Machine Learning for On-Line Hardware Reconfiguration

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

Abstract

As computer systems continue to increase in complexity, the need for AI-based solutions is becoming more urgent. For example, high-end servers that can be partitioned into logical subsystems and repartitioned on the fly are now becoming available. This development raises the possibility of reconfiguring distributed systems online to optimize for dynamically changing workloads. However, it also introduces the need to decide when and how to reconfigure. This paper presents one approach to solving this online reconfiguration problem. In particular, we learn to identify, from only low-level system statistics, which of a set of possible configurations will lead to better performance under the current unknown workload. This approach requires no instrumentation of the system's middleware or operating systems. We introduce an agent that is able to learn this model and use it to switch configurations online as the workload varies. Our agent is fully implemented and tested on a publicly available multi-machine, multi-process distributed system (the online transaction processing benchmark TPC-W). We demonstrate that our adaptive configuration is able to outperform any single fixed configuration in the set over a variety of workloads, including gradual changes and abrupt workload spikes.

1 Introduction

The recent introduction of partitionable servers has enabled the potential for adaptive hardware reconfiguration. Processors and memory can be added or removed from a system while incurring no downtime, even including single processors to be shared between separate logical systems. While this allows for more flexibility in the operation of these systems, it also raises the questions of *when* and *how* the system should be reconfigured. This paper establishes that automated adaptive hardware reconfiguration can significantly improve overall system performance when workloads vary.

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

Previous research [Wildstrom *et al.*, 2005] has shown that the *potential* exists for performance to be improved through autonomous reconfiguration of CPU and memory resources. Specifically, that work showed that no single configuration is optimal for all workloads and introduced an approach to *learning*, based on low-level system statistics, which configuration is most effective for the current workload, without directly observing the workload. Although this work indicated that online reconfiguration should, in theory, improve performance, to the best of our knowledge it has not yet been established that online hardware reconfiguration actually produces a significant improvement in overall performance in practice.

This paper demonstrates increased performance for a transaction processing system using a learned model that reconfigures the system hardware online. Specifically, we train a robust model of the expected performance of different hardware configurations, and then use this model to guide an online reconfiguration agent. We show that this agent is able to make a significant improvement in performance when tested with a variety of workloads, as compared to static configurations.

The remainder of this paper is organized as follows. The next section gives an overview of our experimental testbed. Section 3 details our methodology in handling unexpected workload changes, including the training of our agent (Section 3.1) and the experiments used to test the agent (Section 3.2). Section 4 contains the results of our experiments and some discussion of their implications. Section 5 gives an overview of related work, and Section 6 concludes.

2 Experimental Testbed and System Overview

Servers that support partitioning into multiple logical subsystems (partitions) are now commercially available [Quintero *et al.*, 2004]. Each partition has independent memory and processors available, enabling it to function as if it were an independent physical machine. In this way, partitions (and applications running on separate partitions) are prevented from interfering with each other through resource contention.

Furthermore, these servers are highly flexible, both allowing different quantities of memory and processing resources to be assigned to partitions, as well as supporting the addition and removal of resources while the operating system continues running. Hardware is also available that allows partitioning on the sub-processor level; e.g., a partition can use as little as $\frac{1}{10}$ of a physical processor on the hosting system [Quintero

et al., 2004]. Because reconfigurable hardware is not (yet) *easily* available, the research reported in this paper simulates reconfiguration of partitions on multiple desktop computers.

The remainder of this section gives a high-level overview of the testbed setup. A brief description of the TPC-W benchmark and a discussion of some modifications in our testbed can be found in Section 2.1. The software packages we use are given in Section 2.2. The hardware and simulation of sub-processor partitioning and reconfiguration are explained in Section 2.3. Finally, an overview of the implementation of the tuning agent is presented in Section 2.4. Further testbed and system details can be found in [Wildstrom *et al.*, 2006].

