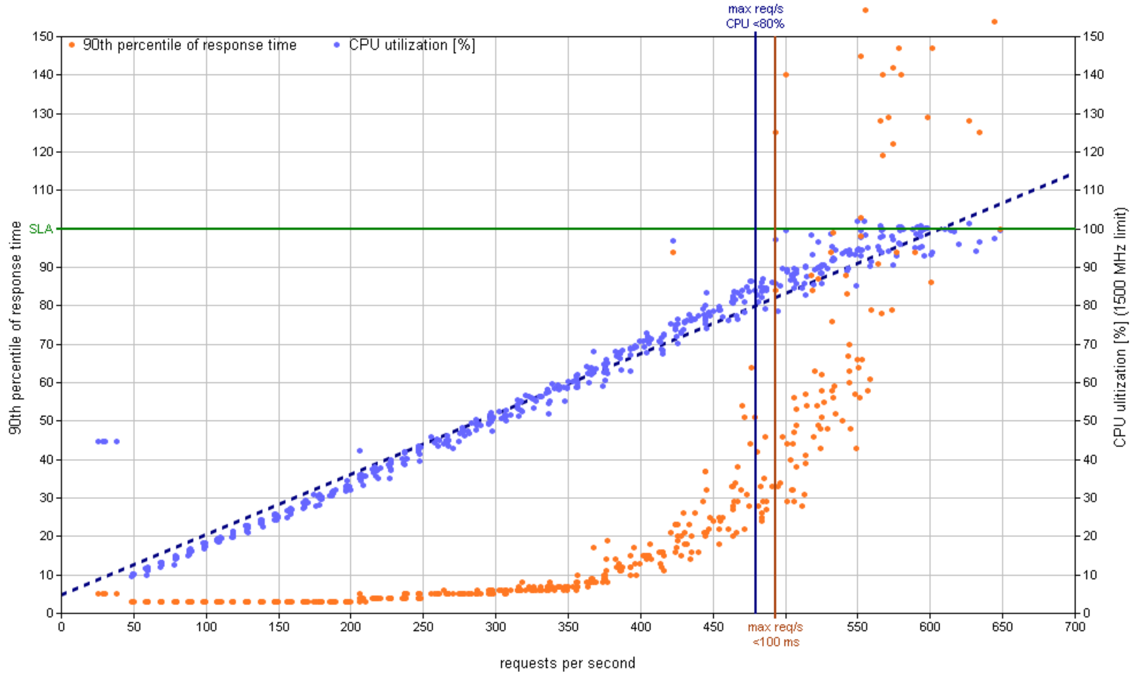

Figure 3: An example plot of average CPU utilization (blue dots) and 90th percentile response time (orange dots) vs. requests per second per server. This data is used to calculate the threshold value T.

case the 90th percentile response time ever violates the SLA, the threshold for the next 15 minutes will be lowered (if necessary) to the number of requests per second per machine at the time of the violation (the orange vertical line).

3.3 Hysteresis Behavior

Noticing that all of the above models lead to severe thrashing behavior, i.e., a policy where machines were turned on and off very rapidly in response to a workload with small variance, we designed an additional component to stabilize the policy. We changed the calculation of N from $N = \lceil R/T + b \rceil$ to use a two-valued hysteresis function. Two different values of b were used, $b_{low} < b_{high}$. The policy is that if $N < N_{low} = \lceil R/T + b_{low} \rceil$, then we set $N = N_{low}$. Otherwise, if $N > N_{high} = \lceil R/T + b_{high} \rceil$, then we set $N = N_{high}$. Based on a small set of experiments, we determined that values of 0.1 and 0.6 lead to good behavior without sacrificing power efficiency.

4 Results

4.1 Matlab Simulation

Table 1 summarizes the results of our various models in terms of requests served, power savings, and thrashing. These results are based on a simplified version of the server cluster that we simulated in Matlab. Since we were working with experiments that ranged over several days, this was the only quick way to test and refine our algorithms and optimize our parameters.

	Requests served	Power savings	Policy changes
Simple model...	0.9986	0.8131	756
...with 0.1-server buffer	0.9995	0.8086	793
...and workload prediction	0.9998	0.8069	635
...and hysteresis	0.9999	0.7998	155

Table 1: The simulation results of four different models of increasing complexity on the training data (1998 World Cup, days 60-67).

