In *2007 Workshop on Adaptive Methods in Autonomic Computing Systems* (AMACS '07), Jacksonville, FL, June 2007.

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

Abstract

As the use of virtualization and partitioning grows, it becomes possible to deploy a multi-tier web-based application with a variable amount of computing power. This introduces the possibility of provisioning only for a minimum workload, with the intention of renting more resources as necessary, but it also creates the problem of quickly and accurately identifying when more resources are needed or unneeded resources are being paid for. This paper presents a machine learning based approach to handling this problem. An autonomous adaptive agent learns to predict the gain (or loss) that would result from more (or less) resources; this agent uses only low-level system statistics, rather than relying on custom instrumentation of the operating system or middleware. Our agent is fully implemented and evaluated on a publicly available multi-machine, multi-process distributed system (the online transaction processing benchmark TPC-W). We show that our adaptive agent is competitive with any static choice of computing resources over a variety of test workloads. We also show that the agent outperforms each static choice in at least one case, implying that it is well suited for a situation where the workload is unknown

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. This has ranged from work in virtual machines [9, 12] to partitioning hypervisors [2, 17] to on-demand resource availability [1, 4]. 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. As this happens, administrators will need to make decisions about how much they should invest in resources (compute time, memory, etc.), and when more (or less) resource capacity is a good investment. This paper presents one approach to learning how to make this decision autonomously.

We consider here the situation where more 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 and response time as metrics. 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.

This paper reports on the results of our autonomous agent. We show that it is possible to outperform many static choices of compute power, over a number of test workloads and sample costs of compute power. Additionally, over the workloads tested, the autonomous agent is significantly better than each static configuration in at least one test. These results are obtained using only raw, low-level system statistics, without the need for custom instrumentation of the middleware or operating system.

The remainder of the paper is organized as follows. The next section discusses the implementation, training, and evaluation of our autonomous agent. Section 3 contains the results of our experiments and some discussion of their implications. Section 4 gives an overview of related work, and Section 5 concludes.

2. The testbed and tuning agent

Through the use of virtualization, it is possible to have a single physical machine or grid of machines implement a set of logical machines, each of which appears to have private access to CPUs, memory, and other resources, enabling each virtual machine to run its own instance of an operating system. This isolation can be used to segregate parts of a larger system, with the benefits of minimization of resource contention and fault isolation. In addition, this provides the capability to increase resources available to part of the system, without allowing the other parts to "steal" this resource.

However, increasing resources has a cost associated with

[*]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.

it. This cost may represent the price of the additional power necessary to run and cool the machine as in a datacenter; it may represent a cost to a hosting service to supply more compute time; or it may represent the price of renting more memory or processors, as in IBM's Capacity on Demand model [4]. In exchange for this cost, the system gains some amount of value as measured by the administrator (e.g., in terms of transactions processed, SLA agreements, or improved availability). The challenge is to ensure that the increase in value exceeds the cost. The research reported in this paper simulates such a resource-on-demand system, specifically with regard to compute time resources. A full description of our testbed is described in our previous research [14, 15]. In this work, we consider value in terms of an SLA agreement where the system recieves credit for satisfying the expected response time of a page, but is penalized for exceeding this threshold. The SLA we simulate is displayed in Figure 1.

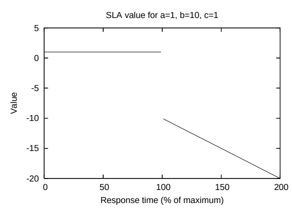


Figure 1. The value of a single transaction

While the cost of increased resources is known or easily estimated, the increase in value can be very hard to determine. Real-time observation of the system and modification of resource availability can help to make this choice. In particular, a tuning agent that can learn to estimate the value of more (or less) resource can help make an informed decision about whether the current quantity of resources is underutilized, overutilized, or sufficient. The agent described in this research performs such a task, as well as handling 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. The agent discussed in this work is based on a learned regression model that uses only low-level system statistics in its prediction. This learned version of the tuning agent is discussed in detail in Section 2.1, and our methodology for testing it is discussed in Section 2.2.

2.1. Training the model for a learning agent

In order for the agent to have the ability to make informed decisions about when to increase or decrease computing power, it must be able to predict the gain or loss in SLA value. The value we consider here, and for the remainder of this paper, is in value per second. Additionally, we would like this prediction to be done using only low-level kernel statistics, which are efficient to collect and independent of the version of OS and middleware.