2.1 TPC-W

TPC-W is a standardized benchmark published by the Transaction Processing Performance Council. It dictates an instantiation of an online bookstore, with 14 dynamically generated web pages as the user interface. These pages are divided into six browsing and eight ordering pages. Relative performance of the System Under Test (SUT) is determined by using one or more external machines, called the Remote Browser Emulators (RBEs), running a set of Emulated Browsers (EBs). These EBs represent individual users of the site and may be browsing the store, searching, and/or placing orders.

Each user follows one of three defined workloads, or *mixes*: *shopping*, *browsing*, or *ordering*. Given a current page requested, the specification defines the respective probabilities of each possible next page for each mix. This results in different ratios of browsing and ordering page references for each mix. These relative percentages are summarized in Table 1.

	Mix		
	Browsing	Shopping	Ordering
Browsing pages	95%	80	50
Ordering pages	5%	20	50

Table 1: Expected access percentages of different pages for the TPC-W mixes, specified by [Garcia and Garcia, 2003].

A single run of the benchmark involves measuring the throughput during a fixed-length measurement interval. The system is allowed to warm up prior to the measurement interval. Throughput is measured in Web Interactions Per Second (WIPS), which is the overall average number of page requests returning in a second.

Many commercial systems that publish performance results involve large numbers or heterogeneous machines in complex setups. For simplicity, this work only considers the situation with a single database server (back-end) and a single web server (front-end); an illustration can be found in Figure 1. Although the throughput of this relatively small system (reported in Section 4) is significantly less than that of a commercially built system, this is due to the substantial difference in overall resources, and the throughput is not unexpected for an experimental system.

The TPC-W specification places some strict restrictions on various facets of the system. One example is that most pages contain a set of random promotional items displayed at the top; these items are expected to be randomly selected each time the page is generated. We relax this requirement: in our implementation, a set of items is cached for 30 seconds and reused for all pages desiring these promotional items during that time period. Other modified specifications can be found in [Wildstrom *et al.*, 2006].

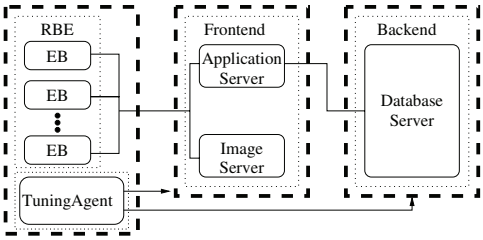

Figure 1: The 3 machines used in the physical setup.

The primary reason specifications are modified is that our intention is to overwhelm the system, whereas the TPC-W specifications are explicitly designed to prevent this from happening. Nonetheless, we use TPC-W because it is a convenient way of generating realistic workloads.

2.2 Software

Any TPC-W implementation must implement at least 3 modules as part of the SUT: a database, an application server, and an image server. The implementation used in this work uses PostGreSQL 8.0.4 for the database and Apache Jakarta Tomcat 5.5.12 as both the application and image servers. The Java code used by the application server (both for dynamic web page generation and database interaction) is derived from a freely available TPC-W implementation from the University of Wisconsin PHARM project [Cain *et al.*, 2001]. Slight modifications are made to the code: first, to use Tomcat and PostGreSQL; second, to allow limited caching of data; and third, to allow the number of connections to the database to be limited. The number of connections is controlled through a Java semaphore (with fairness enabled). The number of connections allowed is one reconfigurable resource, controlled through a new administrative servlet.

Furthermore, the RBE is modified in two ways. First, connection timeouts are not treated as fatal and are rather retried. Also, the EBs are not required to all run the same mix and can change which mix they are using during a benchmark run.

2.3 Hardware

The physical hardware used in this work consists of 2 identical Dell Precision 360n systems, each with a 2.8 GHz processor and 2GB RAM. A gigabit ethernet switch networks the systems through the built-in gigabit ethernet interfaces. As show in Figure 1, one machine is used for each of the front- and back-end systems, while a separate machine hosts the RBE and the tuning agent. In a true reconfigurable system, the tuning agent would likely reside either inside the partitionable system or on a management system.

