Adapting to Workload Changes Through On-The-Fly Reconfiguration

Jonathan Wildstrom*, Peter Stone, Emmett Witchel, Mike Dahlin

Department of Computer Sciences, The University of Texas at Austin
1 University Station C0500, Austin, Texas 78712-1188
{jwildstr,pstone,witchel,dahlin}@cs.utexas.edu
http://www.cs.utexas.edu/~{jwildstr,pstone,witchel,dahlin}

Technical Report UT-AI-TR-06-330

May 31, 2006

Abstract

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. 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

Recent advances in hardware development have made adaptive hardware reconfiguration possible. For example, processors and/or memory may be dynamically added to or removed from a running system [23]. This capability adds more flexibility to systems operation but raises the challenge of determining *when* and *how* to reconfigure. This paper establishes that automated adaptive hardware reconfiguration can significantly improve overall system performance when workloads vary.

In previous research [34], we established that there is a *potential* to improve system performance by reconfiguring CPU and memory resources in a benchmark transaction processing system. In particular, we demonstrated that no single configuration is best for all workloads and introduced an approach to *learning* which configuration is most effective for the current workload based on low-level system statistics. Although this work established that on-line reconfiguration should, in principle, 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 presents an approach to using such a learned model to guide on-line reconfiguration, showing that it can indeed be effective in practice. We extend our training methodology presented previously to train a new, more

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

robust learned model, and then use this model to guide an online agent to perform hardware reconfigurations. 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.

Additionally, a framework is described that allows for general testing of adaptive agents. Our system allows for on-the-fly reconfiguration, driven by a generic agent. Two instantiations of this agent are implemented: the learning agent described above and an omniscient agent used to modify the configuration at the optimal time so as to provide the performance limit for any adaptive agent implementation.

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

Large servers are now available that can be partitioned into one or more logical subsystems [23, 14, 28]. Each of these logical systems has some amount of memory and processors available to it, enabling it to operate as if it were an independent physical machine. By allowing each logical subsystem to run its own instance of the operating system, they are prevented from interfering with each other through resource contention.

Furthermore, these servers can be flexibly configured to allocate different amounts of memory and processing resources to the logical subsystems. Hardware is also available that allows a single physical system to partition itself into one or more logical systems at a sub-processor level; that is, where a logical system may only be permitted to use as little as $\frac{1}{10}$ of a processor on the physical hosting system [23].

However, this flexibility raises the challenge of determining how and when to reconfigure the hardware so as to maximize a system's performance. We have previously shown [34] that an allocation of resources to subsystems that maximizes performance for one set of workloads may be suboptimal for other workloads. In particular, real-time reconfiguration may be needed to maximize performance when the workload changes.

Because reconfigurable hardware is not (yet) easily available, the research reported in this paper simulates reconfiguration of logical subsystems on multiple desktop computers. The remainder of this section details the testbed setup. An overview of the TPC-W benchmark and a discussion of some modified specifications 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.

2.1 TPC-W

The TPC-W Benchmark [30] is a standardized benchmark put out by the Transaction Processing Performance Council.[1] It is designed to determine the relative performance of a System Under Test (SUT) when used to run an online bookstore. The benchmark operates by having an external machine or set of machines, the Remote Browser Emulators (RBEs), run a set of Emulated Browsers (EBs). These browsers represent individual customers of an online bookstore. The customers may browse through the store, view products, perform searches, and sometimes place orders.

The relative probabilities of the customers' actions are defined by the TPC-W specification. There are three workloads, called *mixes*, defined:

1. The *shopping* mix, which represents normal operation of the system, with 80% of the accesses being users browsing the available catalog, and 20% of the accesses are orders being placed;
2. The *browsing* mix, representing a slow commerce period, in which 95% of the users are browsing, and only 5% are ordering; and

[1] TPC-W has recently been replaced by TPC-App, previously known as TPC-W version 2.

3. The *ordering* mix, representing a rush on the latest hot book, in which browsing and ordering users are evenly split.

The differences between these mixes is summarized in Table 1. There are a total of 14 web pages that can be retrieved. These pages are divided into six browsing pages (Home, New products, Best sellers, Product detail, Search request, Search results) and eight ordering pages (Shopping cart, Customer registration, Buy request, Buy confirm, Order inquiry, Order display, Admin request, Admin confirm). The probability of a customer moving from a given page to any other page is well defined by the specification, and each page has its own expected response time.

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

Table 1: Expected percentages of different pages for the TPC-W mixes. The numbers in this table are specified by Garcia [10].

Pages are generated dynamically in response to user queries, and some pages require significantly more processing than others. For example, because the admin pages update the prices and stock in a highly-used database, they consume more resources than simply pulling up the home page. A single run of the benchmark consists of a ramp-up period, followed by a measurement interval. Results are usually summaries with a single measure of throughput—Web Interactions per Second (WIPS). WIPS are the average number of page requests that return in a second (equivalent to the total number of pages retrieved divided by the total time in seconds) during the measurement interval.