In order to acquire training data, 100 random workloads are run against the SUT; each workload is run with each possible quantity of compute power available to the back-end machine (See Table 1). These 500 runs are used in building the training file for the learned model.

Generation of the random workload is accomplished through a 3-step process. First, a total number of EBs for the workload is chosen. This number, E_{tot}, is chosen uniformly at random from $[400, 800]$. Next, two values, E_1 and E_2 are chosen, with E_1 uniformly random over $[0, E_{tot}]$ and E_2 uniformly random over $[0, E_{tot} - E_1]$. E_3 is then defined as $E_{tot} - E_1 - E_2$. This defines the number of EBs running each workload. Finally, because this tends to be biased towards E_1, this triplet is randomly permuted into $\{M_1, M_2, M_3\}$. The workload then consists of M_1 browsing users, M_2 shopping users, and M_3 ordering users.

Slices	Fraction of maximum
0	$\frac{3}{4}$
1	$\frac{13}{16}$
2	$\frac{7}{8}$
3	$\frac{15}{16}$
4	1

Table 1. Correlation between slices and maximum compute time.

Each run has a measurement interval of 300 seconds. After each workload's five runs are done, the SLA value is computed for each individual run[2]. These SLA values are used in generating a set of data points for each workload. While it is possible that the individual run may be abnormally good or bad (there is substantial variability in individual runs), the large number of random workloads helps mitigate this effect.

We propose that low-level system statistics ought to be able to help determine if a system is overloaded, underloaded, or correctly utilized. Furthermore, we would like these statistics to help guide our agent in determining how much extra resource is needed or how much excess resource is present in the system. Previous research [14] has shown that these statistics can help an agent choose an optimal configuration from a set of possible static configurations; we here extend this to predicting the benefit of extra resources.

To gather the system statistics for the training data, a log is collected during each training run using the *vmstat* command. In order to ensure that the *vmstat* data represents the system during the measurement interval, the command is only run for 200 seconds during the middle of the measurement interval.

The statistics reported by *vmstat* are listed in Table 2. In order to make these statistics as configuration-independent as possible, the memory statistics are converted to percentages and the CPU percentages are corrected for the simulated hardware and renormalized. The resulting vector of statistics comprises the input representation for our trained model. During testing of the model, a similar vector is generated on-line; the agent's model uses this vector to predict

[2]The current implementation does not provide for online SLA values.

both the gain in SLA value for adding one slice $\left(\frac{1}{16}\right)$ of compute time and the loss in SLA value for removing one slice. If the gain is greater than the cost of one slice, and the system is not already at the maximum available CPU, more compute time is requested; a similar analysis is done for the possibility of releasing some compute time.

processes (number)	runnable	blocked
memory (KB)	VM used	idle
	inactive	active
swapping (KB/s)	swapped in	swapped out
I/O (blocks/s)	received	sent
System (per second)	interrupts	context switches
CPU (%)	user	system
	idle	waiting

Table 2. The statistics reported by *vmstat*.

Given the SLA value and log of system statistics for a single run, generation of a set of data points is accomplished as follows. First, the collected system statistics are divided into non-overlapping 30 second intervals, for a total of 6 training intervals from each 200 second training run. Each interval is averaged, providing the input representation for a training vector. The target value for each vector from a given run is the difference between the given run's SLA value and the SLA value of the run with one more slice of compute power (i.e., how much benefit there is from adding computing power). In this way, each run gives 6 data vectors, for a total of 2,400 vectors[3]. A similar method is used to generate a set of data vectors representing the cost (in SLA value) of removing computing power.

Given these two sets of data vectors, the WEKA [16] package is used to build a regression model to predict the benefit (cost) of adding (removing) a slice of compute time, given only the system statistics. The WEKA package implements many common machine learning algorithms; for this work, we use M5′ trees [13]. This learning method combines the interpretability of a decision tree with the regression learning of a multi-dimensional linear function approximator. By using M5′ trees, we can view and understand the predictor, without overly reducing the complexity of the learning algorithm.

In general, the trees and rules generated by WEKA are too complex to display in their entirety. The tree for adding compute time has nine possible regression rules, and the tree for removing compute time has twelve rules. The tree learned for increasing compute time and an example rule can be found in Figure 2.

[3]5 runs only allow 4 comparisons, as we cannot add compute power when we already have the maximum.