Although, in practice, the front- and back-end systems are physically independent machines, they are meant to represent partitions within a single reconfigurable system with a total of 2.8GHz of processing power and 2 GB RAM. In order to simulate partitioning such a system into 2 partitions, CPU and memory are artificially constrained on the two machines. The

CPU is constrained by a highly favored, periodic busy-loop process; memory is constrained using the Linux *mlock()* subroutine. The constraints on the two systems are set in such a way as to only allow, overall, one full 2.8 GHz processor and 2GB memory to be used by the combined front- and back-ends. For example, if the back-end is allowed 2.0 GHz, the front-end only has 0.8 GHz available.

By using both of these constraining processes simultaneously, simulation of any desired hardware configuration is possible. Additionally, the processes are designed to change the system constraints on the fly, so we can simulate reconfiguring the system. In order to ensure that we never exceed the total combined system resources, we must constrain the "partition" that will be losing resources before granting those resources to the other "partition."

2.4 The tuning agent

The tuning agent handles real-time reconfiguration. There are three phases to the reconfiguration. First, the number of allowed database connections is modified. Once this completes, any CPU resources being moved are constrained on the machine losing CPU and added to the other; then memory is removed from the donor system and added to the receiving system. Each step completes before the next is begun.

There are two types of tuning agents we simulate. The first is an omniscient agent, which is told both the configuration to switch to and when the reconfiguration should occur based on perfect knowledge. The omniscient agent is used to obtain upper bound results. This uses a known set of configurations that maximizes the system performance, thus allowing us to have a known optimal performance against which to compare.

The second agent is one that makes its own decisions as to when to alter the configuration. This decision can be based on a set of hand-coded rules, a learned model, or any number of other methods. The agent we discuss in this paper is based on a learned model that uses low-level system statistics to predict the better option of two configurations. This learned version of the tuning agent is discussed in detail in Section 3.1

3 Handling Workload Changes

While provisioning resources in a system for a predictable workload is a fairly common and well-understood issue [Oslake *et al.*, 1999], static configurations can perform very poorly under a completely different workload. For example, there is a common phenomenon known as the "Slashdot effect," where a little-known site is inundated by traffic after being featured on a news site. We consider the situation where this site is an online store, which is configured for a normal workload of ordering and shopping users. The large number of unexpected browsing users appearing can overwhelm such a site that is not prepared for this unexpected change in workload. We call this situation a *workload spike*.

In addition to such drastic changes, the situation where the workload gradually changes is also possible and must be handled appropriately.

Unfortunately, in practice, the workload is often not an easily observable quantity to the system. While it is possible to instrument the system in various ways to help observe part or all of the workload, this would require customizing the middleware for each online store.

However, previous work [Wildstrom *et al.*, 2005] has shown that low-level operating system statistics can be used instead to help suggest what configuration should be used in a workload-oblivious way. This approach has the advantage of not requiring any customized instrumentation. By instead using out-of-the-box versions of software, this method allows for easy replacement of any part of the system without needing significant reimplementation. Low-level statistics also do not refer to workload features, and so might be easier to generalize across workloads. For these reasons, we use the low-level statistics in preference to customized instrumentation of either the operating system or the middleware.

3.1 Training the model for a learning agent

In order to automatically handle workload changes, we train a model to predict a target configuration given system statistics about overall resource use. To collect these statistics, each machine runs the *vmstat* command and logs system statistics. Because it is assumed that a true automatic reconfigurable system would supply these statistics through a more efficient kernel-level interface, this command is allowed to take precedence over the CPU constraining process.

vmstat reports 16 statistics: 2 concerning process quantity, 4 concerning memory use, 2 swapping, 2 I/O, 4 CPU use, system interrupts, and context switches. In order to make this more configuration-independent, the memory statistics are converted to percentages. The resulting vector of 32 statistics (16 from each partition), as well as the current configuration of the system, comprise the input representation for our trained model. The model then predicts which configuration will result in higher throughput, and the agent framework reconfigures the system accordingly.