Each feature we added to the model made it more conservative in its resource allocation, so the number of requests served goes up at each step. The power savings drops accordingly, but this is a necessary loss in order to service all of the requests. The thrashing behavior is an interesting aspect of these results. Although not explicitly stated in our problem statement, we believe that limiting the number of policy changes (the number of times a server is either turned on or off) is very important. The process of shutting down a machine and booting it back up is very taxing and degrades its expected lifetime. Without a cost model of electricity and machine replacement and a life expectancy model for the machines, we were unable to incorporate this constraint formally into our optimization problem, but we include it in our results nonetheless.

4.2 Static 1998 World Cup

Figure 4 shows the performance of DRAPE on a particularly dynamic one-hour period of time in the 1998 World Cup trace, where the website was imitated exactly, with static pages.

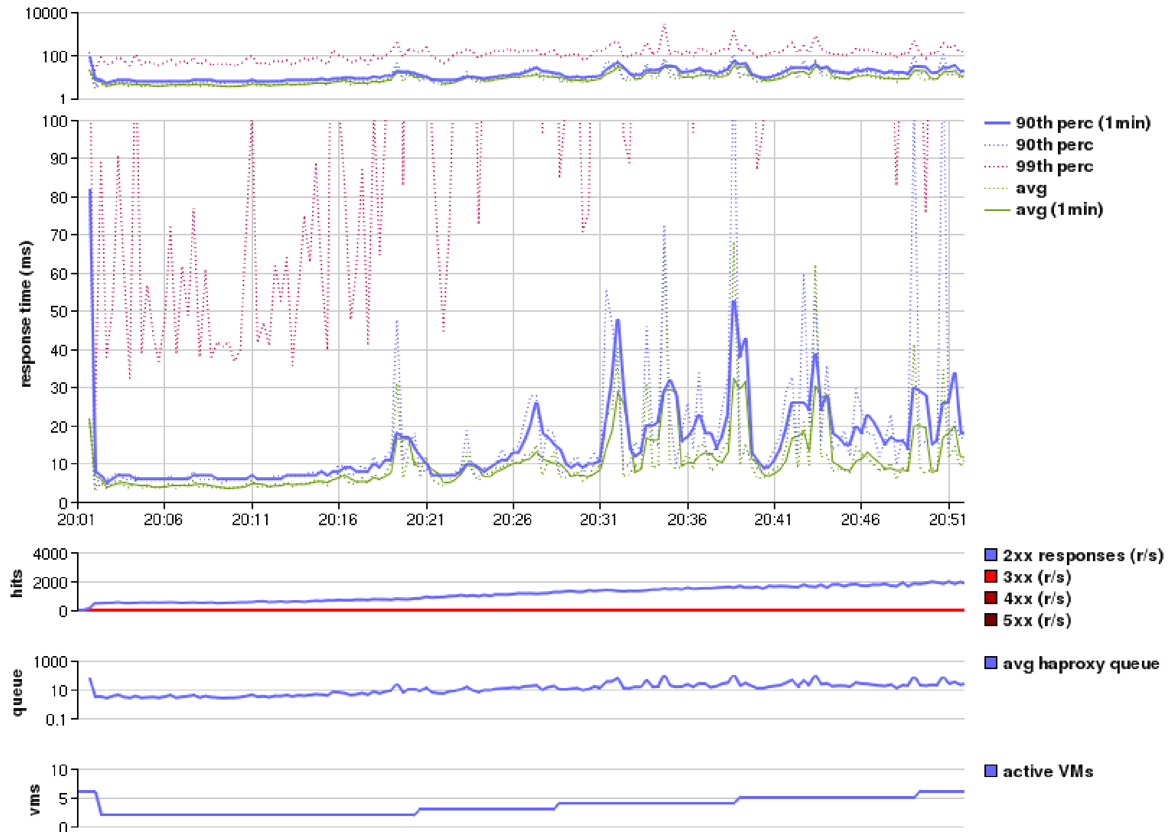

Figure 4: Performance of the DRAPE system on 1 hour of the 1998 World Cup trace in static mode.

DRAPE does a good job of keeping up with the quickly-increasing stream of requests. Although response times increase towards the end of the experiment, the SLA was never violated.

4.3 PHP 1998 World Cup

Figure 5 shows the performance of DRAPE on the same one-hour period as above, but where the website has been replaced by a PHP version. This makes all of the requests take much longer to satisfy, so this is essentially a test of how well our resource usage model can adapt to different situations.

