Autonomous Return on Investment Analysis of Additional Processing Resources

Jonathan Wildstrom,[*] Peter Stone, Emmett Witchel
Department of Computer Sciences, The University of Texas at Austin
{jwildstr,pstone,witchel}@cs.utexas.edu

1. Introduction

With the recent explosion in computing power available in a single system, much attention has been turned towards the concepts of virtualization of these systems. As this trend continues, it is logical to assume that deploying an entire web-based service on an externally managed virtualized environment will soon be not only plausible, but common. In such an environment, the different components of the system would be logical machines, which would each appear to have private access to CPUs, memory, and other resources, and which would each be running an independent operating system. Because resources are not shared between virtual machines, administrators will need to make decisions about how much they should invest in resources and when more (or less) resource capacity is a good investment. An autonomous agent that can determine the Return on Investment (ROI) of the resource can make this management problem easier.

We consider the situation where compute time can be purchased for the database machine of a simple online bookstore, which we model using the standardized TPC-W[1] benchmark. A service level agreement (SLA) defines the value of the system, using throughput, response time, and expected response time as metrics (Figure 1). The autonomous agent must weigh the potential gain (or loss) in value defined by the SLA against the cost of purchasing (or relinquishing) compute time.

We implement such an autonomous agent on a simulated partitionable system (full details are available in our previous research [2]) and show that it is possible to outperform many static choices of compute power, over a range of test workloads and resource costs. The

Figure 1. SLA function

agent uses only raw, low-level system statistics, without the need for custom instrumentation of the middleware or operating system.
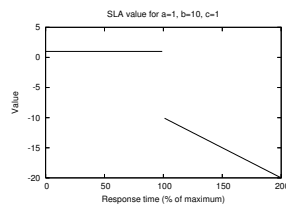
2. The tuning agent

The tuning agent handles autonomous changes to the compute power available to the back-end machine. The agent simulated is one that learns the value of adding or removing compute power. This agent could be a set of hand-coded rules, a learned model, or any number of other methods; Our agent is based on a learned regression model using M5′ trees [1].

The agent is first trained to predict the gain or loss in SLA value per second and then uses this prediction online to determine if it is larger or smaller than the associated cost of compute power. This training is accomplished by collecting low-level system statistics during runs of the benchmark on all possible levels of compute power. 5 quantities of compute power are considered, and 100 sets of runs are used to gather training data. Each set of runs consists of a random workload run for 300 seconds against each of the 5 possible configurations.

Each of the 500 training runs generates 6 input vectors, which consist of the system statistics averaged over a 30 second window and a value of the difference between the SLA value per second of the run's configuration and the SLA value of the configuration with one more slice of compute power (when the configuration is not maximal). These 2,400 vectors comprise the training data for one learner; a similar data set for reducing compute power is also created. Given these two sets of data vectors, the WEKA [3] package is used to build the two regression models used by the agent.

The learned model is evaluated by running it against new randomly generated workloads. However, unlike the training workloads, these workloads are not static throughout the measurement interval; instead, each workload consists of three randomly generated phases of 300 seconds each. Within each phase, the workload is generated in the same manner as the training workloads are generated. 5 random workloads are generated and used for testing the model.

During each run, the agent collects system statistics from the back-end machine. A persistent connection running a favored priority process ensures that the statistics are not blocked by the benchmark. A sliding 30-second window of

---

[*]currently employed by IBM Systems and Storage Group. Any opinions expressed in this paper may not necessarily be the opinions of IBM.

[1]TPC-W is a trademark of the Transaction Processing Performance Council.

these statistics is averaged and normalized, as in the generation of the training data, and used as the input vector to the two learned trees. Each tree predicts a single number, which is the estimated value or loss of more or less compute time. These values are compared to a known static cost and compute time is purchased or released if it is a good investment. In order to achieve some measure of hysteresis, the agent sleeps for 30 seconds if compute time was either purchased or released.

In order to determine the robustness of the agent, all runs are done with 3 possible costs for a CPU slice: 10, 15, and 25. We assume the units here are the same as those used for the SLA measurement.