For our experiments, we consider two possible system configurations using 3 resources: CPU, memory, and number of connections (Table 2). A wide range of possible CPU, memory, and connection limits were investigated, and the selected configurations maximize the throughput for the extreme situations. In particular, connection limits, although not a hardware resource, were necessary to keep the database from becoming overwhelmed by large numbers of queries during heavy browsing periods.

	Database		Webserver		
Config.	CPU	Mem.	CPU	Mem.	Conn.
browsing	$\frac{11}{16}$	0.75 GB	$\frac{5}{16}$	1.25 GB	400
ordering	$\frac{13}{16}$	1.25 GB	$\frac{3}{16}$	0.75 GB	1000

Table 2: Configurations (resource distributions and database connections allowed) used in the experiments.

In order to acquire training data, 100 pairs of varied TPC-W runs are done. Each pair of runs consists of one run of a set workload on each configuration; the workload consists of 500 shopping users and a random number (between 500 and 1000) of either browsing or ordering users (50 pairs each). Each run has 240 seconds of ramp-up time, followed by a

400 second measurement interval, during which 200 seconds of *vmstat* data are also collected.

From the combination of WIPS and system statistics gathered on each run, a set of training data points are created for each pair in the following way. First, the system statistics from each run are divided into non-overlapping 30 second intervals. The average system statistics over each interval are the input representation for one data point. Next, the configuration with the higher throughput (in WIPS) is determined to be the preferred configuration for all data points generated from this pair. In this way, each of the 100 pairs of runs generates 12 data points.

Given training data, the WEKA [Witten and Frank, 2000] package implements many machine learning algorithms that can build models which can then be used for prediction. In order to obtain human-understandable output, the JRip [Cohen, 1995] rule learner is applied to the training data. For the generated data, JRip learned the rules shown in Figure 2.

As can be seen in Figure 2, JRip determines a complex rule set that can be used to identify the optimal configuration for unobserved workload. 8 of the 32 supplied systems statistics are determined to be of importance; these 8 are evenly split over the front- and back-end machines. We can see that all of the front-end statistics used are either related to execution time or memory use, while the back-end statistics sample almost all statistic areas. We also note that the rules specify a means of identifying one configuration over the other; the browsing-intensive configuration is taken as the "default."

If we look at the rules in some more depth, we can identify certain patterns that indicate that the rules are logical. For example, rules 1 and 3 both set a threshold on the amount of time spent by the database waiting for I/O, while rule 5 indicates that a large percentage of the memory on the database is in use. All three of these indicators point to a situation where more memory could be useful (as in the database-intensive configuration). When memory is constrained, the database files will be paged out more frequently, so more time will be spent waiting on those pages to be read back in. Other trends are discussed in [Wildstrom *et al.*, 2006].

To verify our learned model, we first evaluate the performance of JRip's rules using stratified 10-fold cross validation. In order to prevent contamination of the results by having samples from a single run appearing in both the training and test sets, this was done by hand by partitioning the 100 training runs into 10 sets of 10 runs (each set having 120 data points). The 10 trials each used a distinct set as the test set, training on the remaining 9 sets. In this test, JRip correctly predicts the better target configuration 98.25% of the time.

One important facet of the rules learned is that they are domain-specific; although these rules make sense for our distributed system, different rules would be necessary for a system where, for example, both processes are CPU-heavy but perform no I/O (such as a distributed mathematical system). While we do not expect rules learned for one system to apply to a completely different system, training a new set of rules using the same methodology should have similar benefits. By learning a model, we remove the need to explore the set of possibly relevant features and their thresholds manually.

3.2 Evaluating the learned model

We present two types of workload changes: the gradual change and the workload spike. We concentrate our evaluation on the workload spike, as sudden changes in workload require faster adaptation than a gradual change.