Commercially built and tested systems that publish performance results often have one main database server and many independent web servers. Additionally, they often have distinct web cache servers, image servers, and load balancers. When the benchmark was retired, the WIPS record holder [29] reported 21,139.7 WIPS using 27 2-processor web servers, 21 2-processor image servers (one of which is also a load balancer), 13 web caches (11 2-processor, 2 1-processor), and 1 8-processor database server. For simplicity, this work considers the situation where there is one database server (back-end) and one web server (front-end), as illustrated in Figure 1. Our system produces WIPS numbers that, though 3 orders of magnitude less than such a commercial system (due largely to the corresponding difference in processing power and memory), are not out of the ordinary for experimental systems.

The TPC-W specification places some strict restrictions on various facets of the system. For example, many 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 some of these requirements. 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.

Some other specifications that are modified are:

- *Relationship between the number of EBs and the reported WIPS*: The specification states that the number of EBs should be no more than 14 times the WIPS value. We run substantially more EBs than this quantity with the intention of overloading the system.

- *Initial population of the database*: The number of initial customers and related data is supposed to be 2880 times the number of EBs being used. We instead use a smaller number of customers, both to introduce contention in the database and because the number of EBs is not fixed.

- *Web Interaction Response Time (WIRT) constraints are ignored*: Each page has a maximum response time specified; 90% of requests of the page should satisfy the constraint. Because we are pushing the system to extremes, we do not want to ignore results that violate this threshold. Ongoing work uses this as a secondary metric.

The primary reason these 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] Since our results are not intended for submission to the Transaction Processing Performance Council, we are not concerned that our performance numbers cannot be directly compared against commercially built systems.

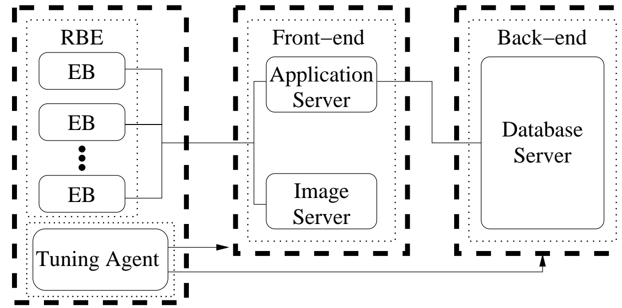


Figure 1: The 3 machines used in the physical setup. The thick, dashed rectangles represent physical machines. The dotted rectangles are processes, and the innermost rounded rectangles are logical units. Network connections are shown as lines and come together at the point where they are managed; i.e., the front-end handles routing of connections to the application and image servers, while the physical machine coalesces the individual EBs' network connections.

2.2 Software

A TPC-W implementation requires three software modules to support the SUT and drive the benchmark: a database server, an application server, and an image server. The implementation in this research uses PostGreSQL 8.0.4 as the database server. The front-end uses Apache Jakarta Tomcat 5.5.12 as a combined application server and image server. The Java code run by the application server to generate the web pages (and interface with the database) is derived from the code freely available from the University of Wisconsin PHARM project [5]. This code implements both the servlets and a Java RBE, which is used to run the benchmark. Slight modifications are necessary to work with Tomcat and PostGreSQL [22]. The servlets are modified to make use of limited caching of data and to use a limited number of connections to the database. The number of connections is controlled by a Java semaphore. Because this semaphore is sometimes in heavy demand on the system, the option to force fairness is enabled. Finally, a new administrative servlet is implemented whose sole function is to change this semaphore to allow more or fewer connections to the database. The number of connections to the database is one reconfigurable option on the system.

Additionally, two modifications are made to the RBE. First, inabilities to connect to the front-end and connection timeouts are retried, rather than being treated as a fatal error. Also, the EB generation code is modified to support a combination of EBs running different mixes and to allow the workload to change during the measurement interval.

2.3 Hardware

The physical setup of the system uses 3 identical Dell Precision 360n machines. Each machine has a 2.8 GHz processor and 2 GB RAM. The machines are networked using built-in gigabit ethernet interfaces and a gigabit ethernet switch. As illustrated in Figure 1, one machine acts as the back-end database machine, one machine is the front-end web server, and one machine drives the benchmark by hosting the RBE. Additionally, this machine hosting the RBE also runs the tuning agent. In a true reconfigurable system, this agent would likely run on an independent system.

Though these computers are physically distinct in practice, they are meant to represent logical partitions of a single reconfigurable computer with a total of 2.8 GHz processing power and 2 GB RAM. To simulate partitioning of one such machine into a front-end and back-end machine, memory and CPU power are artificially constrained on each machine so that, overall, one full 2.8 GHz processor and 2GB of RAM are available for use by the front-end and back-end combined.