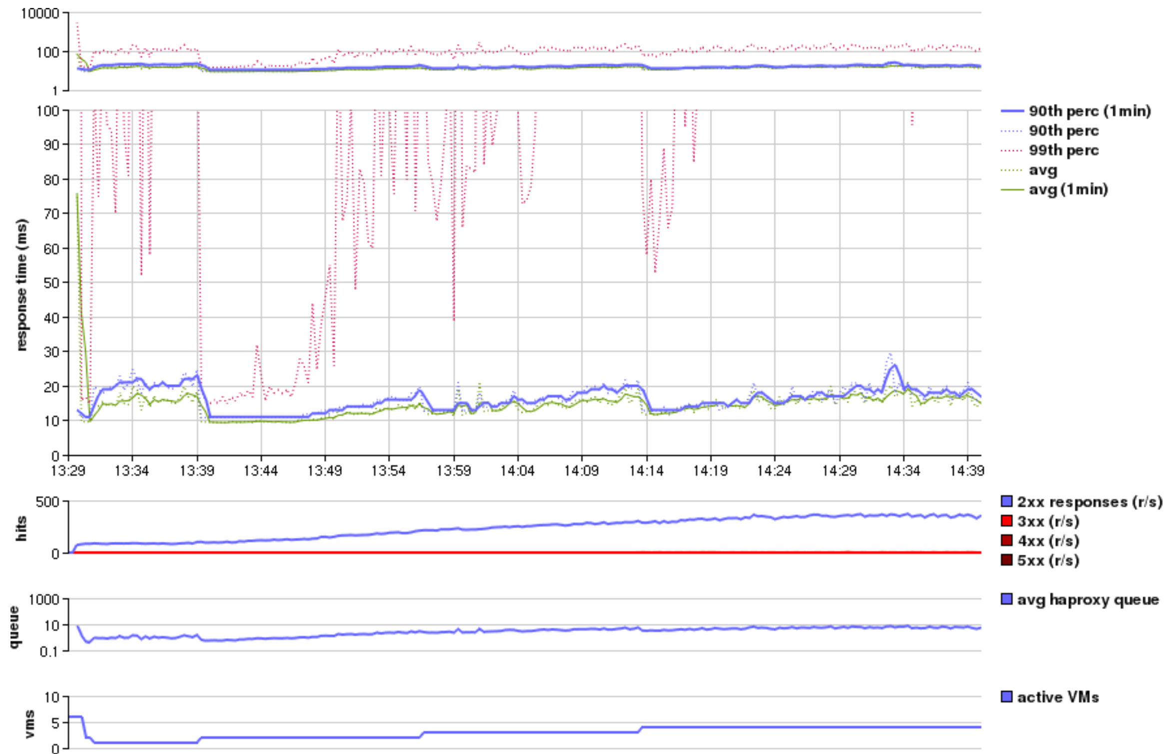

Figure 5: Performance of the DRAPE system on 1 hour of the 1998 World Cup trace in PHP mode.

As you can see, DRAPE once again keeps up with the quickly-increasing rate of requests, turning on servers only when necessary.

4.4 Static-to-PHP 1998 World Cup

Figure 6 shows the performance of DRAPE on constant number of requests, but where the requests are for static pages with probability p, where p moves linearly from 0 to 1 through the duration of the experiment. This test isolates the resource usage model, since the workload model is static.