2.2. Evaluating the learned model

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 used to test the learned 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 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 value or loss of more or less compute time. These values are compared to a known static cost. First, if the predicted value of more compute time exceeds the cost, a new slice is purchased, if available. If time was not purchased, the loss predicted for less compute time is compared to the cost recouped by using less slices; if the savings exceed the loss, a slice is released. In order to achieve some measure of hysteresis, the agent sleeps for 30 seconds if compute time was either purchased or released.

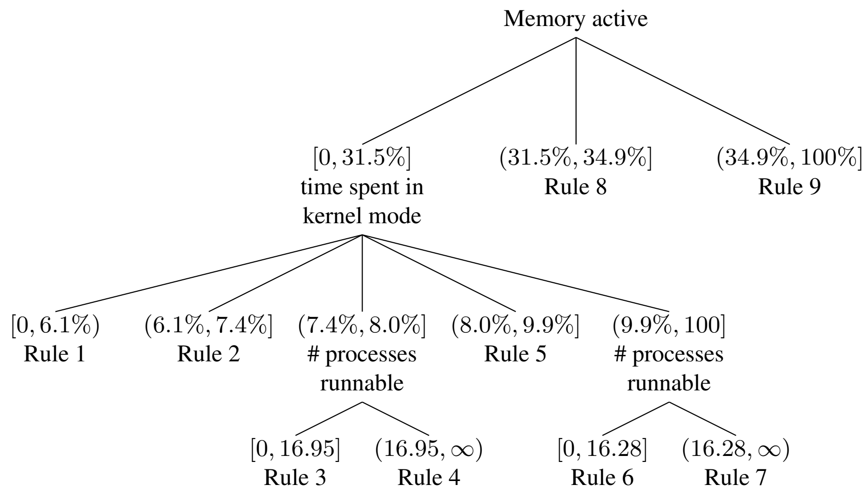
In order to determine the robustness of the agent, all runs are done with 3 possible costs for a CPU slice: 10, 15, and 25. We assume the units here are the same as those used for the SLA measurement. The actual unit is arbitrary, as long as both can be expressed in a common unit.

3. Results and Discussion

To analyze our adaptive agent, the 5 possible static quantities of compute time purchased are first run against the 5 randomly generated test workloads. Each of these runs is performed 15 times. The average SLA value for each static run is then computed. This gives us the raw SLA value for the workload and configuration. However, to get the overall value of the workload and configuration, it is necessary to subtract the cost of the CPU slices used in the static 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[4]. 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.

[4]The SLA value is in units per second and the cost per CPU slice is in units per second, but the number of CPU slices varies over the interval of the run.



Rule 1: SLA gain from 1 more slice =
0.555 * runnable processes +
0.5854 * blocked processes -
796.4686 * % virtual memory used -
82.7233 * % memory idle -
19.3325 * % memory inactive +
22.4356 * % memory active +
0.0182 * blocks per second received -
0.0051 * blocks per second sent +
0.0002 * context switches per second +
18.4187 * % CPU time in user space -
526.532 * % CPU time in kernel space +
66.1193 * % CPU time idle -
3.3529 * % CPU time waiting for I/O +
87.2663

Figure 2. The tree and one regression rule learned for increasing compute time. Each of the rules is a simple linear function approximator that calculates a weighted sum of the numeric values of the input vector.

The results of the adaptive agent and static configurations are shown graphically in Figure 3. The relationship between number of slices and fraction of maximum compute time is found in Table 1.

When we look at the results for the adaptive agent, we see that, although it does not always outperform all the static configurations, it is generally able to compete favorably. Although it is never the single statistically best configuration, there are two important points to notice. First, the adaptive agent is generally competitive with the best static configuration. 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 has the capability to outperform all static choices for some situation and could be a useful tool if the exact workload is not known.

From the average CPU choices shown in Figure 4, we can see that the amount of CPU used throughout the test changed, but examination of the specific choices over time shows an unexpected benefit of the agent. For workload 3, we see some unusual oscillation 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.

The graphs also show 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. However, we can also see that it is willing to purchase the maximum CPU if it believes it will help, as in workload 2. Finally, we can see that the agent often makes changes in the CPU purchased as the workload changes. In workload 1, the purchased CPU in phase 2 is noticeable less than in phase 1; in workload 2, the agent steadily purchases CPU through phase 2 and into phase 3.