For simplicity, we describe only the simulation of a browsing spike below; ordering spikes are simulated in a similar fashion. Three random numbers are generated $\{X_1, X_2, X_3\} \in [500, 1000]$. The base workload consists of 500 shopping users with X_1 ordering users. This workload reaches the steady state (240 seconds of ramp-up time), and then there are 200 seconds of measurement time. At that point, the workload abruptly changes to a browsing intensive workload, with X_2 browsing users replacing the ordering users. The spike continues for 400 seconds. After this time, the workload abruptly reverts to an ordering intensive workload, with X_3 ordering users. After 200 more seconds, the measurement interval ends.

We also simulate a gradual change. For this, we begin with a workload that consists of 500 shopping users and 800 browsing users. After 240 seconds of ramp-up time and 200 seconds of measurement time, 200 browsing users are converted to the ordering mix; this repeats every 100 seconds until all the users are running the ordering mix, at which point the measurement interval continues for 300 more seconds.

Each workload is tested under 4 hardware configurations. As baselines, both static configurations execute the workload. Additionally, because we know when the spike takes place and when it ends, we can test the optimal set of configurations with the omniscient agent. For the gradual change, we can see where each configuration is optimal and force the omniscient agent to reconfigure the system at that point. Finally, the learning agent is allowed to completely control the configuration based on its observations.

The agent continuously samples the partitions' system statistics and predicts the optimal configuration using a sliding window of the most recent 30 seconds of system statistics. If a configuration chance is determined to be beneficial, it is made immediately. After each configuration change, the agent sleeps for 30 seconds to allow the system statistics to reflect the new configuration.

4 Results and Discussion

Evaluation of the learning agent is performed over 10 random spike workloads. Of these workloads, half are browsing spikes and half are ordering spikes. Each workload is run 15 times on each of the static configurations, as well as with the learning agent making configuration decisions, and finally with the omniscient agent forcing the optimal hardware configuration. The average WIPS for each workload are compared using a student's t-test; all significances are with 95% confidence, except where noted.

Overall, the learning agent does well, significantly outperforming at least one static configuration in all 10 trials, and outperforming both static configurations in 7 of them. There are only 2 situations where the adaptive agent does not have the highest raw throughput, and in both case, the adaptive agent is within 0.5 WIPS of the better static configuration.

The ordering-biased configuration should be used if *any* column in the following table is satisfied:

Database	Execution time waiting for I/O	$> 6.96\%$		$> 5.45\%$		
	Memory active					$> 82.76\%$
	Context switches per second	> 509.38				
	Blocks received per second			< 2761.88		
App. Server	Execution time in user space		$< 44.90\%$			
	Execution time in kernel space				$< 32.53\%$	
	Memory active	$> 30.15\%$				
	Memory idle			$< 49.19\%$	$< 40.94\%$	

Otherwise, the browsing-biased configuration should be used.

Figure 2: JRip rules learned by WEKA. Numbers are averages over a 30-second interval.

	Configuration			
Spike Type	adapt.	browsing	ordering	omnisc.
ordering	77.0	69.2	69.1	77.0
browsing	70.7	64.6	69.5	72.1

Table 3: Average results (in WIPS) of different configurations running a spike workload.

The agent never loses significantly to either static configuration. Results can be found in Table 3.

In addition to performing well as compared to static configurations, the learning agent even approaches the accuracy of the omniscient agent. In 4 of the 5 ordering spike tests, the adaptive agent is no worse than 0.5 WIPS below the omniscient agent, actually showing a higher throughput in 2 tests[1]. One typical example of the throughputs of each of the four configurations on an ordering spike can be seen in Figure 3.

