Machine Learning for On-Line Hardware Reconfiguration

Keywords: Adaptive systems, Autonomous agents, Applications

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

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 [Quintero *et al.*, 2004]. 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 [Withheld, 2005], 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 uses a learned model to establish just that. Specifically, we train a robust learned model of the predicted performance of different hardware configuations, and then use this model to guide an online agent to perform 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 [Quintero *et al.*, 2004]. 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 [Quintero *et al.*, 2004].

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 gives a high-level overview of the testbed setup. A brief description 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. Further testbed and system details can be found in [Withheld, 2006].

2.1 TPC-W

The TPC-W Benchmark 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.

There are a total of 14 dynamically generated web pages that can be retrieved. These pages are divided into six browsing pages and eight ordering pages. 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 response time requirement. There are three defined workloads, called *mixes*: the *shopping* mix, the *browsing* mix, and the *ordering* mix. The differences between these mixes is 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 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 during the measurement interval.

Commercially built and tested systems that publish performance results often have large numbers of machines in complex setups. 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. Although our system produces WIPS numbers, reported in Section 4 that are significantly less than on such a commercial system (due largely to the corresponding difference in processing power and memory), they are not out of the ordinary for experimental systems.

[1] TPC-W has recently been replaced by TPC-App, previously known as TPC-W version 2.

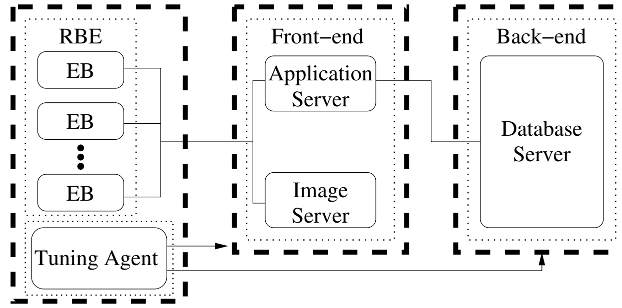


Figure 1: The 3 machines used in the physical setup.

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.

Some other specifications that are modified are:

- *Relationship between the number of EBs and the reported WIPS*: We run substantially more EBs than permitted with the intention of overloading the system.
- *Initial population of the database*: We use a small initial dataset to introduce contention in the database.
- *Web Interaction Response Time (WIRT) constraints are ignored*: Because we are attempting to maximize throughput, we do not want to ignore results that violate the constraints; 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 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 [Cain *et al.*, 2001]. 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. 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, CPU power is artificially constrained using a highly favored, periodic busy-loop process. Similarly, memory is constrained through a process using the Linux *mlock()* subroutine. Constraints are specified such 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.

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, 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 favored, persistent connection to each machine, 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, 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 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 [Oslake *et al.*, 1999], these static configurations can perform very poorly under a completely different workload. For example, there is a common phenomenon known as the "Slashdot effect". In this situation, an established but often little-known site is featured on a news site such as Slashdot. This site is then inundated by users linking from the news article. We consider the situation where this site is an online store with a set of loyal customers regularly ordering products. As such, 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 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 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 [Withheld, 2005] 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 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.

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

Table 2: Configurations used in the experiments

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 are 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 [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. 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. Other trends are discussed in [Withheld, 2006].

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 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 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

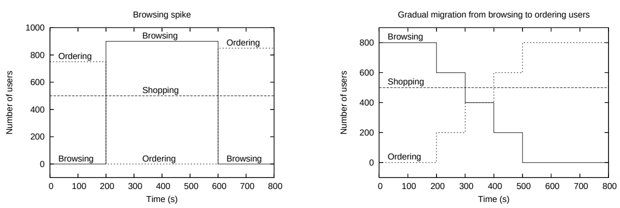
We present two types of workload changes: the gradual change and the workload spike. We concentrate our evalu-

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



(a) Workload spike

(b) Gradual change workload

Figure 3: Examples of tested workloads

ation on the workload spike, as sudden changes in workload are more necessary to adapt quickly to than a gradual change.

For simplicity, we describe only the simulation of a 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. At that point, the workload abruptly changes to a browsing intensive workload, with between 500 and 1000 users, spawning or terminating users as needed. 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 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, the workload begins changing through a step method. Users are converted 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. 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.

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. Average results can be found in Table 3

	Configuration			
Type of Spike	adapt.	browsing	ordering	omnisc.
average ordering	77.0	69.2	69.1	77.0
single ordering	74.6	69.0	68.2	75.9
average browsing	70.7	64.6	69.5	72.1
single browsing	72.7	65.3	69.3	73.3

Table 3: Results (in WIPS) of different configurations running different spike workloads

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[2]. One typical example of the throughputs of each of the four configurations on an ordering spike can be seen in Figure 4.

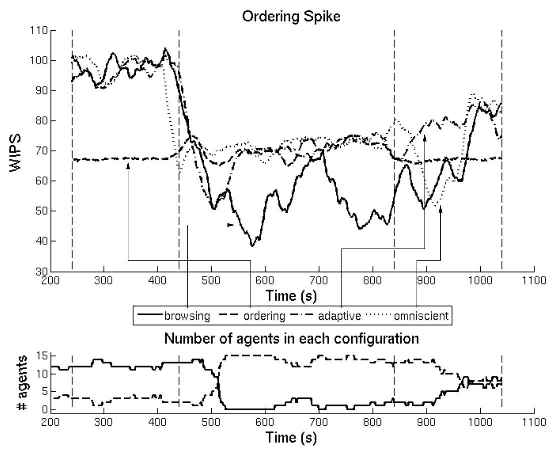


Figure 4: WIPS throughput for each of the configurations during an ordering spike, averaged over 15 runs. The graph at the bottom shows how many learning agents had chosen each configuration at a given time; these choices are all averaged in the "adaptive" graph.

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 (at approximately 900 seconds in Figure 4). These drops can sometimes confuse the agent. The cause of this anomaly has been identified[3]; future work will eliminate this slight confusion to the agent.

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[4]. 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.

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.

Menascé et al. [2001] 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. 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.

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. 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.

Urgaonkar et al. [2005] use a queuing model to assist in provisioning a multi-tier Internet application. 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 work 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.

Waldspurger [2002] 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 ma-

[2]Presumably due to measurement noise.

[3]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

[4]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.

chines 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.

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.

Mahabhashyam and Gautam [2004] 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).

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

References

[Cain *et al.*, 2001] Harold W. Cain, Ravi Rajwar, Morris Marden, and Mikko 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`.

[Cohen, 1995] William 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] Daniel F. Garcia and Javier Garcia. TPC-W e-commerce benchmark evaluation. *Computer*, 36(2):42–48, February 2003.

[Karlsson and Covell, 2005] Magnus Karlsson and Michele 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.

[Mahabhashyam and Gautam, 2004] Sai Rajesh Mahabhashyam and Natarajan 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.

[Menascé *et al.*, 2001] Daniel A. Menascé, Daniel Barbará, and Ronald 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] James Norris, Kieth Coleman, Armando Fox, and Gerge 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.

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

[Quintero *et al.*, 2004] Dino Quintero, Zbigniew Borgosz, WooSeok Koh, James Lee, and Laszlo Niesz. Introduction to pSeries partitioning. International Business Machines Corporation, November 2004. `http://www.redbooks.ibm.com/redbooks/pdfs/sg246389.pdf`.

[Urgaonkar *et al.*, 2005] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, and Pawan 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.

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

[Withheld, 2005] Author Withheld. 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.

[Withheld, 2006] Author Withheld. Adapting to workload changes through on-the-fly reconfiguration. Technical report, University of —-, 2006.

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