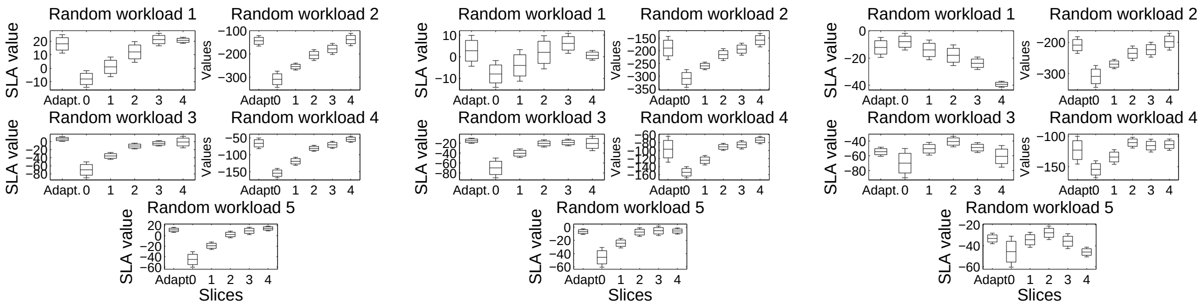
It is clear that there is a potential benefit to this autonomous compute time decision process. In most cases, the agent is competitive with the optimal static configuration, indicating that it is a viable alternative. Since the optimal static configuration is not always the same, this agent enables 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.

4. Related Work

The concept of purchasing extra CPU time has, until recently, been primarily restricted to use in datacenters, where the initial provisioning of a machine hinders rapid changes and, therefore, flexibility. This section reviews the work most related to the concepts reported in this paper.

Tesauro et al. [10] discuss a method for using reinforcement learning to determine how to best allocate servers in a data center in order to maximize the value of the entire data center. However, in order to use a reinforcement learning approach, the learner must have some knowledge of the value of a given transaction or set of transactions. As such, this requires some knowledge within the agent of the online SLA value. In contrast, our work is trained with only an aggregate value for a given run.

Kusic and Kandasamy [7] propose using a Limited



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

Figure 3. Results of the adaptive agent and static configurations with various costs. Boxes show the average and 95% confidence interval; the whiskers show the 99% confidence interval. The static configuration number corresponds to a number of slices as listed in Table 1.

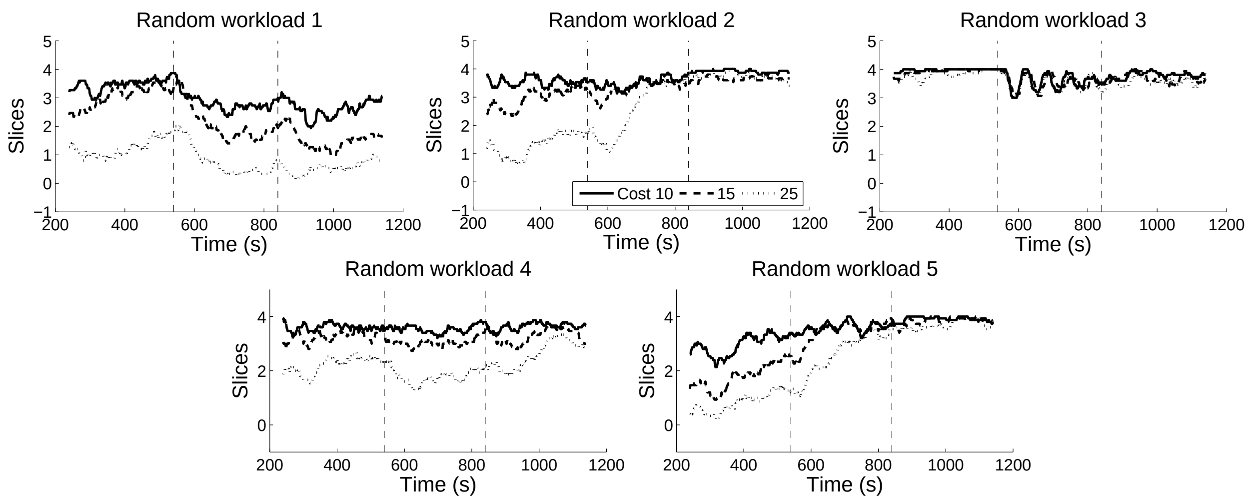


Figure 4. Average CPU slices purchased by the adaptive agent for various costs on the 5 random workloads.