For example, when the front-end is allowed to use 1.8 GHz, the back-end has 1.0 GHz available. The processors are constrained through the use of a highly favored process that spins in a busy loop for a short, fixed period of time. By actively spinning for a given percentage of the time, only the remaining idle processor time can be used for benchmark-related work. For example, if the processor is actively spinning for 75% of the available cycles, this simulates a 0.7 GHz machine. In order to avoid overly bursty performance, the process yields the processor at least once every 160 ms.

Similarly, memory is constrained by using the Linux *mlock()* subroutine to pin a certain percentage of memory. A separate process is run that allocates and pins a configurable amount of memory. When this process is told to pin 1.5 GB of memory, this simulates a machine with only 0.5 GB of memory available for use. This implementation is similar to VMware ESX Server's *ballooning* technique [32], although the physical memory is merely held in our case, as we are not operating on virtual machines.

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, if specified. 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.

While true reconfigurable systems often have a dedicated connection of some sort (parallel, serial, dedicated ethernet, etc.) between the tuning agent and the system [12], ours is restricted to communication over the shared ethernet. On a heavily loaded system, this restriction can delay the reconfiguration tasks significantly. To avoid this delay, the tuning agent maintains a persistent connection to each machine. This connection is "niced" to a highly favored priority (although not as favored as the CPU constraining process), so that reconfiguration can take precedence over the database and web server.

There are two types of tuning agents we simulate. The first is an omniscient agent. This version of the agent is told both the configuration to switch to and when the reconfiguration should occur based on complete knowledge of when the workloads will switch. At each workload switch, the configuration that is better for that type of workload is used. The omniscient agent is used to obtain upper bound results when there is a known set of configurations that will maximize the performance of the system, thus allowing us to have a known maximum 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 relatively predictable workload is a fairly common and well-understood issue [9, 20], these static configurations can perform very poorly under a completely different workload. For example, we consider a small local bookstore that has recently started taking orders online. The website would likely see a set of regular customers shopping online, and a large number of customers browsing the site, but not yet ready to place orders online. These browsing users would be primarily served from cached data, and there would be less resources needed for the database. As this would be the expected audience for the site, the hardware would be configured to give only enough resources to the database to handle the regular customers, while assigning the bulk of the resources to handle the webserver. However, as customers become more comfortable with the site, they are likely to start ordering books through the system. Over time, the workload will gradually change from a primarily browsing site to have more ordering users. As this happens, there will be more load on the database and the site's throughput will fall as the database system is starved for resources. In this situation, the machine has gone from being properly configured to being poorly configured due to the change in workload.

In addition to such gradual changes, sudden, drastic changes in workload are also possible and must be handled quickly. There is a common phenomenon known as the "Slashdot effect" [33]. In this situation, an established but often little-known site is featured on a news site such as Slashdot [26]. This site is then inundated by users linking from the news article. If the site is an online store with a set of loyal customers regularly ordering products, it would be

configured for that expected workload. The large number of users appearing, who often have no intention of ordering any products, can overwhelm sites that are not prepared for this unexpected change in workload. We call this situation a *workload spike*.

Unfortunately, in practice, the workload is often not an observable quantity to the system. It is possible to instrument the system in various ways to help observe part or all of the workload. For example, instrumentation could be added to the middleware to indicate which web pages are being fetched in real-time. The percentage of each page would be an indicator of the workload. Alternately, the system could be modified to analyze the TCP/IP stream flowing between the two systems, or between the webserver and the users. Statistics output from this analyzer could form an interpretation of the workload

However, previous work [34] has shown that low-level operating system statistics can be used instead to help analyze the workload and suggest what configuration should be used. These system statistics have the advantage of not requiring any instrumentation to be added to the middleware. By using out-of-the-box versions of middleware, the system allows for easy replacement of any part of the system without the need for significant reimplementation. Low-level statistics 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 system 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.

The statistics reported by *vmstat* can be found in Table 2. In order to make this more configuration-independent, the memory statistics are converted to percentages. The resulting vector of statistics, 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.

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: Statistics reported by *vmstat*

For our experiments, we consider two possible system configurations. The first gives the bulk of the resources on the system to the database machine. In this configuration, the database receives $\frac{13}{16}$ of the CPU time and $\frac{5}{8}$ of the memory, leaving under only $\frac{3}{16}$ of the CPU and $\frac{3}{8}$ of the memory to handle all the web server activity. In this configuration, the system is allowed 1000 connections between the web server and the database. This configuration is ideal for the ordering intensive workloads.