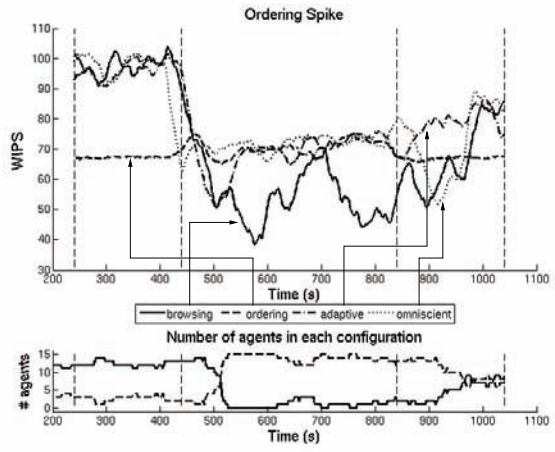

Figure 3: WIPS throughput for each of the configurations during an ordering spike, averaged over 15 runs. The graph at the bottom shows how many learning agents had chosen each configuration at a given time; these choices are all averaged in the "adaptive" graph.

In handling browsing spikes, the agent consistently exceeds the throughput of one static configuration and always performs at least as well as the other static configuration. In 2 of the 5 tests, the agent actually wins significantly over both configurations. However, the agent has more room for improvement than in the ordering spike tests; on average, the learning agent is 1.3 WIPS below the omniscient agent.

In many of the trials, we see some sudden, anomalous drops in throughput (at approximately 900 seconds in the browsing and omniscient configurations in Figure 3). These drops can sometimes confuse the agent. We believe this confusion is due to abnormal database activity during the anomaly. The cause of the anomalies has been identified[2]; future work will eliminate these anomalies.

In addition to spikes of activity, we test gradual changes in workload to verify that the agent is capable of handling gradual changes as detailed in Section 3.2. Over 20 runs, the average throughputs of the browsing- and ordering-intensive configuration are 70.7 and 69.3 WIPS, respectively. The learning agent handles this gradual change gracefully, winning overall with an average throughput of 76.6 WIPS. There is also room for improvement, as the omniscient agent, which switches configurations when there are 400 users running each of the browsing and ordering workloads, significantly outperforms the learning agent with an average throughput of 79.1 WIPS.

This method for automatic online reconfiguration of hardware has definite benefits over the use of a static hardware configuration. Over a wide variety of tested workloads, it is apparent that the adaptive agent is better than either static configuration considered. While the agent has room for improvement to approach the omniscient agent, omniscience is not a realistic goal. Additionally, based on the rule set learned by the agent, it is apparent that the problem of deciding when to alter the configuration does not have a trivial solution.

5 Related Work

Adaptive performance tuning through hardware reconfiguration has only recently become possible, so few papers address it directly. Much of the work done in this field thus far deals with maintaining a service level agreement (SLA). While this work is similar (relevant examples are cited below), this is a fundamentally different problem than that which we are investigating, where there is no formal SLA against which to determine compliance. This section reviews the most related work to that reported in this paper.

IBM's Partition Load Manager (PLM) [Quintero *et al.*, 2004] handles automatic reallocation of resources. While

[1] Presumably due to measurement noise.

[2] The database was not reanalyzed after population; as a result, all administrative tasks took a very long time performing sorts, which put a massive strain on the database

extensively configurable in terms of thresholds and resource limits, it must be hand-configured and follows strictly the rules set out by the thresholds. In contrast, our approach learns the appropriate thresholds to invoke a reallocation.

Menascé et al. [2001] discuss self-tuning of run-time parameters (threads and connections allowed) using a model based approach. The authors suggest that a similar method should be extensible to hardware tuning. This requires constructing a detailed mathematical model of the system with the current workload as an input; our work treats the system as a black box and workload as an unobservable quantity.

Karlsson and Covell [2005] propose a system for estimating a performance model while considering the system as a black box using a Recursive Least-Squares Estimation approach, with the intention that this model can be used as part of a self-tuning regulator. This approach is intended to help meet an SLA goal (using latency as the metric) but only to get as close as possible to the SLA requirement without exceeding it, rather than maximizing performance.