Lookahead Controller for balancing the revenue from an SLA with the cost to operate the cluster; rather than requesting more or less CPUs, their work deals with the frequency of the CPU. In order to model the controller, a queuing model is needed that understands the system both in terms of tiers and connections and has facts about the response time of the system for different frequencies built in. Our work obviates the need for this information by using only the low-level system statistics.

Chen et al. [3] address automatically adding and removing database servers to a multi-tier system using a k-nearest-neighbors approach. Although they use similar system statistics to this work, they also use some metrics from the application. Their work is also primarily concerned with avoiding any violations of the SLA, rather than the trade-off between SLA gain or loss and the cost of operating the additional hardware. In contrast, our system is willing to consider increasing SLA violations if the resulting savings on compute time cost exceed the penalties associated with the SLA violations.

Urgaonkar et al. [11] use a queuing model to assist in provisioning a multi-tier Internet application. This work takes into account the concept of unused capacity, but assumes that it is already present and does not have a usage cost associated with it. Our work takes advantage of the growing field of on-demand resources.

Norris et al. [8] address the problem of handling excess workload by renting extra systems in a datacenter from other users. However, they do not address the issue of determining the value of extra systems, instead focusing on the issue of how the servers are priced. In contrast, we assume the price is fixed and known and the value of the additional capacity is learned.

5. Conclusion

As more companies begin offering resource-on-demand systems and virtualization solutions with variable resource allocations, there is likely to be growth in the number of businesses run on these types of systems. While the current method of capacity planning tends toward overprovisioning of systems, this new implementation model will enable administrators to target the expected workload for their permanent system capacity, while increasing their resource limits when needed. We demonstrate that an autonomous agent can quickly make the decision about when the application needs resources, freeing the administrator from the need to constantly monitor the system.

This paper presents one such autonomous agent. We show that our agent is able to make competitive choices balancing cost of compute time and gain in value, when measured against an SLA that defines both a per-transaction reward for satisfying a requirement and penalty for violating the requirement. The agent is also able to take advantages of intermediate configurations, which are not available as static choices, by oscillating between configurations.

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. Finally, we plan to extend our framework to report online SLA values and experiment with reinforcement learning and other online learning algorithms.

Acknowledgments

This research was supported in part by NSF awards IIS-0237699, CNS-0615104, and an IBM faculty award.

References

[1] Amazon Elastic Compute Cloud (Amazon EC2). Amazon.com, Inc., 2006. `http://aws.amazon.com/ec2/`.

[2] W. J. Armstrong, R. L. Arndt, D. C. Boutcher, R. G. Kovacs, D. Larson, K. A. Lucke, N. Nayar, and R. C. Swanberg. Advanced virtualization capabilities of POWER5 systems. *IBM Journal of Research and Development*, 49(4/5):523–532, July 2005.

[3] J. Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of backend databases in dynamic content web servers. In ICAC [6], pages 231–242.

[4] IBM eServer–Capacity on Demand. International Business Machines Corporation, 2006. `http://www.ibm.com/servers/eserver/about/cod/`.

[5] *Proceedings of the 2nd International Conference on Autonomic Computing*, Seattle, WA, June 2005.

[6] *Proceedings of the 3rd International Conference on Autonomic Computing*, Dublin, Ireland, June 2006.

[7] D. Kusic and N. Kandasamy. Risk-aware limited lookahead control for dynamic resource provisionling in enterprise computing systems. In ICAC [6], pages 74–83.

[8] J. Norris, K. Coleman, A. Fox, and G. Candea. OnCall: Defeating spikes with a free-market application cluster. In *Proceedings of the 1st International Conference on Autonomic Computing*, pages 198–205, New York, NY, May 2004.

[9] Parallels Workstation. Parallels, Inc., 2006. `http://www.parallels.com/en/products/workstation/`.

[10] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In ICAC [6], pages 65–73.

[11] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier Internet applications. In ICAC [5], pages 217–228.

[12] VMware ESX Server. VMware, Inc., 2006. `http://www.vmware.com/products/vi/esx/`.

[13] 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.

[14] 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.

[15] J. Wildstrom, P. Stone, E. Witchel, R. J. Mooney, and M. Dahlin. Towards self-configuring hardware for distributed computer systems. In ICAC [5], pages 241–249.

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

[17] XenEnterprise. XenSource, Inc., 2006. `http://www.xensource.com/products/xen_enterprise/index.html`.