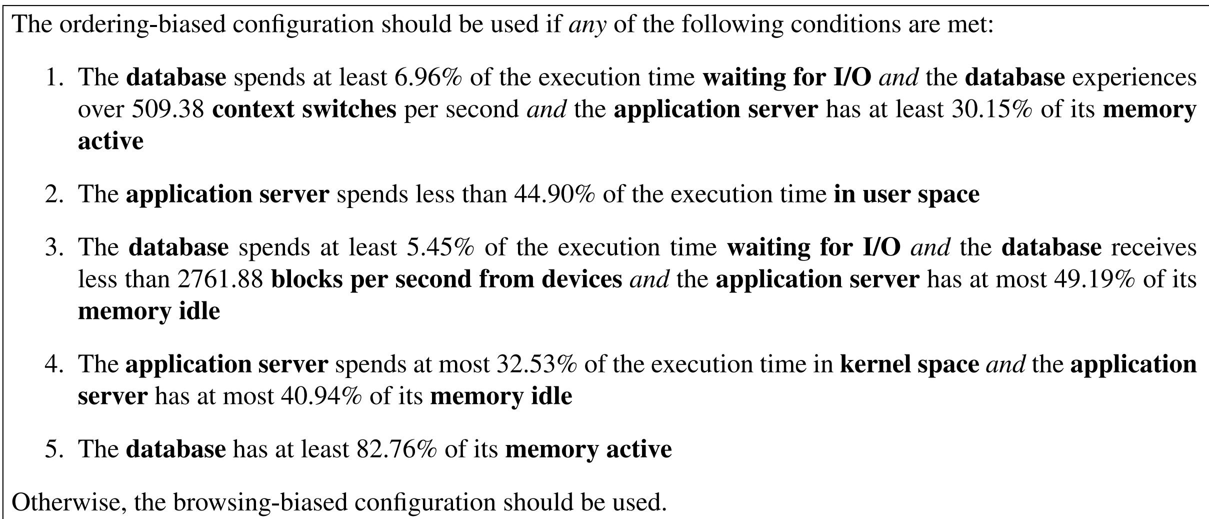
The other configuration is designed to handle the browsing intensive workloads, giving $\frac{5}{16}$ of the CPU to the web server and $\frac{5}{8}$ of the memory. The database still retains $\frac{11}{16}$ of the CPU and $\frac{3}{8}$ of the memory. Although this appears to be a large subset of resources to give to the database for a browsing workload, many of the web pages are required to access the database, even if only to query data about an item. In this configuration, the system is also restricted to 400 connections between the web server and the database. A wide range of possible connection limits was investigated, and the selected numbers maximize the throughput for the given configurations.

In order to acquire training data, a series of 100 pairs of varied TPC-W runs are done. Each pair of runs uses a workload that consists of 500 shopping users and a random number of either browsing or ordering users (50 pairs

each). The random number ranges from 500 to 1000. The runs have 240 seconds of ramp-up time, followed by 400 seconds during which the WIPS are measured. During this measurement interval, 200 seconds of *v*mstat data was also collected. Each pair of runs consists of one run on each configuration.

From the combinations of WIPS on each run and the 200 seconds of system statistics, a set of training data points are created in the following way. First, the WIPS from the two runs are compared, and the configuration with the higher throughput is determined to be the preferred configuration for all data points generated from this pair. Then, each collection of system statistics is divided into non-overlapping 30 second intervals. Each interval is averaged to generate the system statistics for one data point. In this way, each of the 100 pairs of runs generates 12 data points. This training data is more representative of real world data than we had previously investigated; our previous work used only a fixed set of known workloads for training and testing.

Given training data, the WEKA [35] 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 [7] rule learner is applied to the training data. For the generated data, JRip learned the rules shown in Figure 2.



The ordering-biased configuration should be used if *any* of the following conditions are met:

1. The **database** spends at least 6.96% of the execution time **waiting for I/O** *and* the **database** experiences over 509.38 **context switches** per second *and* the **application server** has at least 30.15% of its **memory active**

2. The **application server** spends less than 44.90% of the execution time **in user space**

3. The **database** spends at least 5.45% of the execution time **waiting for I/O** *and* the **database** receives less than 2761.88 **blocks per second from devices** *and* the **application server** has at most 49.19% of its **memory idle**

4. The **application server** spends at most 32.53% of the execution time in **kernel space** *and* the **application server** has at most 40.94% of its **memory idle**

5. The **database** has at least 82.76% of its **memory active**

Otherwise, the browsing-biased configuration should be used.

Figure 2: JRip rules learned by WEKA

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. Of 32 system statistics supplied total, it determined that there were 8 of importance in making decisions. These 8 are evenly split over the front- and back-end machines; however, we can see that all of the front-end statistics are either related to where execution time is being used and how well memory is being used. The back end statistics, cover not only those two categories, but also cover context switches and device I/O. 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.

Another trend we can identify (from rules 2 and 4) is that there appears to be a proper balance of execution time on the application server. Rule 2 works with a lower threshold on the amount of time spent in user space, while rule 4 sets an upper bound on the amount of time spent in the kernel. While these would seem unrelated to the database needing more resources, a properly balanced system would have the front-end make a request to the database (which would spend a small amount of time in the kernel creating the socket, sending the request, and receiving the response), and would then resume execution in user space. If the application server is not spending that requisite time in user space

(or is spending an excessive quantity of time in the kernel), it stands to reason that there is a problem establishing the connections and getting the responses. The logical conclusion of this is that the database is not able to quickly handle the incoming requests.

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.