3. Results and Future Work

To analyze our adaptive agent, the 5 possible static quantities of compute time purchased are first run against the randomly generate workloads. Each of these runs is performed 15 times. Subtracting the cost of the CPU slices used from the average SLA value for each static run gives us the the overall value of the workload and configuration.

The adaptive agent also makes 15 runs; however a separate set of 15 runs is necessary for each of the three costs. For each of these runs, an independent value is computed by taking the single run's SLA value and removing the average cost per second of CPU purchased. The values from the 15 runs for each cost are averaged to get the overall average value for the adaptive agent for the given CPU slice cost.

The results for the adaptive agent show that it is generally able to compete favorably. In only 1 of the 15 cases is the agent significantly worse than the best static configuration. More important, however, is that there is at least one situation where each static configuration is significantly worse than the adaptive agent. This implies that the agent could be a useful tool if the exact workload is not known.

The average value of the system is shown in Figure 2 and the average CPU purchased by the agent on one random workload is shown in Figure 3. It is clear that the amount of CPU used throughout the test changed, with the agent determining that the extra CPU was important in phase 3 even when the CPU was the most expensive. Results from another workload (not shown) show an unusual oscillation between 3 and 4 slices during the second phase of the test. This seems to imply that the agent believed the optimal CPU choice for that phase was actually between 3 and 4 slices, and this oscillation was an attempt to approximate the possibility of having 3.5 CPU slices. This allows the agent to take advantage of a configuration that is not available as a static configuration, looking for a "sweet spot" between configurations.

Figure 3 also shows that the agent is generally more willing to purchase CPU slices when they are cheaper, as is expected, and is more conservative when the price increases.
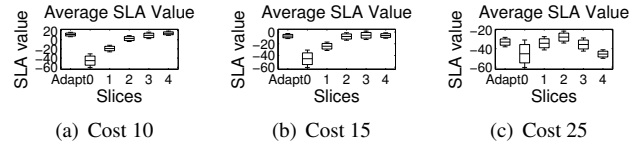

| (a) Cost 10 | (b) Cost 15 | (c) Cost 25 |

Figure 2. Results of one random workload. Boxes and whiskers show the 95% and 99% confidence intervals.
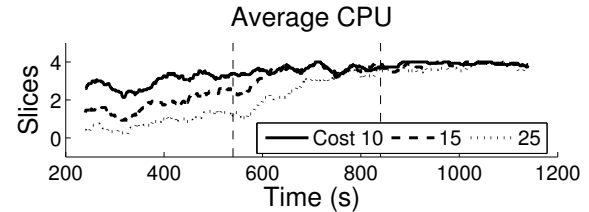

Figure 3. Average CPU slices purchased by the adaptive agent for various costs on the 5 random workloads.

However, we can also see that it is willing to purchase the maximum CPU if it believes it will help.

It is clear that there is a potential benefit to this autonomous compute time decision process. In most cases, the agent is competitive with the optimal static configuration, indicating that it is a viable alternative. Since the optimal static configuration is not always the same, this agent enables the system to be configured for an unknown workload, whereas the best static configuration may not be known. Additionally, the autonomy of the agent to toggle back and forth between configurations allows the agent to take advantage of partial compute time slices without needing to always pay the price of a full slice.

Our ongoing research involves further work with improving the learning agent, including learning the value of other resources (such as memory). Another research direction involves the agent taking into account an additional one-time cost of switching the resource state; this could be the cost to the owner of the temporary unavailability of the resource. We also want to experiment with non-static costs (e.g., the first slice of compute time is more expensive than the second) and with different SLAs or applications.

References

[1] Y. Wang and I. H. Witten. Induction of model trees for predicting continuous classes. In *Proceedings of the Poster Papers of the European Conference on Machine Learning*, pages 128–137, 1997.

[2] J. Wildstrom, P. Stone, E. Witchel, and M. Dahlin. Machine learning for on-line hardware reconfiguration. In *Proceedings of the 20th International Joint Conference On Artificial Intelligence*, pages 1113–1118, Hyderabad, India, January 2007.

[3] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools with Java implementations*. Morgan Kaufmann, San Francisco, 2000.