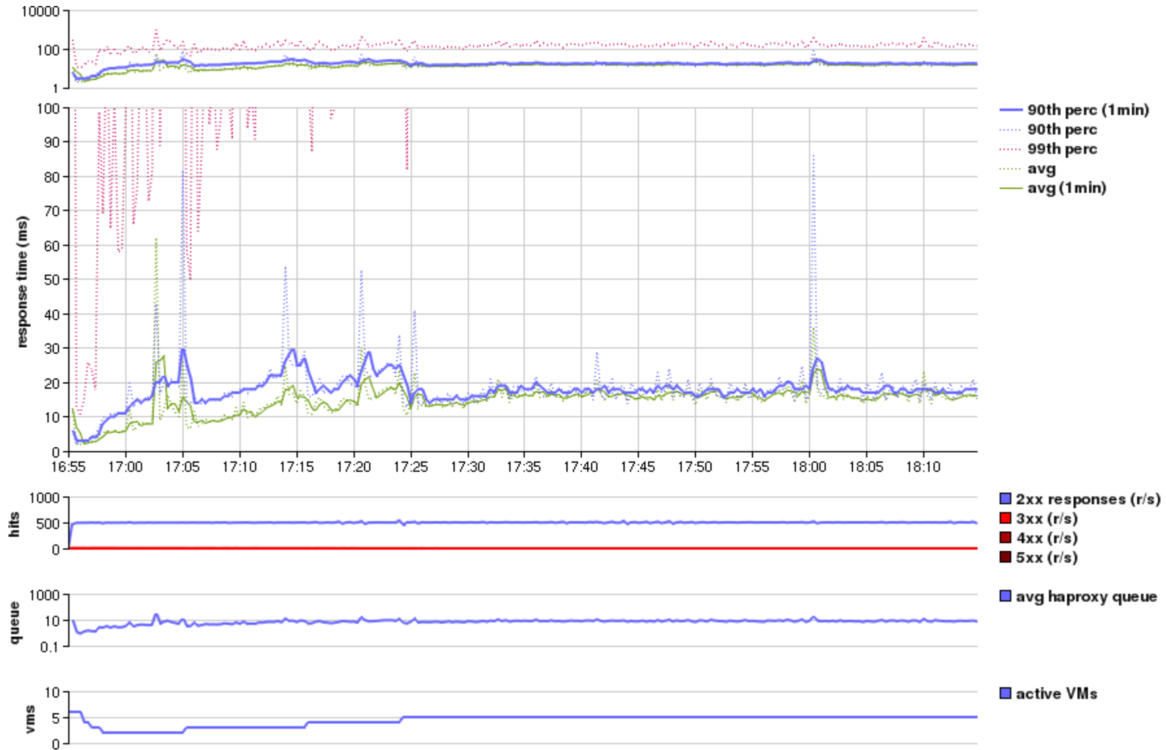

Figure 6: Performance of the DRAPE system on 1 hour of the 1998 World Cup trace, starting with 100% static page requests and ending with 100% PHP requests.

You can see that while the number of incoming requests stays constant throughout the experiment, the response times increase until DRAPE decides that another server is necessary, due to the dynamic resource usage model updating its threshold calculation at each step.

5 Conclusions

We have shown that for reasonably well-behaved and realistic workloads on a web-server application, it is possible to dynamically turn servers on and off in response to changes in resource demand in order to reduce power consumption without sacrificing performance.

Although our problem environment was fairly simple, we believe some of the same methods can be used to solve more general problems involving multiple applications of different types running on the same cluster of servers.

6 Future Work

There are several things that we'd like to explore that were either set aside because of time constraints or came to light because of this work:

- We originally planned to have multiple VMs per physical server in order to explore the interdependencies of resource allocation over multiple web applications, but in the end we limited ourselves to a single application. We hope that the successful results of this project can be applied to the more general problem of multiple applications.
- With multiple applications competing for the same servers, the action space will be much larger. We envision a reinforcement learning algorithm will be most useful in this environment. ALisp [4], due to Bhaskara Marthi and Stuart Russell, might prove to be the best method for constraining the policy to speed up learning and eliminate costly decisions during learning.
- Instead of setting each machine's workload threshold at 80% CPU usage, we think it would be valuable to inform our resource model with a variance estimate. For example, in the CPU utilization model, we could assume that the data points are linear with a normally-distributed error term, and use highest threshold for which the, say, 95% confidence interval of CPU usage is less than, say, 90% CPU utilization. Also, it makes sense to calculate the resource model for each machine individually rather than for all the machines as a group. We have already seen evidence that the lack of individual modeling hampers DRAPE's performance.

- For this project, we hard-coded the parameter values of our learning algorithms. It seems that the optimal values, however, are a function of the workload itself. So it makes sense to allow DRAPE to continuously recalculate optimal values for the parameters, i.e., the linear regression window size for workload prediction and resource utilization.

References

[1] M. Arlitt and T. Jin. 1998 World Cup Web Site Access Logs. http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[2] Rajarshi Das, Gerald Tesauro, and William E. Walsh. Model-based and model free approaches to autonomic resource allocation. Technical report, IBM T.J. Watson Research Center, 2005.

[3] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 595–604, New York, NY, USA, 2006. ACM Press.

[4] Bhaskara Marthi. ALisp. http://www.cs.berkeley.edu/ bhaskara/alisp/.

[5] Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer, Malgorzata Steinder, Asser Tantawi, and Alaa Youssef. Managing the response time for multi-tiered web applications. Technical report, IBM T.J. Watson Research Center, 2005.

[6] Willy Tarreau. HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. http://haproxy.1wt.eu/.

[7] Gerald Tesauro. Online resource allocation using decompositional reinforcement learning. Technical report, IBM T.J. Watson Research Center, 2005.

[8] Gerald Tesauro, Rajarshi Das, William E. Walsh, and Jeffrey O. Kephart. Utility-function-driven resource allocation in autonomic systems. In *Proceedings of the Second International Conference on Autonomic Computing*, 2005.

[9] Gerald Tesauro, Nicholas K. Jong, Rajarshi Das, and Mohamed N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. Technical report, IBM T.J. Watson Research Center, 2005.

[10] VMWare. VMWare Infrastructure SDK. http://www.vmware.com/support/developer/vc-sdk/.