To verify our learned model, we first evaluate the performance of JRip's rules using stratified 10-fold cross validation, in which results are averaged over 10 separate trials where the learning algorithm is training on 90% of the data and tested on the remaining 10% held out as independent test cases. 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 (each set having 120 data points). The 10 trials each used a distinct partition as the test set, while training on the remaining 9 partitions. In this test, JRip correctly predicts the better target configuration 98.25% of the time.

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 are more necessary to adapt quickly to than a gradual change.

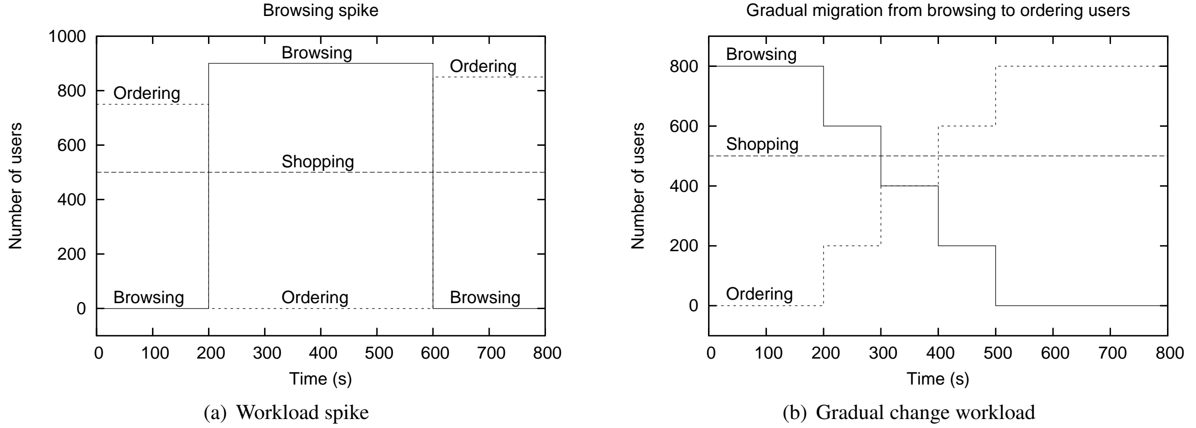
In order to simulate workload spikes, we warm the system up with one workload. Once that workload reaches steady state for a period of time, we change the workload suddenly to a different mix of users. After a period of time we return to a configuration similar to but not necessarily identical to our start configuration. We test both browsing and ordering spikes; for simplicity, we will only describe the browsing spike below. The base workload consists of 500 shopping users with a random number (between 500 and 1000) of ordering users. This workload reaches the steady state (240 seconds of ramp-up time), and then there are 200 seconds of measurement time, as before. At that point, the workload abruptly changes to a browsing intensive workload, with between 500 and 1000 users. Any available ordering users are converted to the new mix, while any surplus users are terminated. If more users are needed, they are spawned immediately.

The spike continues for 400 seconds. After this time, the workload abruptly reverts to an ordering intensive workload, with a newly generated random number of additional users. After 200 more seconds, the measurement interval ends. A diagram of the browsing spike workload can be found in Figure 3(a).

We also want to simulate a gradual change. For this, we begin with a workload that consists of 500 shopping users and some large number of browsing users. We allow this workload to reach a steady state by letting it run for 240 seconds of ramp-up time, followed by 200 seconds of measurement time. We then convert these users to the ordering mix in quarters, allowing 100 seconds of time between each switch. Finally, when all the users are in the ordering mix, we allow them to run for 300 more seconds of measurement time. We convert from browsing users to ordering users in keeping with our motivating example. A diagram of the workload can be found in Figure 3(b).

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.



(a) Workload spike

(b) Gradual change workload

Figure 3: Examples of tested workloads

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 agent never loses significantly to either static configuration. The random workloads and average WIPS results for the ordering spike trials can be found in Table 3 and results for the browsing spike trials are in Table 4.

Number of users in phase			Configuration			
1 (browsing)	2 (ordering)	3 (browsing)	browsing	ordering	adaptive	omniscient
652	598	669	67.8	69.3	**77.9**	77.9
960	871	991	68.8	69.5	**75.1**	75.4
959	595	780	71.2	69.9	**78.7**	78.2
984	973	682	69.0	68.2	**74.6**	75.9
781	778	989	69.1	68.8	**78.5**	77.8

Table 3: WIPS results for ordering spike workloads in various configurations. Bold numbers indicate the best non-omniscient result in each row. Numbers in parentheses show no significant difference to the best result; italicized numbers indicate situations where the omniscient agent was significantly better.

In addition to performing well as compared to static configurations, we can see that 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[3]. One typical example of the throughputs of each of the four configurations on an ordering spike can be seen in Figure 4.

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, we see that the agent actually wins significantly over both configurations. However, the agent has more room for improvement than in the ordering spike

[3]We theorize that this is likely due to random variance.