Urgaonkar et al. [2005] use a queuing model to assist in provisioning a multi-tier Internet application. This approach is intended to handle multiple distinct servers at each tier and assumes that there are unused machines available for later provisioning. Our approach is intended to have just one server at each tier with a fixed total amount of processing power.

Norris et al. [2004] address competition for resources in a datacenter by allowing individual tasks to rent resources from other applications with excess resources. This frames the performance tuning problem as more of a competitive task; we approach the problem as a co-operative task.

Cohen et al. [2005] use a clustering approach to categorizing performance and related problems. Using similar low-level statistics to us, they identify related problems, given a signature summarizing the current system status. This work is primarily geared toward simplifying problem diagnosis and analysis, whereas our work works toward automatically correcting the problem.

6 Conclusion

The rapid development of reconfigurable servers indicates that they will become more commonly used. These servers run large, distributed applications. To get the most out of the server, each application should receive only the hardware resources necessary to run its current workload efficiently. We demonstrate that agents can tailor hardware for workloads for a given application.

This work's main contribution is the introduction of a method for automatic online reconfiguration of a system's hardware. This method shows significant improvement over a static allocation of resources. Although an agent is only trained for one specific domain, the method is general and is applicable to a large number of possible combinations of operating systems, middleware, and workloads.

Our ongoing research agenda includes further work with the learning agent to approach the optimal results, as well as investigation on different workloads. We also want to predict the benefit of a reconfiguration, both on our simulated reconfigurable machines and on true reconfigurable hardware.

Acknowledgments

The authors thank Ray Mooney and the rest of the Cognitive Systems reading group for valuable input throughout the research and helpful critiques of the paper. This research was supported in part by NSF CAREER award IIS-0237699, a DARPA ACIP grant, and an IBM faculty award.

References

[Cain *et al.*, 2001] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural evaluation of Java TPC-W. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, January 2001.

[Cohen *et al.*, 2005] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 105–118, October 2005.

[Cohen, 1995] W. W. Cohen. Fast effective rule induction. In *Proceedings of the 12th International Conference on Machine Learning (ICML-95)*, pages 115–123, 1995.

[Garcia and Garcia, 2003] D. F. Garcia and J. Garcia. TPC-W e-commerce benchmark evaluation. *Computer*, 36(2):42–48, February 2003.

[Karlsson and Covell, 2005] M. Karlsson and M. Covell. Dynamic black-box performance model estmation for self-tuning regulators. In *Proceedings of the 2nd International Conference on Autonomic Computing*, pages 172–182, June 2005.

[Menascé *et al.*, 2001] D. A. Menascé, D. Barbará, and R. Dodge. Preserving QoS of e-commerce sites through self-tuning: A performance model approach. In *Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 224–234, October 2001.

[Norris *et al.*, 2004] 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, May 2004.

[Oslake *et al.*, 1999] M. Oslake, H. Al-Hilali, and D. Guimbellot. Capacity model for Internet transactions. Technical Report MSR-TR-99-18, Microsoft Corporation, April 1999.

[Quintero *et al.*, 2004] D. Quintero, Z. Borgosz, W. Koh, J. Lee, and L. Niesz. Introduction to pSeries partitioning. International Business Machines Corporation, November 2004.

[Urgaonkar *et al.*, 2005] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier Internet applications. In *Proceedings of the 2nd International Conference on Autonomic Computing*, pages 217–228, June 2005.

[Wildstrom *et al.*, 2005] J. Wildstrom, P. Stone, E. Witchel, R. J. Mooney, and M. Dahlin. Towards self-configuring hardware for distributed computer systems. In *Proceedings of the 2nd International Conference on Autonomic Computing*, pages 241–249, June 2005.

[Wildstrom *et al.*, 2006] J. Wildstrom, P. Stone, E. Witchel, and M. Dahlin. Adapting to workload changes through on-the-fly reconfiguration. Technical Report UT-AI-TR-06-330, The University of Texas at Austin, Department of Computer Science, AI Laboratory, 2006.

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