Number of users in phase			Configuration			
1 (ordering)	2 (browsing)	3 (ordering)	browsing	ordering	adaptive	omniscient
800	837	916	65.1	69.5	**71.2**	70.8
762	942	895	65.3	69.3	**72.7**	73.3
951	915	774	64.2	**69.4**	(69.0)	70.5
711	933	830	64.9	(70.0)	**71.5**	73.8
631	891	671	63.7	**69.4**	(69.2)	72.0

Table 4: WIPS results for browsing spike workloads in various configurations. Bold numbers indicate the best non-omniscient result in each row. Numbers in parentheses show no significant difference to the best result; italicized numbers indicate situations where the omniscient agent was significantly better.

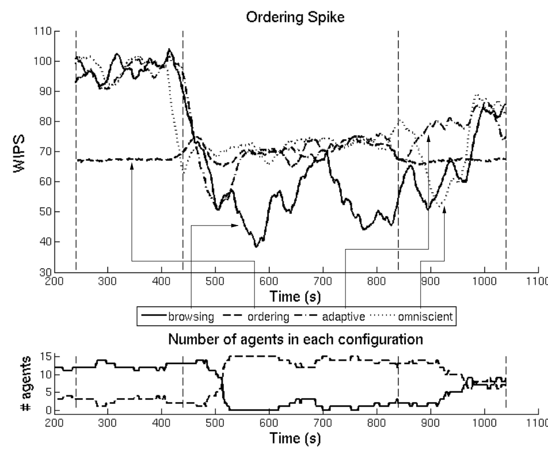


Figure 4: WIPS throughput for each of the configurations during an ordering spike, averaged over 15 runs. The graph at the bottom shows how many learning agents had chosen each configuration at a given time; these choices are all averaged in the "adaptive" graph.

tests; on average, the learning agent is 1.3 WIPS below the omniscient agent. An example of the throughputs of each of the four configurations on a browsing spike can be seen in Figure 5.

In both examples, we see some sudden, anomalous drops (at approximately 900 seconds in Figure 4 and approximately 650 and 850 seconds in Figure 5). These drops can confuse the agent; in Figure 5, we see that, until the anomaly, the agents are changing from the ordering to the browsing configuration during the spike. After the anomaly, the agents briefly begin reverting back to the ordering configuration. The cause of this anomaly has been identified[4]; future work will eliminate this slight confusion to the agent.

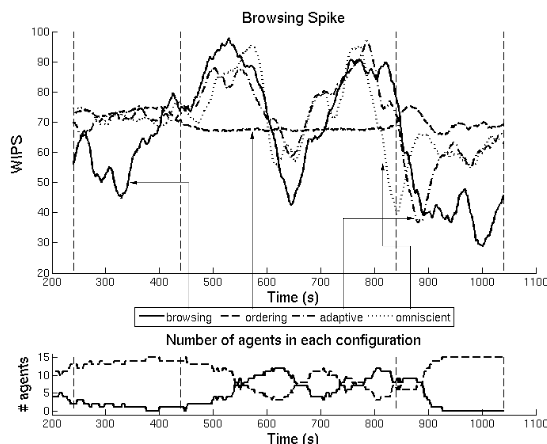


Figure 5: WIPS throughput for each of the configurations during a browsing spike, averaged over 15 runs. The graph at the bottom shows how many learning agents had chosen each configuration at a given time.

In addition to spikes of activity, we test gradual changes in workload to verify that the agent is capable of handling gradual changes. For this test, 800 browsing users become ordering users as detailed in Section 3.2[5]. 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. However, 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. The average throughput over time for this test can be seen in Figure 6.

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

The concept of 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 (and certain 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.

[4]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

[5]We also performed the ordering-to-browsing test, but due to severe under-performance of the browsing intensive configuration, the "correct" configuration to choose was the ordering-intensive one. Our agent chose that configuration almost all the time, resulting in a throughput that was indistinguishable.

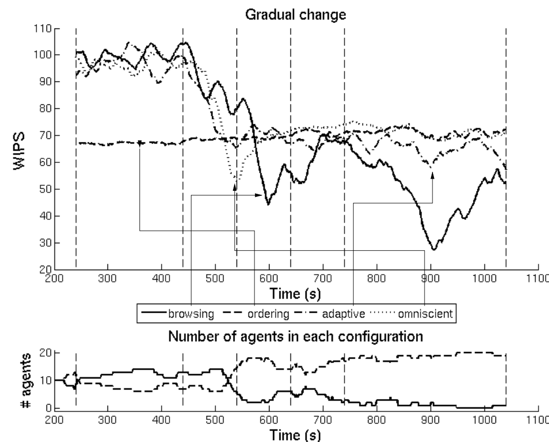


Figure 6: WIPS throughput for each of the configurations during a gradual change of user mix, averaged over 10 runs. The graph at the bottom shows how many learning agents had chosen each configuration at a given time.

5.1 Model-based and control-theoretic approaches

Menascé et al. [18] discuss self-tuning of run-time parameters using a model based approach; in this work, the parameters concern the numbers of threads and connections allowed to the webserver. The authors suggest that a similar method should be extensible to hardware tuning as well. A hill-climbing algorithm is used to tune the parameters for the currently observed workload. This requires constructing a detailed mathematical model of the system; our work treats the system as a black box. Additionally, this work uses the current workload as an input to the performance model, whereas we treat the workload as an unobservable quantity.

Urgaonkar et al. [31] use a queuing model to assist in provisioning a multi-tier Internet application. While the end goal is similar to our goal in this paper, there are two important differences in this approach. First, this approach is fundamentally intended to handle multiple distinct servers at each tier, whereas our approach is intended to have just one server with a variable amount of processing power. Additionally, this system assumes that there are unused machines available for later provisioning, while we assume that the system is limited in resources in order to avoid overprovisioning.

Aiber et al. [3] investigate optimization of a system according to high-level business objectives. In order to accomplish this, a model is built using general-purpose, predefined building blocks. Performance improvement is largely done through optimization of the individual pieces of the model. This requires modeling the system, which we do not need. Additionally, this work focuses more on different treatments of different SLAs (with corresponding application stacks), whereas we consider that there is only one application on the server, and a non-formal SLA.

Kandasamy et al. [15] use a control-theoretic approach to self-optimizing a system. In their situation, the goal is not to maximize performance overall, but to minimize power consumption. This involves prediction of the future workload and using that prediction to estimate the optimal frequency to run the processor at. We consider cases where the server is likely to be overwhelmed, and, as such, the processor will always run at full power.

Karlsson and Covell [16] propose a system for estimating a performance model while considering the system as a black box. They use a Recursive Least-Squares Estimation approach to attempt to model the performance of the system without knowing the details of its internals, with the intention that this model can be used as part of a self-tuning regulator. While this approach appears to help meet an SLA goal (using latency as the metric), it does not aim to maximize the performance; rather it tries to get as close as possible to the SLA requirement without exceeding it.

5.2 Performance management of a single system

Strickland et al. [27] propose a system for allowing idle processor cycles to be "scavenged" for other work (such as SETI@Home [25]), while also allowing these scavengers to be throttled in order to minimize impact on performance.

This work assumes that there is such idle time; the performance arena we are focusing on is unlikely to have substantial idle processor cycles.

Waldspurger [32] investigated resource management as pertains to memory allocations among virtual machines. He identifies various ways in which memory use can be maximized, including reclaiming unused memory from some machines and sharing memory among machines. However, all resource management is based on static rules and no effort is made to learn or predict memory requirements.

Diao et al. [8] analyze how to set certain parameters of the Apache web server in order to keep CPU and memory usage near a pre-set parameter. The authors make the assumption that there is an optimal setting for those parameters, whereas we have shown that optimal resource usage is dependent on workload.

Gomez et al. [11] use neuroevolution learning methods to dynamically reallocate hardware resources for high processor performance. Their learning is at an intra-chip level, intending to maximize the performance of an individual processor, while our learning is on a system level and is intended to maximize the performance of the entire distributed system.

5.3 Performance management of a datacenter

Norris et al. [19] 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.

Mahabhashyam and Gautam [17] discuss the issue of providing Quality of Service (QoS) guarantees through dynamic resource allocation. Their work is primarily geared toward situations where there are multiple classes of requests, each with a separate QoS requirement. In contrast, we only consider one class of requests (all users are equally important).

5.4 Other related work

Other prior work addresses performance-tuning with the intention of maintaining a fixed level of service, as opposed to maximizing throughput. For example, Abdelzaher et al. [1] outline a system that maintains multiple complete content trees, each with a different quality setting. As workload increases, quality can be decreased in order to satisfy the maximum number of users. Roşu et al [24] discuss automatic hardware allocation in real-time systems by analyzing resource requirements.

Cohen et al. [6] 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.

Additional work has also been done with offline analysis of a system. Aguilera et al. [2] works on discovering the latency of individual nodes in a network. This work is done with no additional instrumentation to the system and is designed to determine the causal paths through the network and analyze the total response time as to how much time was spent in each node. Hellerstien et al. [13] analyzes the performance of a system over large spans of time with statistical models, which could then determine online when unexpected changes occur. Brown et al. [4] analyzes the dependency of each page in the TPC-W benchmark on each database table. While these approaches could be used as a basis for online configuration, to the best of our knowledge such steps have not been taken.

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 makes two main contributions. First, we introduce 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 situations.

Second, we present and implement a framework for testing reconfiguration agents online. While we present just two agents, the framework is capable of handling any agent desired, from an omniscient agent to a simple heuristic agent, to an advanced learning agent. Thus, it allows different agents to be compared to each other easily. Additionally, as no instrumentation of the middleware is needed, individual components can be easily replaced (for example, using MySQL instead of PostGreSQL), or the entire system could be changed (such as using an implementation of Sun's PetStore [21] web-based store simulation). Although the learner would need retraining, possibly with different base configurations, the framework would not need any modifications.

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 learn how to predict the benefit of a reconfiguration, both on our simulated reconfigurable machines and eventually on true reconfigurable hardware.

Acknowledgments

The authors thank Ray Mooney, Hany Ramadan, and Jungwoo Ha for valuable input throughout the research and Ray Mooney and Shimon Whiteson for 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

[1] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1), January 2002.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems*, October 2003.

[3] S. Aiber, D. Gilat, A. Landau, N. Razinkov, A. Sela, and S. Wasserkrug. Autonomic self-optimization according to business objectives. In *Proceedings of the 1st International Conference on Autonomic Computing*, pages 206–213, New York, NY, May 2004.

[4] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed application environment. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, May 2001.

[5] 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. Code available at `http://www.ece.wisc.edu/~pharm/tpcw.shtml`.

[6] 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.

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

[8] Y. Diao, J. L. Hellerstein, S. Parekh, and J. Bigus. Managing web server performance with autotune agents. *IBM Systems Journal*, 42(1), 2003.

[9] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, and M. Woodside. Performance analysis of distributed server systems. In *Proceedings of the 6th International Conference on Software Quality*, pages 15–26, 1996.

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

[11] F. Gomez, D. Burger, and R. Miikkulainen. A neuroevolution method for dynamic resource allocation on a chip multiprocessor. In *Proceedings of the INNS-IEEE International Joint Conference on Neural Networks*, pages 2355–2361. IEEE, 2001.

[12] Hardware management console for pseries operations guide. International Business Machines Corporation, November 2003. `http://publib16.boulder.ibm.com/pseries/en_US/infocenter/base/hardware_docs/pdf/380590.pdf`.

[13] J. L. Hellerstein, F. Zhang, and P. Shahabuddin. Characterizing normal operation of a web server: Application to workload forecasting and problem detection. In *Proceedings of the Computer Measurement Group*, 1998.

[14] hp-ux virtual partitions (vPars). Hewlitt-Packard Company, January 2003. `http://www.hp.com/products1/unix/operating/manageability/partitions/library/vpars_wp203.pdf`.

[15] N. Kandasamy, S. Abdelwahed, and J. P. Hayes. Self-optimization in computer systems via online control: Application to power management. In *Proceedings of the 1st International Conference on Autonomic Computing*, pages 54–61, New York, NY, May 2004.

[16] 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, Seattle, WA, June 2005.

[17] S. R. Mahabhashyam and N. Gautam. Dynamic resource allocation of shared data centers supporting multiclass requests. In *Proceedings of the 1st International Conference on Autonomic Computing*, pages 222–229, New York, NY, May 2004.

[18] 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.

[19] 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.

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

[21] Java pet store demo. Sun Microsystems, Inc. http://developer.java.sun.com/developer/releases/petstore/.

[22] C. Plattner. Getting java tpc-w to work with postgresql and tomcat. http://www.inf.ethz.ch/personal/plattner/work/tpcw-postgresql.html.

[23] D. Quintero, Z. Borgosz, W. Koh, J. Lee, and L. Niesz. Introduction to pSeries partitioning. International Business Machines Corporation, November 2004. http://www.redbooks.ibm.com/redbooks/pdfs/sg246389.pdf.

[24] D. I. Roşu, K. Schwan, S. Yalamanchili, and R. Jha. On adaptive resource allocation for complex real-time applications. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 320–329, December 1997.

[25] SETI@Home. http://setiathome.ssl.berkeley.edu/.

[26] Slashdot. http://www.slashdot.com.

[27] J. W. Strickland, V. W. Freeh, X. Ma, and sudharshan S Vazhkudai. Governor: Autonomic throttling for aggressive idle resource scavenging. In *Proceedings of the 2nd International Conference on Autonomic Computing*, pages 64–75, Seattle, WA, June 2005.

[28] Sun Enterprise™ 10000 server: Dynamic system domains. Sun Microsystems, Inc., February 1999. http://www.sun.com/datacenter/docs/domainswp.pdf.

[29] TPC Benchmark™W Full Disclosure Report for IBM eServer xSeries 440 with IBM eServer xSeries 330 using Microsoft SQL Server 2000 Enterprise Edition. Transaction Processing Performance Council, December 2002. http://www.tpc.org/results/FDR/tpcw/ibm.x440.w.fdr.02091201.pdf.

[30] Transaction Processing Performance Council. *TPC Benchmark™ W (Web Commerce) Specification*, February 2002. Version 1.8.

[31] 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, Seattle, WA, June 2005.

[32] C. A. Waldspurger. Memory resource management in VMware ESX Server. In *Proceedings of the 5th Symposisum on Operating Systems Design and Implementation*, December 2002.

[33] Wikipedia: Slashdot effect. http://en.wikipedia.org/wiki/Slashdot_effect.

[34] 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, Seattle, WA, June 2005.

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